REGISTER FILE ARCHITECTURE OPTIMZATION IN A COARSE-GRAINED RECONFIGURABLE ARRAY

by

Zion Siu-On Kwok
B.A.Sc., University of British Columbia, 2003

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

Master of Applied Science

in

The Faculty of Graduate Studies

Electrical and Computer Engineering

The University of British Columbia

April 2005

ABSTRACT

REGISTER FILE ARCHITECTURE OPTIMIZATION IN A COARSE-GRAINED RECONFIGURABLE ARRAY

This thesis investigates the impact of the global and local register file architecture on a reconfigurable system based on the ADRES architecture. The register files consume a significant amount of area on the reconfigurable device, and their architecture has a strong impact on the performance. We found that the global registers should be tightly connected to as many functional units as possible, while the connection of the local register files to their neighbours is less critical. We found that the global register file should contain 14 registers, while each local register file should only contain two registers. We used these results to propose two new architectures that demonstrate between -33% and 383% higher instructions per cycle per unit area compared to the original 4x4 and 8x8 array architectures, with 56% and 88% average improvement over a set of benchmarks for the new 4x4 and 8x8 array architectures, respectively.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ALU Arithmetic Logic Unit

ASIC Application-Specific Integrated Circuit

BIST Built-In Self-Test

CAD Computer-Aided Design

CLB Configurable Logic Block

DMA Direct Memory Access

DSP Digital Signal Processing

FIFO First-In First-Out

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

FU Functional Unit

IDCT Inverse Discrete Cosine Transform

II Iteration Interval

ILP Instruction-Level Parallelism

IMEC Interuniversity MicroElectronics Centre

IP Intellectual Property

IPC Instructions Per Cycle

LLP Loop-Level Parallelism

LUT Look-Up Table

MIPS Million Instructions Per Second (processor)

MPEG Moving Picture Experts Group

RAM Random-Access Memory

RC Reconfigurable Cell

RISC Reduced Instruction Set Computer

RTL Register-Transfer Level

SCRAP Steve's Coarse-grained Reconfigurable Architecture Program

SIMD single-instruction multiple-data

SRAM Static Random-Access Memory

VHDL Very high speed integrated circuits Hardware Description Language

VLIW Very Long Instruction Word (processor)

XML eXtensible Markup Language

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Dr. Steven Wilton for the topic, his support, his counsel, and just being such an understanding advisor who gives so much time and encouragement, as well as the occasional treat to his students. I also want to thank Dr. Guy Lemieux, with whom I have been privileged to work closely in EECE 583 and EECE 259, for sharing his experience.

I would also like to thank people I have looked up to: Julien Lamoureux, Ernie Lin, Kara Poon, Andy Yan, Noha Kafafi, James Wu, Martin Ma, and Victor Aken'Ova. They have introduced me to lab life, helped me get up on my feet, and made research something to look forward to. It has been a pleasure to be a part of the FPGA research group with Marvin Tom, Eddy Lee, Anthony Yu, Brad Quinton, Natalie Chan King Choy, Cary Feldstein, Scott Chin, and David Yeager. I've had good neighbours in the lab: Dipanjan Sengupta, Samad Sheikhaei, Zahra Ebadi, Cristian Grecu. Others who have enriched my experience in the SOC Lab include: Baosheng Wang, Andy Kuo, Jess Chia, Santosh Sood, Mohammad Hekmat, Neda Nouri, Amit Kedia, Peter Hallschmid, and Rod Foist. The friendly people and staff like Sandy, Roozbeh, and Roberto, have also made the lab a good place to work.

I am also grateful for Nelson Siu, Changsong Shen, and Mike Wang for welcoming me to their own corners of the world. Prayers and concern from my friends and family have been much appreciated. Lastly, I thank the Lord Jesus Christ for His grace and getting me through this time of learning and growth.

Chapter 1 INTRODUCTION

1.1 Motivation

Coarse-grained reconfigurable architectures promise high computing parallelism and low power consumption. Published architectures such as MorphoSys [1], PipeRench [2], and ADRES [3] demonstrate higher computing performance than general-purpose processors and very long instruction word (VLIW) processors, especially in loop-intensive digital signal processing (DSP) applications. Coarse-grained reconfigurable architectures are more suitable for applications requiring datapaths than field-programmable gate arrays (FPGAs), which are more effective for random logic [2]. Compared to fine-grained reconfigurable architectures and FPGAs in particular, coarse-grained reconfigurable architectures have larger reconfigurable functional units which require fewer programming bits and routing resources within each block, leading to power, area and delay reductions inside each functional unit. These power savings are important, especially in mobile systems.

As in any compute engine, the memory architecture has a significant effect on the performance of coarse-grained reconfigurable architectures [4]. MorphoSys [1] uses a frame buffer to stream image data. In REMARC [5], the MIPS microprocessor loads the data into the data register for the co-processor to access. The instruction memory is also significant; instructions can be stored in instruction registers, in context registers, or in random-access memories (RAMs) [5, 6, 7, 8, 9]. Furthermore, many coarse-grained architectures contain additional register files within the reconfigurable fabric for use as scratch-pad memory [1, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

One such coarse-grained reconfigurable architecture is ADRES [3]. ADRES, which is being developed at the Interuniversity Microelectronics Centre (IMEC) in Leuven, Belgium, contains both a general-purpose Very Long Instruction Word (VLIW) processor and a reconfigurable fabric. Estimates obtained using two area models described in this thesis will show that 36% to 62% of the area of the device is memory. Of this, just under half is devoted to register files used as scratch-pad memory and used to communicate between the VLIW processor and the reconfigurable fabric. Table 1.1 shows a breakdown of the area devoted to memory in our baseline ADRES-based architecture with 16 functional units and five contexts. Clearly, the area devoted to the register files is significant; hence, we would expect that the register file architecture would have a significant effect on the efficiency of the reconfigurable architecture. In this thesis, we show that this statement is true, and that we can design a significantly more efficient reconfigurable architecture by optimizing the register files.

Table 1.1: Typical Contributions of Various Memory Resources to Chip Area

Type of Memory		Area	Adjusted Area
Register Files	Data Global Register Files	10%	8%
	Predicate Global Register Files	2%	1%
	Data Local Register Files	13%	6%
	Predicate Local Register Files	2%	2%
FU Output Registers	FU Data Output Registers	3%	4%
	FU Predicate Output Registers	1%	1%
Context Memory	Input Multiplexer Context Memory	4%	2%
	Operation Code Memory	3%	2%
	Immediate Data Memory	23%	12%
Total		62%	36%

1.2 Research Goals

This research consists of an experimental analysis of the register file architecture in one representative reconfigurable processor, an architecture based on the ADRES architecture from IMEC [3], as shown in Figure 1.1. This architecture contains two types of register files: global register files that can be accessed by some (or all) of the reconfigurable functional units on the fabric, and local register files, each of which is associated with a specific reconfigurable functional unit. In particular, we ask three questions for each type of register file:

1) How big should each register file be?
2) How should the register file be connected to the rest of the fabric?
3) How many read and write ports should each register file have?

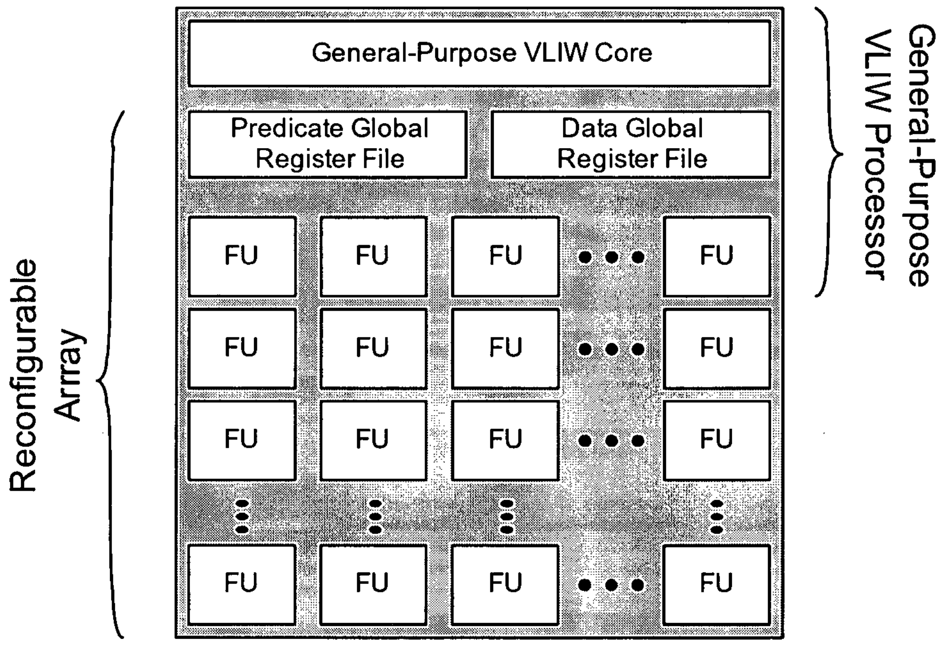

General-Purpose VLIW Core
Predicate Global Register File
Data Global Register File
FU
General-Purpose VLIW Processor
Reconfigurable Arrray

Figure 1.1: Baseline Architecture Based on ADRES [3]

We will then use these answers to construct a modified version of the ADRES architecture which has -7% to 38% smaller area and runs code between -29% and 200% faster than the original architecture. Although the numerical results are specific to this processor, the trends that we observe may apply to other reconfigurable systems which have global or local register files.

1.3 Significance

The reduction in area offers several benefits. Since this reduction in area means fewer transistors are required, we can expect a higher number of chips to fit on each silicon wafer, higher yield resulting from the rarer occurrence of manufacturing chips that have a physical defect, and potentially lower power consumption. The first two make the chip more economical. The latter makes the chip more attractive for applications where power is limited or expensive, such as mobile applications that are powered by a battery.

The performance gain that comes from an optimized register file architecture allows more applications to be executed on the reconfigurable system every second or more complex processing in real-time applications, where tight deadlines are imposed on each task.

1.4 Organization of this Thesis

The thesis is organized as follows. Background of the related research will be presented first in Chapter 2. In Chapter 3, we will describe our baseline architecture, shown in Figure 1.1, which is very similar to the ADRES architecture [3]. Chapter 4 will describe our experimental methodology and Chapter 5 will present experimental results illustrating the impact of various architectural parameters on both the area and performance of the device. Then in Chapter 6, we will use the results from Chapter 5 to propose two new architectures that are significantly smaller and faster than the baseline architecture. Finally, the conclusions, future work, and contributions are stated in Chapter 7.

1.5 Contributions of this Thesis

The contributions of this thesis are as follows:

1. We presented a methodology for evaluating ADRES-based reconfigurable architectures. This included the use of the IMPACT-I [73] and DRESC [74] compilers used for ADRES [3]. As well, the SCRAP tool used in [79] was enhanced to generate an XML architecture file, modularized VHDL code that included global register files, and a parameterized area model that could produce area estimates that correlate well with Synopsys post-synthesis area estimates, but runs in less than a second.
2. We performed experiments on the degree of connectivity of global and local register files to the functional units and found that shared connectivity of every functional unit to the global register files is vital to obtain high performance.
3. We verified that increasing the number of read and write ports on global and local register files improves performance, but at the cost of significant area.
4. We performed sweeps of the size of global and local register files and discovered in particular that only very few registers are needed in each local register file. This was attributed to the low utilization of local register files by the benchmark kernels.
5. We constructed enhanced 4x4 array and 8x8 array architectures that applied this new knowledge about register files to achieve on average 56% and 88% higher instructions per cycle per unit area, respectively, compared to the baseline architectures. For individual benchmarks, the enhanced 4x4 array architecture increased the instructions per

cycle per unit area by -6% to 152% and the enhanced 8x8 array architecture increased the instructions per cycle per unit area by -33% to 383%.

A portion of this work has been published in [16].

Chapter 2

BACKGROUND AND PREVIOUS WORK

In this chapter, the realm of reconfigurable custom computing is introduced with a particular emphasis on coarse-grained reconfigurable architectures.

2.1 Reconfigurable Custom Computing

Reconfigurable custom computers consist of programmable functional blocks that are connected by programmable routing, both of which are programmed using configuration bits [17]. They are reconfigurable because their hardware configuration can be modified and they perform custom computing because this flexible hardware configuration can be tailored to meet the needs of each specific application. This means that one such reconfigurable custom computer can give good performance, area, and power consumption for a wide variety of applications. Reconfigurable custom computing solutions possess some of the speed of dedicated hardware while also holding some of the programmability associated with software execution. According to Hartenstein [18], reconfigurable custom computing bridges the gap between the application-specific integrated circuits (ASICs) and microprocessors.

Today, general-purpose processors are very popular in desktop and laptop computers. However, based on audio, image, video, and cryptography application benchmarks, Lee et al. [19] showed that more than 90% of memory accesses on general-purpose processors can be considered overhead, which can be avoided in ASICs and systolic arrays, where data can be forwarded directly to perform the next operation. They also report potential instruction-level parallelism (ILP) from 6 to 558, opening up the possibility of significant speedups on certain applications. As well, the need to load instructions from memory, decode them, and then execute the instructions constitutes a significant overhead in processing time [17]. Moreover, sometimes the instruction set does not have a single instruction that performs the exact function that is required [17]. Multiple instructions may be required to implement a function that could be implemented as specialized logic that is much smaller and faster.

Custom computing refers to a class of systems that contains exactly the type of logic that is required for an application. For example, some applications require more on-chip memory than others, while others require multipliers or Ethernet controllers. Custom computing is efficient because there is no wasted functionality and complex functions that may require multiple instructions on a general-purpose processor can be implemented in hardware that is more optimized in terms of speed, area, and power consumption. ASICs are an example of custom computers, where the circuit is designed specifically to perform a certain task. This circuit can therefore be well-optimized in terms of speed, density, and power. However, it is expensive and time-consuming to design and manufacture an ASIC for every different task that must be performed. To layout and fabricate such a chip typically requires two to five months [20] and can cost one million dollars for the mask set alone which is used in fabrication.

Field-programmable gate arrays (FPGAs) are important to reconfigurable custom computing because they provide the flexible hardware building blocks that can be configured to perform different functions. As an array of these blocks, FPGAs can be configured to implement digital circuits for a diverse range of applications including network routers and robotics controllers.

Devices which can be used to implement reconfigurable custom computing include island-style FPGAs, FPGAs with fixed intellectual property (IP) logic cores, system-on-a-chip designs with embedded FPGA cores, fine-grained reconfigurable architectures, and coarse-grained reconfigurable architectures. Each of these will be discussed in the following subsections. Hard-wired FPGAs are also described because they fit nicely in the discussion about the tradeoffs between programmable and fixed logic. We also mention very long instruction word (VLIW) processors because of their similarity to some coarse-grained reconfigurable architectures. Coarse-grained reconfigurable architectures will be described further in Section 2.2 and form the basis of the discussions in the thesis.

2.1.1 Island-Style FPGAs

Field-programmable gate arrays (FPGAs) consist of an array of programmable logic blocks that are interconnected by programmable interconnect. The programmable logic blocks can be configured to perform many different digital logic functions. The programmable interconnect is controlled by routing switches that connect wires together to form paths between the logic blocks.

Designs using FPGAs enjoy short turn-around time, low non-recurring engineering cost, freedom from the complexities and risks of physical design, and flexibility to modify the functionality, making them favourable compared to ASICs in many situations. Where the turn-around time is vital, production volume is low, or where simplicity of design effort is desired, FPGAs are an attractive option. Their applications have expanded from simple glue logic to large stand-alone designs. The price for this flexibility is that they have lower performance, lower logic density, and higher power consumption than ASICs [20].

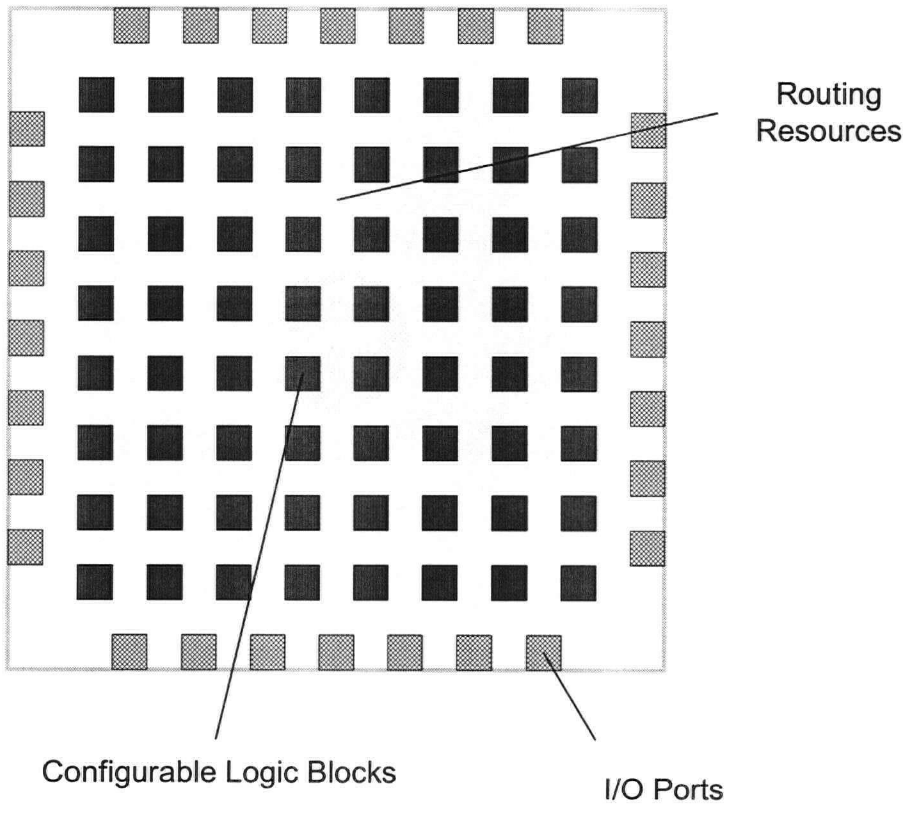


Figure 2.1: LUT-Based Island-Style Architecture

One popular architecture that has formed the basic pattern of early commercial FPGAs [21, 22] and has been the subject of much academic research [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35] is the *look-up table* (LUT)-based island-style FPGA architecture. The LUT-based island-style FPGA architecture consists of a matrix of configurable logic blocks (CLBs) arranged like islands in a sea of routing resources. Each CLB contains one or more LUTs, that act as programmable gates, and D flip-flops. The routing resources are organized in wires that form horizontal and vertical channels. These channels intersect at switch blocks and connect to the CLBs in connection blocks. Surrounding the entire matrix are input and output ports that interface the logic in the FPGA with the outside world. A typical island-style architecture is shown in Figure 2.1 and Figure 2.2.

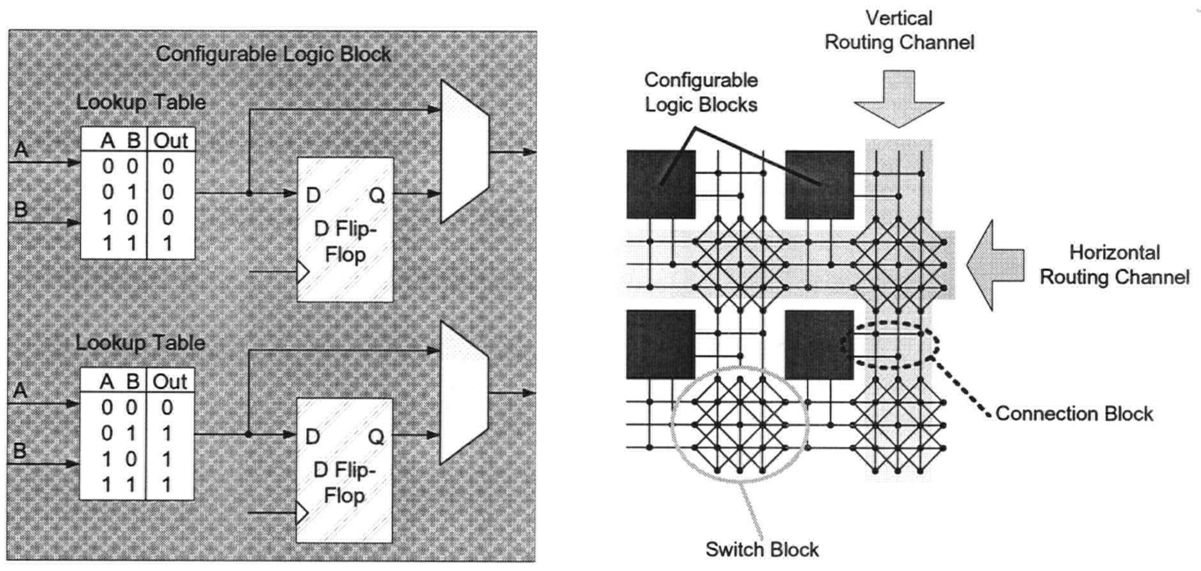


Figure 2.2: Details of the LUT-Based Island-Style FPGA

2.1.2 Addition of Fixed IP Cores

In addition to general-purpose logic resources, state-of-the-art commercial FPGAs such as Altera's Stratix-II [36] and Xilinx's Virtex-4 [37] now include adders, multipliers, memories, processors, clock management circuitry, and support for many I/O standards. These fixed intellectual property (IP) cores lead to significant increases to performance, savings in area, reductions in power, and simplifications of customer designs for applications in which these cores can be used. Figure 2.3 illustrates an FPGA with fixed logic cores.

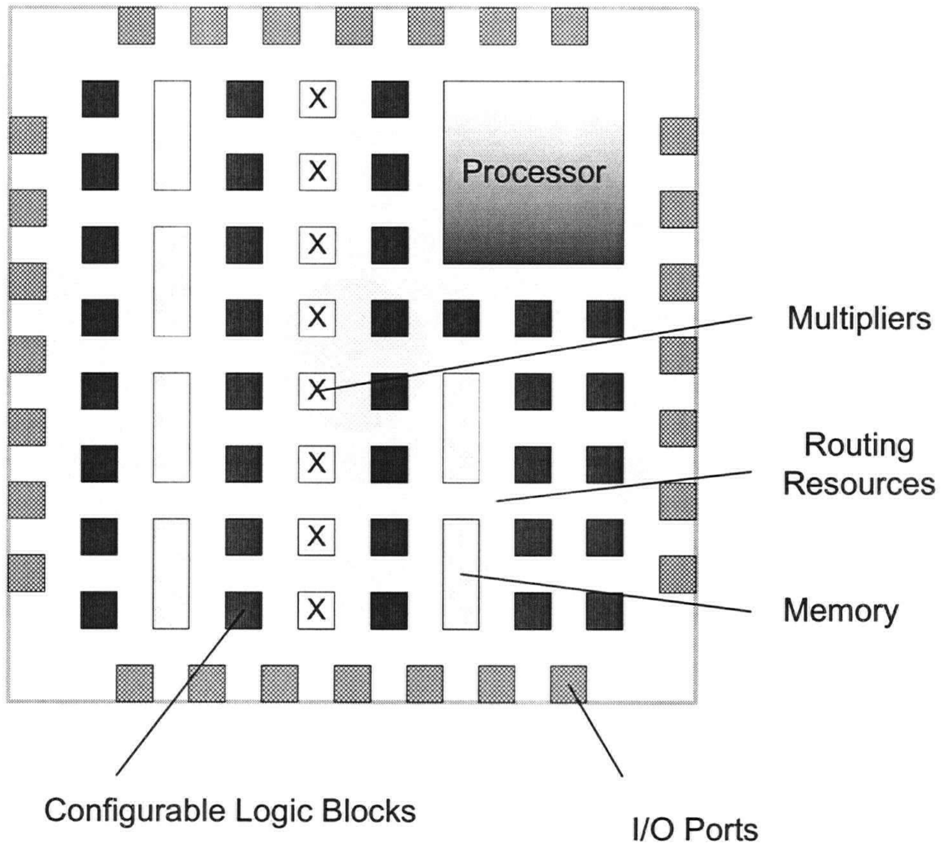


Figure 2.3: FPGA with Fixed IP Cores

2.1.3 Hard-Wired FPGAs

Even with these fixed cores, FPGAs still have lower performance, density, and power efficiency than ASICs. Performance and power efficiency can be improved by hard-wiring the routing on the FPGA [38]. The routing inside the CLB can be hard-wired as well, either using vias [38, 39] or two metal layers [40]. Designs destined for hard-wired FPGAs are easy to prototype using an FPGA and then can be made into fixed logic later on. The density can also be increased because the configuration memory can be eliminated [38]. To build on this concept, Altera's HardCopy II [41] converts the FPGA design into an ASIC made up of smaller cells, but customization is still performed using two metal layers. This further improves performance, density, and power efficiency. In essence, these two approaches allow the FPGA users to build fast and efficient, fixed logic without the cost of a full fabrication mask set or the complexities and risks of physical design. However, it requires approximately 8 weeks [40] and the FPGA is no longer reconfigurable. Though this is acceptable for custom computing, a hard-wired FPGA with fixed logic and routing cannot be used for reconfigurable custom computing.

2.1.4 System-on-a-Chip Designs with Embedded Programmable Logic Cores

The system-on-a-chip (SoC) design methodology involves the re-use and integration of off-the-shelf intellectual property (IP) cores into the design of a single chip in efforts to reduce design complexity. These IP cores are designed and tested by third-party vendors to provide a specific functionality. The SoC designer would assemble a number of these IP cores together to construct a larger chip. Given the popularity of FPGAs and their reconfigurability, SoC designers are starting to embed FPGA-like programmable logic cores [42, 43, 44, 45, 46, 47, 48, 49] in chips along with fixed logic IP cores. Figure 2.4 gives an example of this.

FPGA cores offer the flexibility to modify the chip's function when the specifications change, the possibility of producing a single chip that can be dynamically customized for a family of related applications, the opportunity to design a new chip by customizing only the embedded FPGA in platform-based design, and use of the programmable logic to perform on-chip tests [50]. As part of a reconfigurable computer, such a programmable fabric could possibly be configured during run-time as a custom functional unit [51, 52] or co-processor [53, 54].

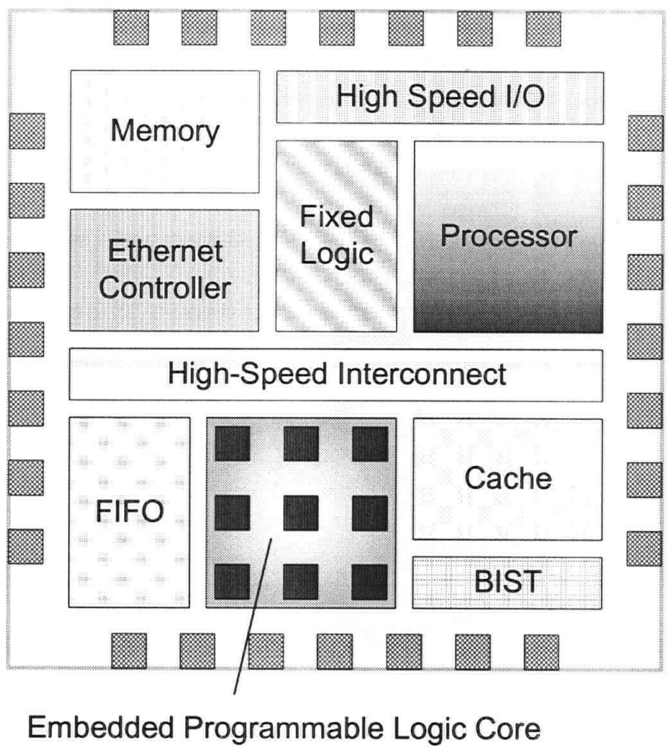


Figure 2.4: System-on-a-Chip Design with an Embedded Programmable Logic Core

There are two broad categories of programmable logic cores: hard cores and soft cores. Hard cores [42, 43, 44, 45, 46, 47] are circuits of programmable logic described by a layout that is ready to be integrated with the rest of the logic on the chip. Like most FPGAs, the hard cores have customized layouts to maximize their density. Soft cores [48, 49] are circuits of programmable logic that are described using a high-level hardware description language (HDL), that can be synthesized along with the other logic on the chip using a standard computer-aided design (CAD) flow. This eliminates the need to add the programmable IP core in the physical layout stage. It also means the size of the soft core can be adjusted to what is needed. However, the soft cores are slower and consume significantly more area than the hard cores.

2.1.5 Fine-Grained Reconfigurable Architectures Coupled with a Microprocessor

Fine-grained reconfigurable architectures have logic blocks that operate on bits that are connected by programmable logic. Components such as logic gates, LUTs, and flip-flops, which are used to implement Boolean logic functions, usually characterize these architectures [1]. While an FPGA fits this description, we describe in this subsection fine-grained reconfigurable architectures that have an FPGA-like fabric coupled with a microprocessor.

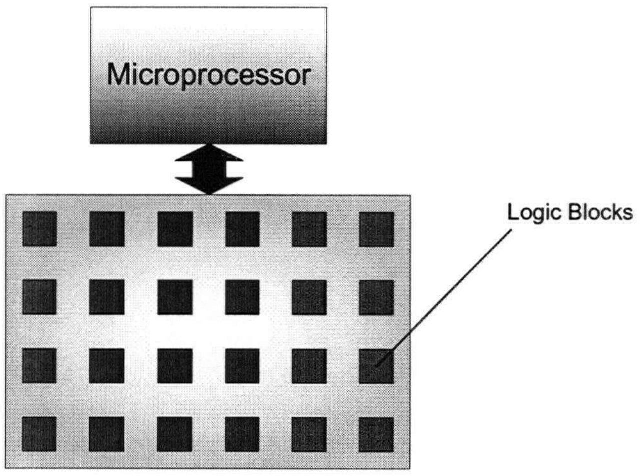


Figure 2.5: Fine-Grained Reconfigurable Architecture

As mentioned in the previous subsection, an embedded FPGA can be combined with a microprocessor to serve as a functional unit or a co-processor as shown in Figure 2.5. This has been researched substantially, even while FPGAs were still stand-alone devices. Programmable logic used as functional units [51, 52, 55, 56, 57] and co-processors [53, 54, 58, 59, 60, 61] in fine-grained reconfigurable architectures have delivered large increases in performance compared to microprocessors. For example, the reconfigurable functional unit in PRISC [51] increased performance by 22%, while Splash [58] demonstrated a 2700 times speedup over a Sun Workstation on a DNA sequence matching application.

Even though fine-grained reconfigurable logic is suitable for implementing functional units and co-processors, there are applications where custom processing capabilities are not needed. Rather, parallel processing on bytes or words using standard instructions are sufficient. An FPGA can do this, but its logic and routing flexibility exceed what is necessary. FPGAs are more efficient for implementing random logic than for applications with datapaths [2].

2.1.6 Coarse-Grained Reconfigurable Architectures

Medium and coarse-grained reconfigurable architectures represent a class of programmable logic architectures that operate on data that are 4 to 32 bits wide. There are typically functional units resembling arithmetic logic units (ALUs) that operate on this data as well as interconnect that routes data in 4- to 32-bit wide buses. Like fine-grained reconfigurable architectures, medium and coarse-grained reconfiguralbe architectures have blocks with programmable functionality which communicate over programmable interconnect.

Coarse-grained reconfigurable architectures are typically comprised of an array of configurable functional units (FUs) with programmable interconnect, as shown in Figure 2.6. They are similar to FPGAs except that larger functional units take the place of LUTs and the routing buses primarily carry words rather than bits. Compared to FPGAs, coarse-grained reconfigurable architectures require far fewer configuration bits, which can also be thought of as instruction memory.

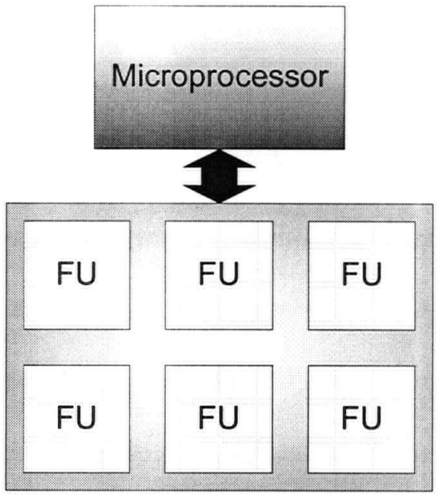


Figure 2.6: Coarse-Grained Reconfigurable Architecture

2.1.7 VLIW Processors

Very long instruction word (VLIW) processors, like coarse-grained reconfigurable architectures, have multiple functional units. VLIW processors also have local memory in the form of one or more register files. They differ in the fact that VLIW processors usually have functional units that only communicate via the shared global register file while the functional units in reconfigurable architectures communicate through a programmable interconnect. This requires VLIW processors to have many read and write ports on the global register file, slowing down access speed [62]. This also results in a scalability bottleneck; VLIW processors typically have four or at most eight functional units [62], far fewer than the 16 or 64 functional units found in some coarse-grained architectures [1, 3, 5, 8, 9, 10]. To scale the VLIW processor further in terms of the number of functional units, one to four functional units are typically clustered so that each cluster has a separate register file. However, the communication between clusters is complex [62].

2.1.8 Clustered VLIW Processor with a Reconfigurable Inter-Cluster Bus

The clustered VLIW with a reconfigurable inter-cluster bus, illustrated in Figure 2.7, is able to scale well to a greater number of functional units [62]. These clusters communicate via a bus with a ring topology, with connection boxes associated with each cluster's register file. Again, the communication between functional units is restricted to shared register file access. In fact, this clustered VLIW is very similar to CRISP [13], a coarse-grained reconfigurable architecture, which has four functional units and a register file in each slice. The functional units in CRISP can communicate directly with functional units in the same slice as well as through shared access to register files. Since the clustered VLIW architecture has programmable functional units and programmable interconnect, it too could be classified as a coarse-grained reconfigurable architecture.

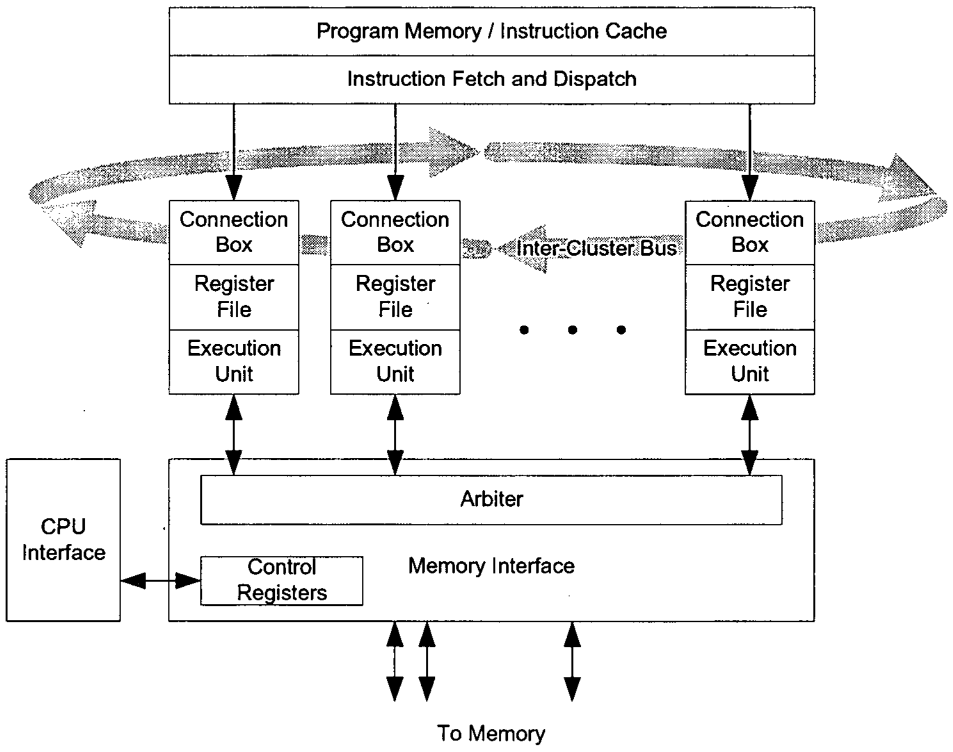


Figure 2.7: Clustered VLIW Processor With an Inter-Cluster Bus (From [62])

2.2 Coarse-Grained Reconfigurable Architectures

This section delves deeper into the area of coarse-grained reconfigurable architectures and presents the applications, distinguishing features, example architectures, and architectural studies.

2.2.1 Applications

Coarse-grained reconfigurable architectures are suitable for accelerating loops by exploiting instruction-level parallelism (ILP) and loop-level parallelism (LLP). One area of interest has been digital signal processing (DSP) applications, which have many large loops in the form of fast Fourier transforms (FFTs), finite impulse response (FIR) filters, and correlations [63, 64, 65]. Similarly, MorphoSys [1] was designed for image processing and another architecture [66] for parallel particle filtering. However, when there are many loop-carried data dependencies, the large number of functional units in these architectures cannot be fully utilized because there is not enough parallelism available in the loop [67]. This observation is similar to Amdahl's Law, which indicates that data dependencies in the form of sequential code restricts the maximum speedup available from parallel execution [68].

2.2.2 Distinguishing Features

Some features that distinguish the coarse-grained reconfigurable architectures from one another are the nature of the functional units, the interconnect, and the register files.

As summarized in Table 2.1, functional units (FUs) have granularities of 4 bits, 8 bits, 16 bits, or 32 bits, and can perform addition, multiplication, and other operations. In the case of MorphoSys [1], functional units are wired to execute the same instruction across a row or down a column like a single-instruction multiple-data (SIMD) processor. Another approach has been to fabricate a linear array of fixed-logic standard cells targeted for an application domain [70]. In the application domain of DSPs, specialized functional units have also been proposed [63, 64, 65, 66].

Table 2.1: Coarse-Grained Reconfigurable Architectures Organized By Granularity of Functional Units

Granularity of Functional Units	Coarse-Grained Reconfigurable Architectures
4 bits	PipeRench [2], CHESS [7], D-Fabrix [69]
8 bits	PADDI [6], DReAM [15]
16 bits	MorphoSys [1], REMARC [5], PADDI [6], PADDI-II [10], Montium [14], DReAM [15], RaPiD [70]
32 bits	ADRES [3], RAW [8], MaRS [9], Kress ALU Array III [12], CRISP [13], Chameleon [71]

The interconnect has 4- to 32-bit wide buses, which can be arranged in a mesh, where functional units communicate with their neighbours [1, 5, 7, 8, 11, 15], or like a crossbar [6, 10], in which the functional units communicate with many other functional units. There are direct FU to FU connections [1, 5, 7, 11, 12, 14, 15], buses that span several functional units [1, 2, 5, 7, 11, 14, 15, 70], and network routers [8, 9]. Some architectures have several layers of routing, possessing elements of more than one style [1, 5, 7, 11, 14, 15].

The structure and connectivity of register files varies between different architectures. REMARC [5] contains local register files that are only accessible by a single FU. Input register files are found in [6, 10, 11, 14]. The presence of register files with read and write access shared between functional units characterizes the architectures in [1, 5, 9, 13, 72]. This shared access enables communication between functional units in addition to temporary storage [9].

2.2.3 Examples of Coarse-Grained Reconfigurable Architectures

In this subsection, we describe three architectures: RaPiD-I [70], MorphoSys [1], and ADRES [3].

2.2.3.1 RaPiD-I

The RaPiD-I architecture [70] consists of a linear array of functional units arranged in a manner similar to standard cells. There are 16 sets of functional units, each set containing two integer 16-bit ALUs, one 16-bit x 16-bit multiplier, six 16-bit registers, and three small local memories. These functional units are connected by segmented 16-bit wide routing buses to form a linear datapath, as shown in Figure 2.8. The inputs and outputs come in the form of I/O streams at both ends of the datapath. These I/O streams are implemented as FIFOs that read and write data.

The functional units within each cell are independent of one another. Each functional unit input has an input multiplexer that selects from eight 16-bit buses; each functional unit output can be configured to be registered and can drive up to eight 16-bit buses via tri-state buffers. Local memories can be configured to have input registers as well.

There are ten 16-bit wide buses with varying segment lengths that span the length of the linear datapath. Just as there are switch blocks located at the intersections between horizontal and vertical routing channels in FPGAs, there are buffers and registers located between adjacent segments of the same horizontal 16-bit bus. These buffers and registers allow two segments to be concatenated to form a longer segment, as well as pipeline stages to be inserted.

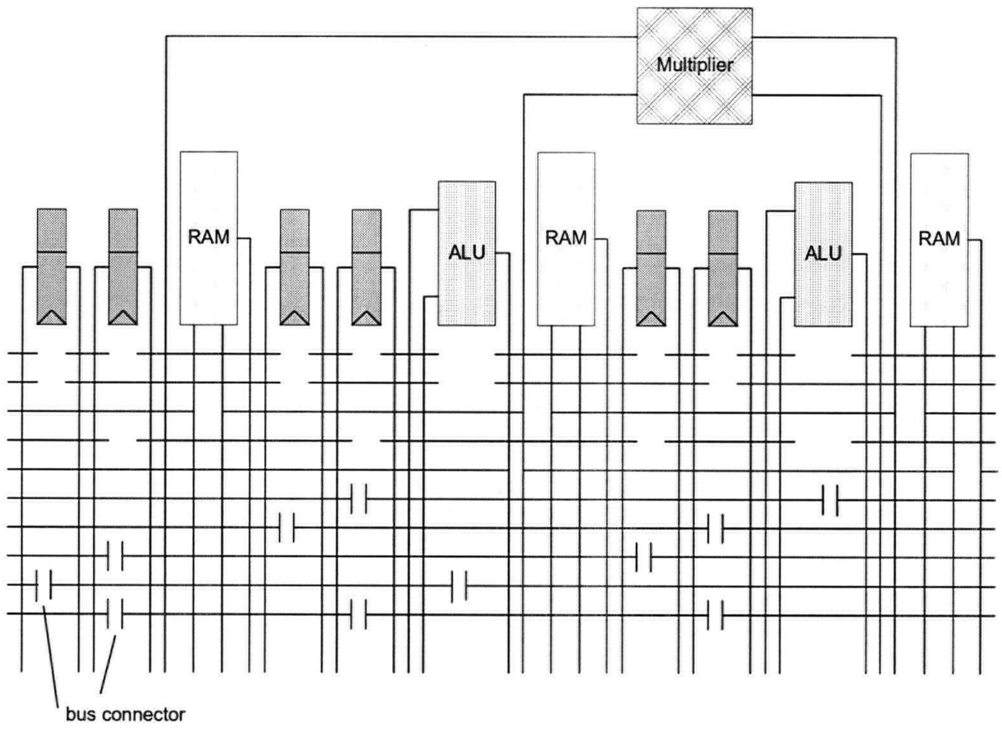


Figure 2.8: One of Sixteen Sets of Functional Units in the Rapid-I Architecture (From [70])

2.2.3.2 MorphoSys

MorphoSys [1] is an image processing reconfigurable architecture. It consists of a reduced instruction set computer (RISC) processor, an array of 64 reconfigurable cells (RCs), a frame buffer, and a direct memory access (DMA) controller, as shown in Figure 2.9. With the layout specified in 0.35 μm fabrication process technology, the 8x8 array of reconfigurable cells takes up 96 mm^2, roughly half of the chip. Simulations using HSPICE show that it has a clock speed of 100 MHz.

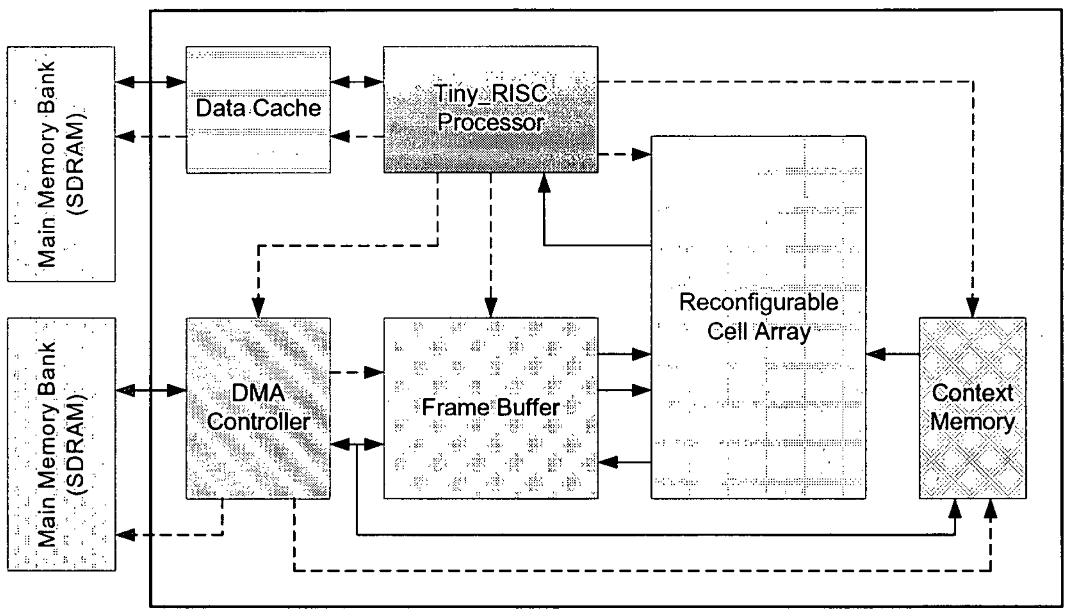


Figure 2.9: Block Diagram of MorphoSys Architecture (From [1])

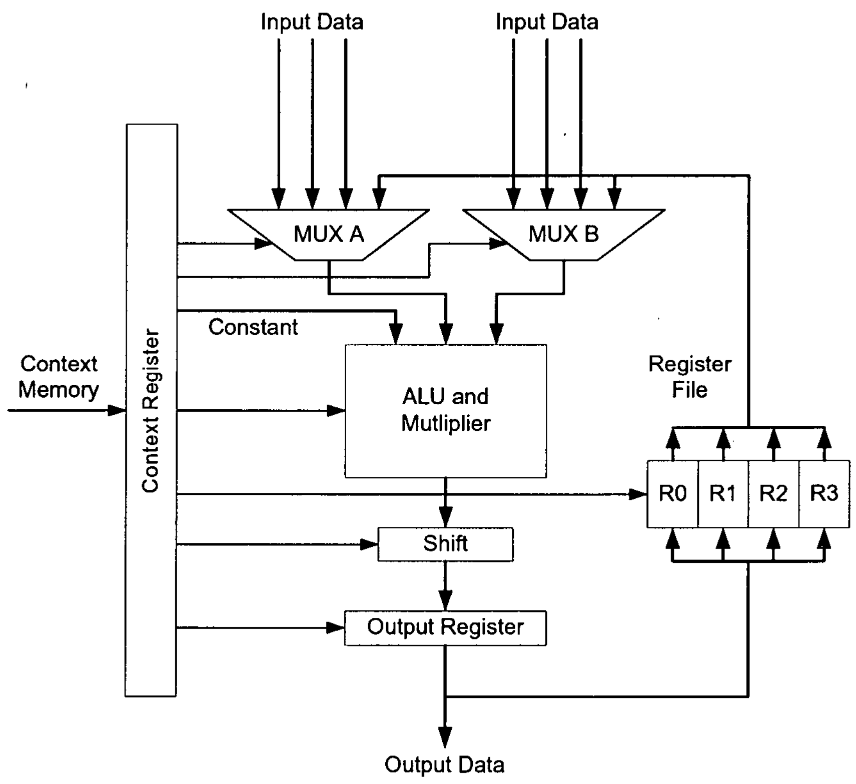


Figure 2.10: Reconfigurable Cell in the MorphoSys Architecture (From [1])

Each reconfigurable cell, illustrated in Figure 2.10, has a 28-bit fixed point ALU and a 16-bit x 12-bit multiplier. There are two multiplexers for the two 16-bit inputs to the ALU and a 12-bit constant input. As well, a register file with four 16-bit registers is available to store data from the ALU's output register for subsequent use as an ALU input.

As shown in Figure 2.11, the interconnect between the reconfigurable cells has three components: nearest neighbour, intra-quadrant, and inter-quadrant interconnect. Each reconfigurable cell is connected to its four nearest neighbours to form a two-dimensional mesh topology. Inside each quadrant of 4x4 reconfigurable cells in the 8x8 array, the reconfigurable cells are connected directly to the three other cells in the same row and the three other cells in the same column. Between quadrants, horizontal and vertical 16-bit buses called express lanes exchange data from the four reconfigurable cells in one quadrant to four reconfigurable cells in another quadrant that are in the same row or column.

Unlike most reconfigurable systems where each functional unit has its own set of configuration bits, MorphoSys shares the same set of configuration bits between eight reconfigurable cells in a row, or alternatively, eight reconfigurable cells in a column. This technique is called context broadcast. The context memory can store up to 32 row or column contexts. New contexts can be loaded into available context memory while the reconfigurable array executes instructions from another set of contexts. Moreover, entire rows or columns can be disabled, so that data transfers require only one row or column of context.

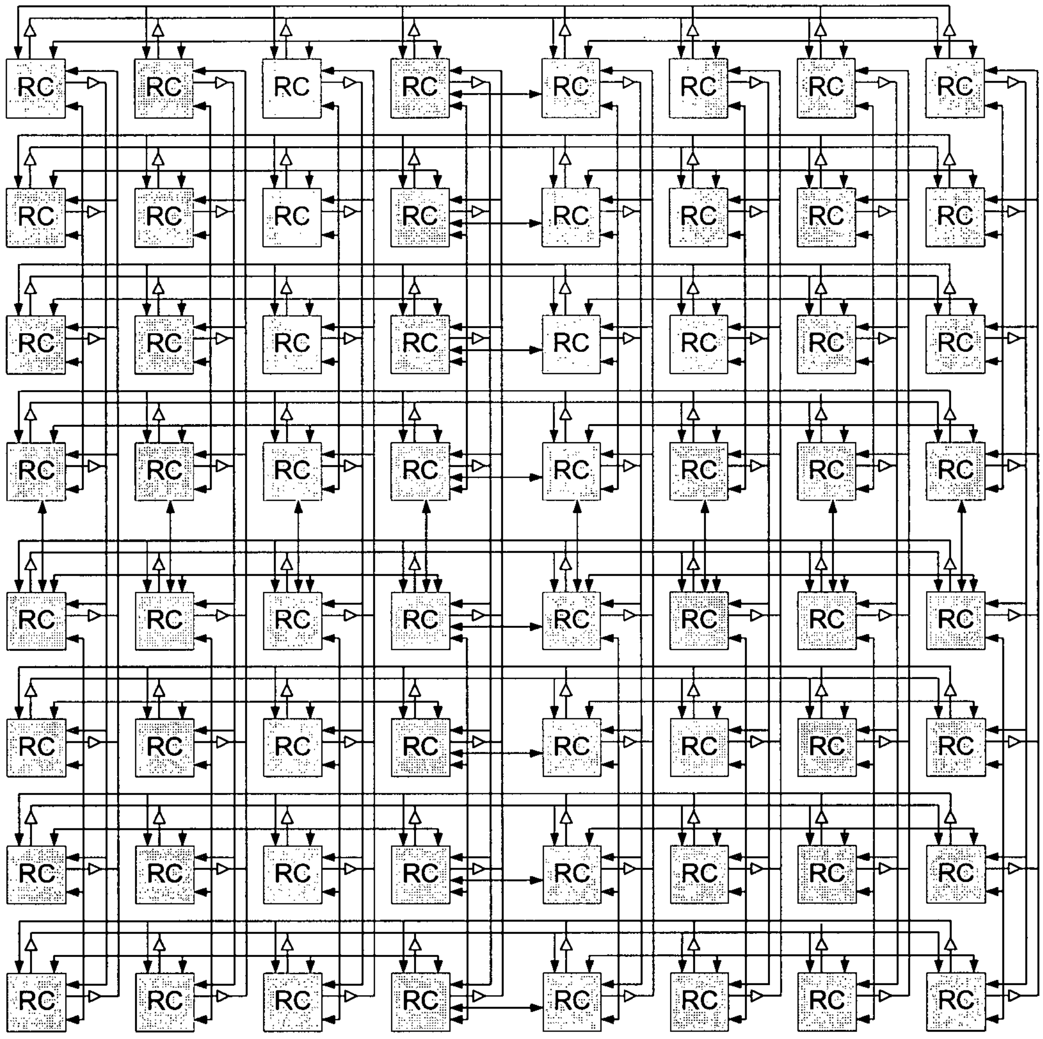


Figure 2.11: MorphoSys Interconnect Architecture (From [1])

2.2.3.3 ADRES

The ADRES architecture [3] consists of a reconfigurable array coupled with a general-purpose VLIW processor. Highly pipeline-able loops are identified by a compiler [73, 74] and executed on the reconfigurable array, while sequential code is executed by the general-purpose VLIW processor. The general-purpose VLIW processor and the reconfigurable unit communicate via two global register files.

ADRES differs from MorphoSys in at least five ways. First, ADRES contains a VLIW processor that is tightly coupled to the reconfigurable array by the shared global register files, unlike MorphoSys, which has a separate RISC processor. Second, MorphoSys has a frame buffer -- something not found in ADRES. Third, every functional unit in ADRES can execute a unique instruction at any given time, whereas reconfigurable cells in MorphoSys execute the same instruction across a row or column. Fourth, ADRES has multipliers in some of its functional units, while every reconfigurable cell in MorphoSys includes a multiplier. Fifth, ADRES operates using 1-bit predicates and 32-bit data, while MorphoSys operates using 16-bit data. This suggests that ADRES can execute instructions conditionally using predication while MorphoSys cannot.

A very similar architecture based on ADRES, shown in Figure 1.1, will be the framework of the experiments in this thesis. Its details will be given in Chapter 3.

2.2.4 Architectural Studies

In order to develop more efficient coarse-grained reconfigurable architectures, researchers have experimentally measured the impact of the functional unit architecture, the interconnect architecture, the memory interface architecture, and the register file architecture on the performance and density of a coarse-grained reconfigurable architecture. In this subsection, we consider each of these separately.

First, the functional units in the coarse-grained reconfigurable architectures have been studied, particularly in terms of performance. Increasing the number of functional units can significantly reduce the number of instruction cycles required to execute a loop, but it was found that the loop-carried data dependencies limit this speedup [2, 67]. Bansal et al. [67] found that dividing a functional unit into two smaller fully connected functional units, one of which performs multiplication while the other the remaining operations, results in a performance gain of 0 to 13%. The performance gain is not observed, however, when the multiplication and arithmetic units cannot be used simultaneously in certain benchmark programs. Ienne et al. [75] state that predication support, which allows the execution of conditional *if* statements, is fundamental to parallelism, and is therefore essential for obtaining the best results. Compton et al. [76] try to allocate just the appropriate amount of each type of functional unit for specific application domains. Goldstein et al. [2] have compared the computational density of functional units with granularities of 2, 4, 8, 16, and 32 bits, measured in bit-operations per unit time per unit area, and found that the computational density remained relatively constant for different granularities. While area can be optimized for functional units with larger word widths, the delay of the adder's carry chain also increases. Nevertheless, they predict that the computational density would increase linearly with increasing bit widths if logical operations were implemented rather than arithmetic operations.

Second, the functional unit interconnect has been explored. Compton et al. [77] show how track placement for a segmented routing architecture can be assigned within 1.2% of the optimal spreading of wires, where the number of routing wires accessible to each node is balanced. By fully connecting functional units in clusters of four, Bansal et al. [67] increased the performance by 0 to 41% above the performance when the functional units are neither grouped into clusters nor fully connected. In a different study, Bansal et al. [78] found that connecting a functional unit to two other functional units in the same row and two in the same column is sufficient to achieve good performance. Compared to connecting to only two functional units, one in the same row and one in the same column, connecting to four gives significantly higher performance. Connecting to six functional units gives only slightly higher performance. These experiments assumed the presence of row and column buses. In the absence of row and column buses, Wilton et al. [79] showed that connecting a functional unit to six functional units gives performance comparable to the performance achieved by fully connecting the functional units, but with much smaller area.

The architecture of the memory interface has also been studied. Ienne et al. [75] showed that most basic blocks of dataflow, or code segments without control instructions, require very few memory accesses. Therefore, having one memory read and write port is only 15-20% slower than having an unlimited number of memory ports for the reconfigurable logic. Lee et al. [80] also showed that the inclusion of an address generator can improve performance, though its effect varies between different applications.

As for register files, Tabrizi et al. [9] have explicitly referred to them as a second layer of communication between functional units. Colavin et al. [62] allocated two read ports and two write ports on a shared register file for inter-cluster communication and two read ports and one write port for the local cluster of functional units, based on empirical data. A more general study by Ienne et al. [75] showed that it is important for performance to have at least two or three write ports on the global register file for the reconfigurable logic to access, though this may be costly in terms of area.

2.3 Focus of this Thesis

The goal of this research is to study the role of register files in coarse-grained reconfigurable architectures, which were introduced in Subsection 2.1.6 and explored further in Section 2.2. Chapter 3 describes the reconfigurable array of a coarse-grained reconfigurable architecture that is based on the ADRES architecture discussed in Subsection 2.2.3. Chapter 4 describes an experimental flow for conducting an architecture study on the register files similar to studies in Subsection 2.2.4. The results of the experiments are shown and discussed in Chapter 5. The knowledge gained about register files is then applied to create two new and improved architectures in Chapter 6.

Chapter 3 FRAMEWORK

In this chapter, we describe the framework upon which our study is built.

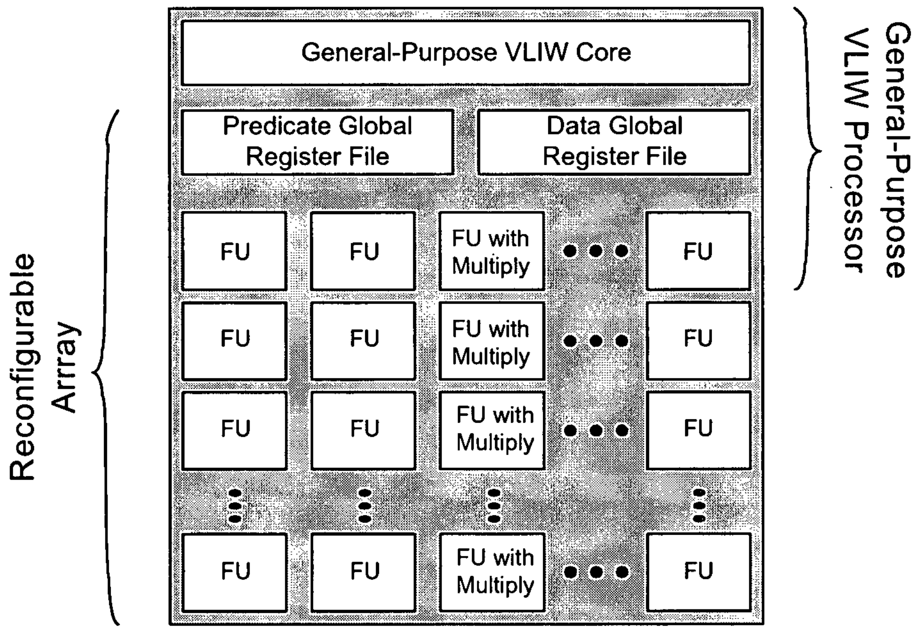
To make our results concrete, we have performed our studies in the context of a specific reconfigurable architecture, ADRES [3]. As described in Chapter 2, ADRES contains an array of coarse-grained logic blocks integrated with a very long instruction word (VLIW) processor. Application code is mapped to the architecture by first using the IMPACT-I compiler [73], developed at the Center for Reliable and High-Performance Computing in the University of Illinois at Urbana-Champaign, to perform architecture-independent optimization and parallelism extraction, followed by DRESC [74] to perform architecture-dependent mapping. Section 3.1 will describe the reconfigurable array of the ADRES-based architecture, Section 3.2 will describe the IMPACT-I and DRESC compilers, and Section 3.3 will state the architectural design space that will be explored in this thesis.

3.1 Architecture

In this section, the computation units, the register files, the interconnect, and the context memory of the reconfigurable array will be described in detail.

3.1.1 Computation Units

The reconfigurable fabric consists of a heterogeneous array of configurable functional units (FUs). In this thesis, we will consider two array sizes: 4x4 and 8x8. Each FU can perform a subset of forty-five 32-bit operations in each clock cycle. All FUs can perform a variety of arithmetic (signed and unsigned), logic, shifting, and comparison instructions. One out of every four columns has FUs that can also perform 16-bit x 16-bit multiplication, as shown in Figure 3.1.



General-Purpose VLIW Core
Predicate Global Register File
Data Global Register File
FU
FU with Multiply
General-Purpose VLIW Processor
Reconfigurable Arrray

Figure 3.1: Columns of Functional Units With Multipliers

The FUs in the top row are different than the others. These FUs 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 FUs when the device is executing parallel code on the reconfigurable array. These FUs have direct connections to the two global register files through dedicated read and write ports.

Figure 3.2 shows a single configurable functional unit (FU) in the baseline architecture. The important features of the FU are depicted in the figure, but the number of signals and read and write ports will vary between the different architectures that are discussed in this thesis. Each FU contains both a configurable ALU (the function of which can be specified using context memories, analogous to the configuration memories in an FPGA or the instruction memory in a microprocessor) and two local register files; these register files will be described in more detail below.

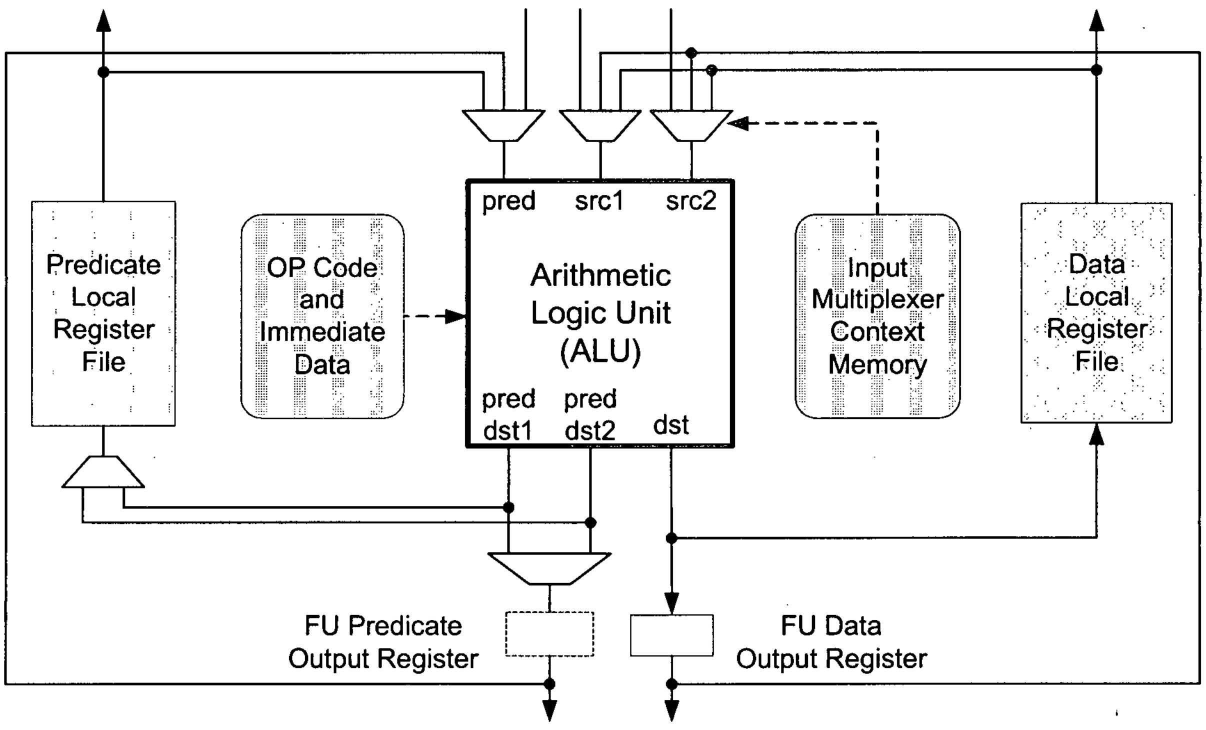

pred
src1
src2
Predicate Local Register File
OP Code and Immediate Data
Arithmetic Logic Unit (ALU)
Input Multiplexer Context Memory
Data Local Register File
pred dst1
pred dst2
dst
FU Predicate Output Register
FU Data Output Register

Figure 3.2: Configurable Functional Unit in the ADRES-based Architecture

3.1.2 Register Files

As described above, this architecture has two types of register files: two local register files within each FU, and two global register files, as shown in Figure 3.3.

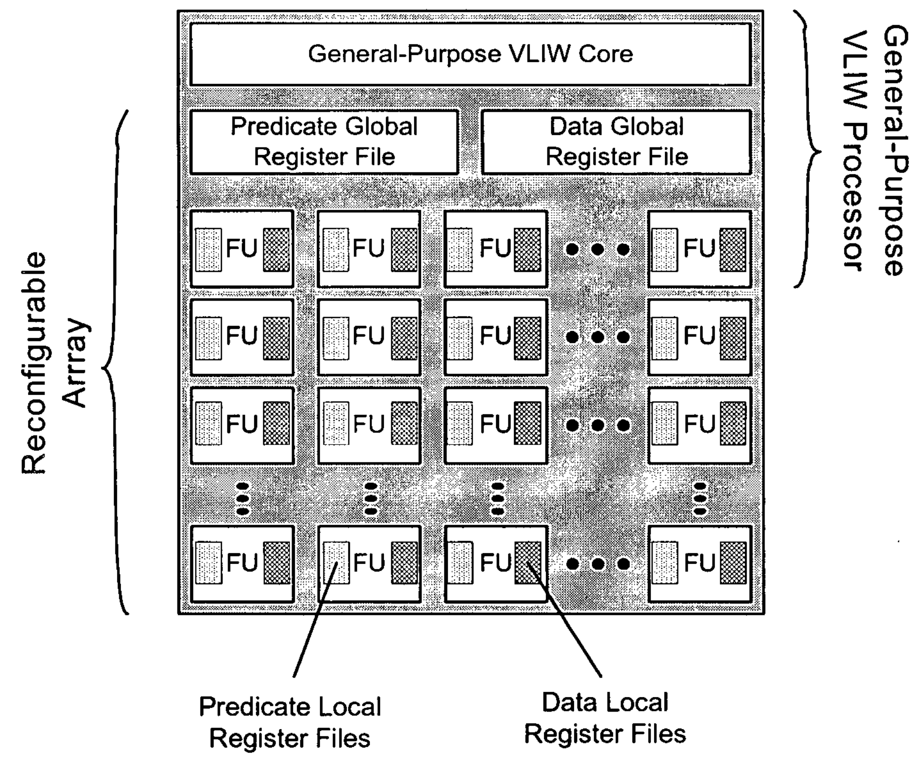

General-Purpose VLIW Core
Predicate Global Register File
Data Global Register File
FU
General-Purpose VLIW Processor
Reconfigurable Arrray
Predicate Local Register Files
Data Local Register Files

Figure 3.3: Global and Local Register Files in the ADRES-based Architecture

First consider the two global register files. These register files are used for two purposes: (1) to transfer data between the general-purpose VLIW processor and the reconfigurable array, and (2) as scratch-pad memory for the functional units in the top row of the reconfigurable array when they operate as the general-purpose VLIW processor. There is one global register file (32 bits wide) to store data values, and one global register file (1 bit wide) to store predicate values.

As described above, in the baseline architecture, each data global register file is connected to each FU in the top row through two dedicated read ports, one for each data ALU input, and one dedicated write port. Thus, in a 4x4 array, each data global register file has eight read ports and four write ports; in an 8x8 array, each data global register file has sixteen read ports and eight write ports.

Each predicate global register file is connected to the FUs in the top row through one dedicated read port and one dedicated write port. Thus, in a 4x4 array, each predicate global register file has four read ports and four write ports; in an 8x8 array, each predicate global register file has eight read ports and eight write ports.

There are also two local register files within each functional unit. These register files can be used as scratch-pad memory by the functional units in the reconfigurable array, and can be used to transfer data between functional units. Each functional unit contains one 32-bit wide register file to store data values, and one 1-bit wide register file to store predicate values.

The main difference between global and local register files is that global register files have multiple read and write ports to support the operation of the general-purpose VLIW processor while the local register files typically have one write port and possibly several read ports. Another difference is that the write enable signals on the global register files can be controlled by predicate signals generated by functional units, while the write enable signals of local register files are determined solely by values stored in configuration bits.

Note that the functional unit output registers can simply be viewed as local register files with one read port and one write port. There have been attempts to amalgamate the output register with the local register file, but the preliminary performance results have generally been disappointing. These results suggest that the DRESC compiler does not handle them this way. Since output registers and register files are specified differently in the architecture description, DRESC may assume different functions for them.

3.1.3 Interconnect

The functional units in the reconfigurable array communicate with the general-purpose VLIW processor via the two global register files. There are input multiplexers to each functional unit that receive data and predicate signals from other functional units. As well, the local register files may or may not be shared between different functional units, depending on the experiment.

The FUs are interconnected using the "closest" architecture as described in [79] and shown in Figure 3.4, Figure 3.5, and Figure 3.6. The interconnect buses that carry these signals are in the form of direct connections, and there are no shared buses or segmented routing. In this interconnect architecture, each of the two data ALU inputs is driven by registered versions of four neighbouring data FU outputs including the data output from the local FU. The single predicate ALU input is driven by registered versions of seven neighbouring predicate FU outputs including the predicate output from the local FU.

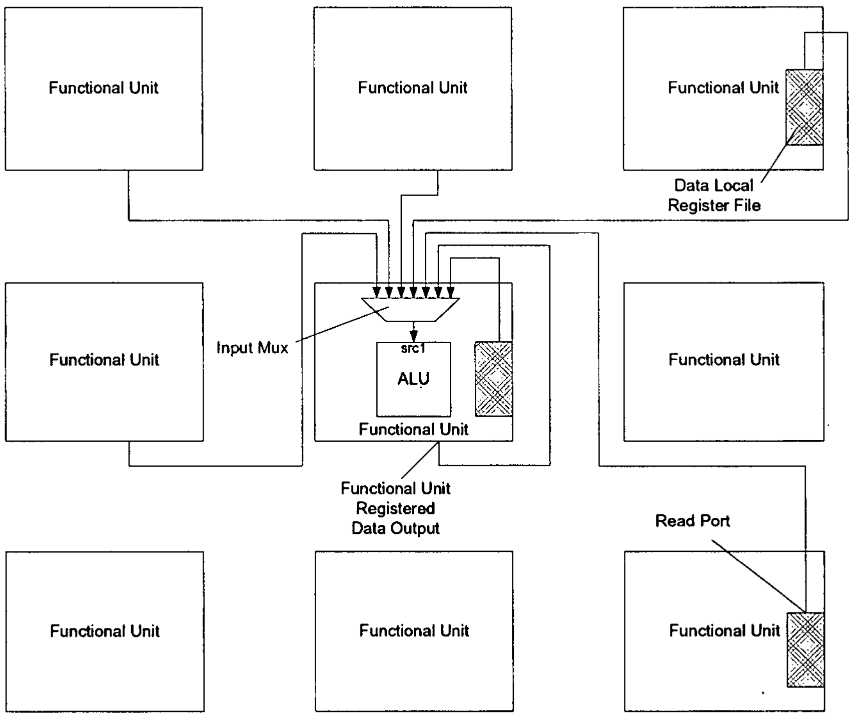


Figure 3.4: Interconnect Pattern for *src1* data ALU input in the Baseline Architecture

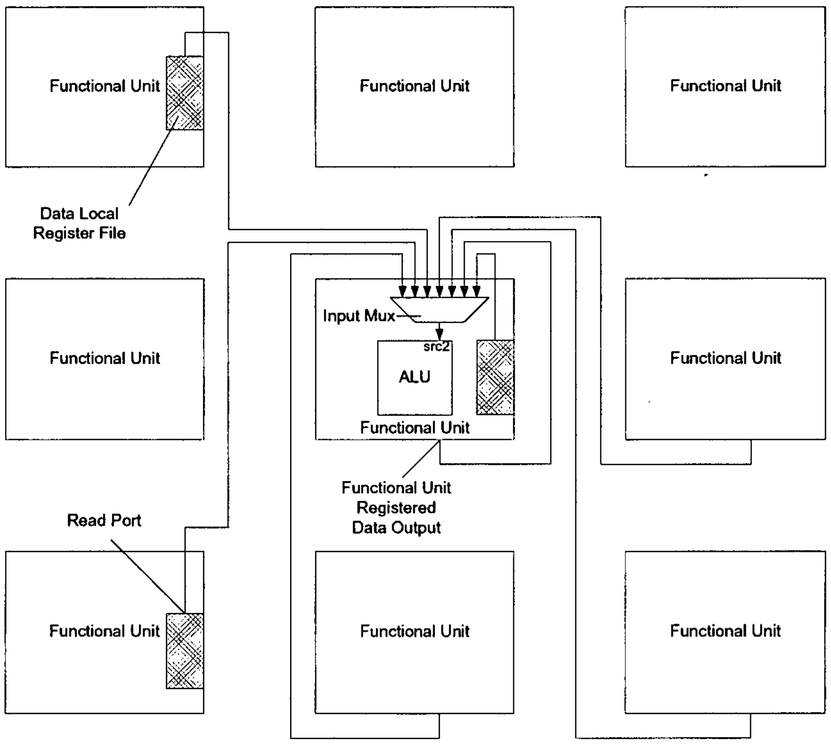


Figure 3.5: Interconnect Pattern for *src2* data ALU input in the Baseline Architecture

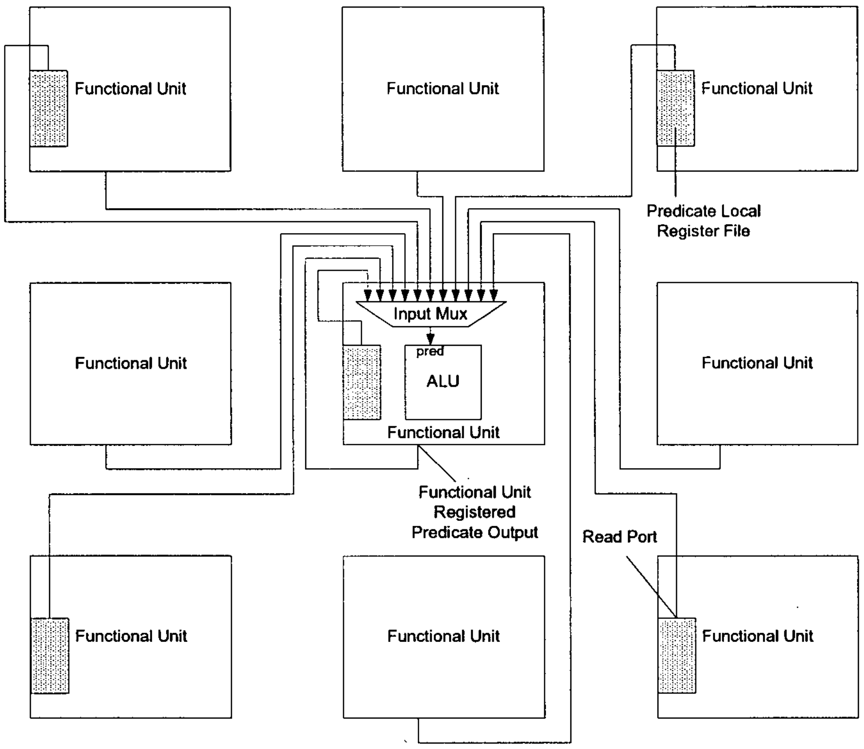


Figure 3.6: Interconnect Pattern for *pred* predicate ALU input in the Baseline Architecture

In the baseline architecture, each local register file has one read port and one write port. As shown in Figure 3.2, the write port of each local register file is driven by the unregistered output of the ALU within the local FU. The read port of each local register file, however, is connected to six sinks: the two inputs of the ALU in the local FU, and one input in each of four neighbouring FUs (the four FUs diagonally adjacent to the local FU). These connections are shown in Figure 3.7.

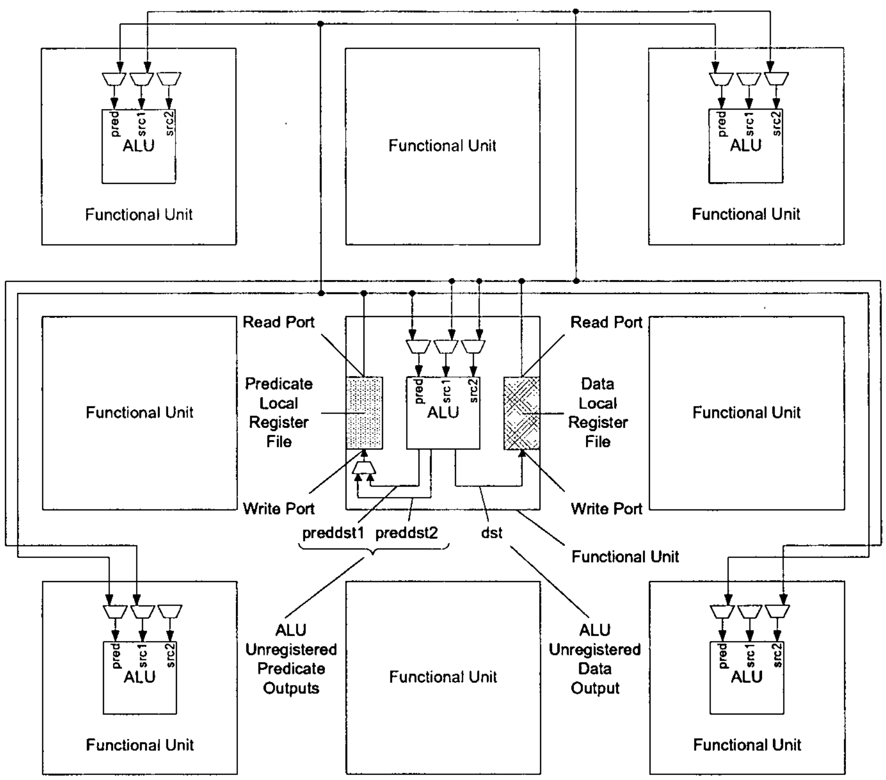


Figure 3.7: Interconnect Pattern for Local Register File Read Ports in the Baseline Architecture

Now we describe the interconnect of control signals. The loop stop signal, which is used to inform the VLIW processor that the reconfigurable array has finished a computation, is multiplexed from the three functional unit predicate output registers in the upper left corner of the functional unit array, as shown in Figure 3.8.

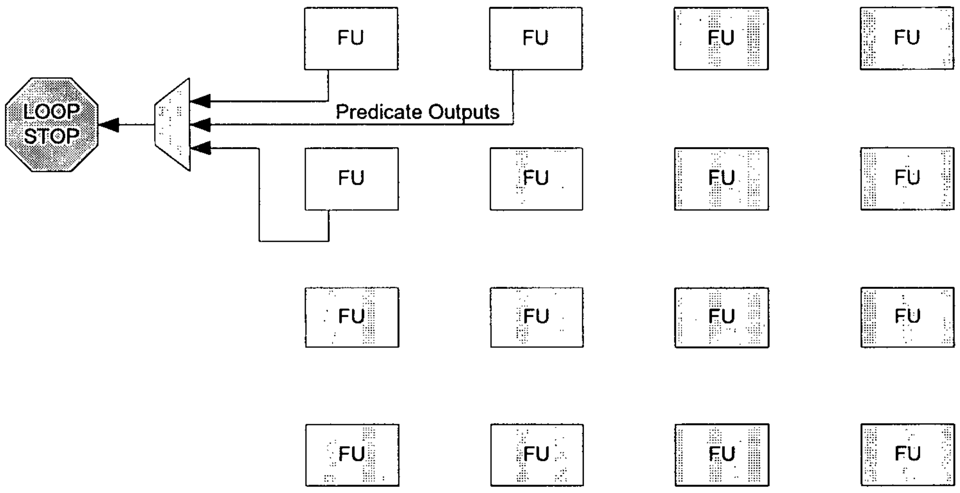
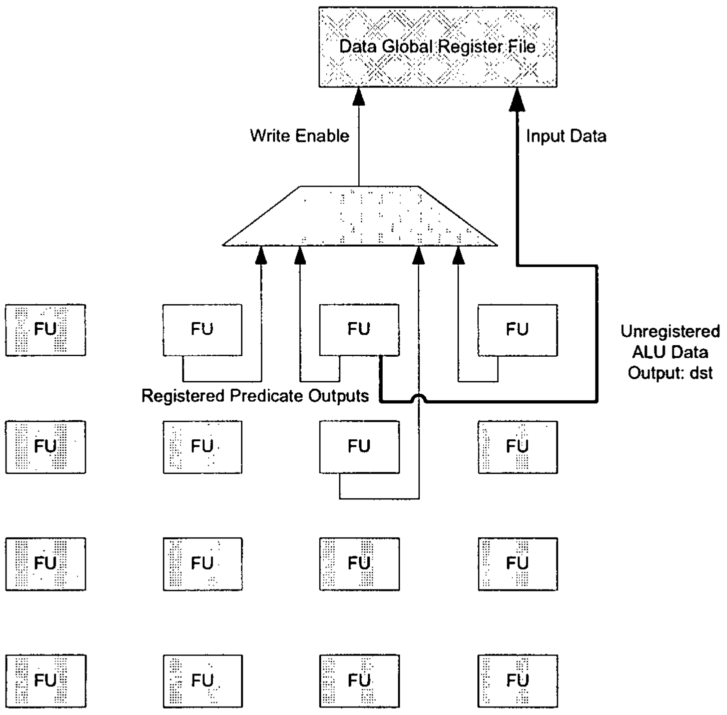


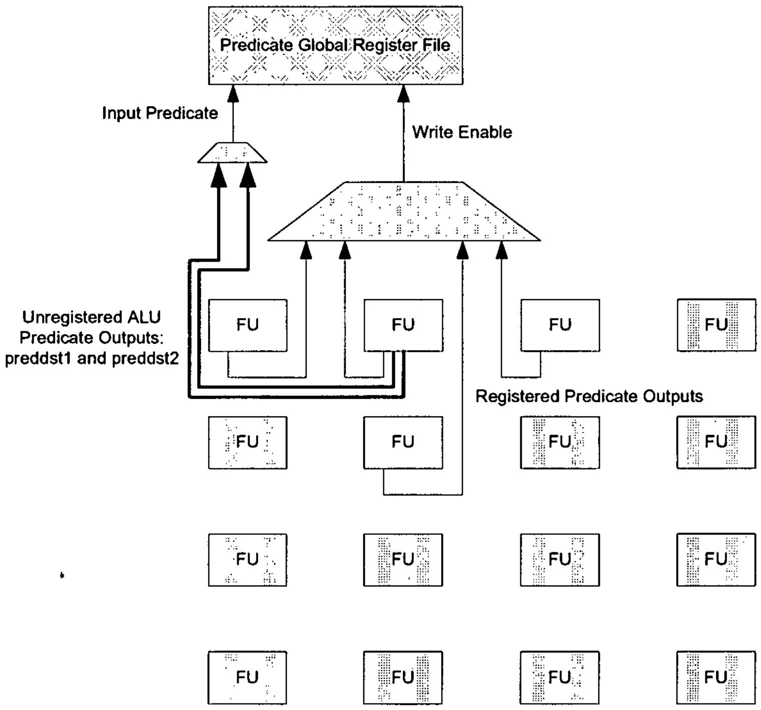
Figure 3.8: Generation of Loop Stop Signal

As shown in Figure 3.9, the write enable signal for a write port on the data global register files is supplied by the predicate outputs of the same functional unit that supplies the input data; additional inputs to the write enable multiplexer come from the predicate outputs of the functional units immediately to the left, to the right, and below. Similarly, as shown in Figure 3.10, the write enable signal for a write port on the predicate global register file is supplied by the predicate output register of the same functional unit that supplies the input data; additional inputs to the write enable multiplexer come from the predicate outputs of the functional units immediately to the left, to the right, and below. In Chapter 5, we will increase the number of functional units which supply potential data and predicate inputs to each write port; in these architectures, the additional functional units supply additional registered predicate outputs for the write enable multiplexer but the nearest neighbours of these additional functional units do not supply additional registered predicate outputs for the write enable multiplexer.



Data Global Register File
Write Enable
Input Data
FU
Unregistered ALU Data Output: dst
Registered Predicate Outputs

Figure 3.9: Generation of Write Enable Signal for the Data Global Register File



Predicate Global Register File
Input Predicate
Write Enable
Unregistered ALU Predicate Outputs: preddst1 and preddst2
FU
Registered Predicate Outputs

Figure 3.10: Generation of Write Enable Signal for the Predicate Global Register File

3.1.4 Context Memory

One set of configuration bits is required for each context. The configuration bits for a context represents the configuration during each clock cycle of a loop iteration and are selected using multiplexers. For the 4x4 array, 1 to 18 contexts are typically required for the set of benchmarks with approximately 2000 configuration bits in each context; for the 8x8 array, 1 to 8 contexts are typically required for the set of benchmarks with approximately 8000 configuration bits in each context. The number of required configuration bits remains constant throughout the lifetime of the loop, while the selected context is changed at every clock cycle.

3.2 Compilers

This section introduces two compilers, IMPACT-I and DRESC, that are used to schedule and map C programs onto the ADRES-based architectures. Figure 3.11 shows where they fit in the compilation flow.

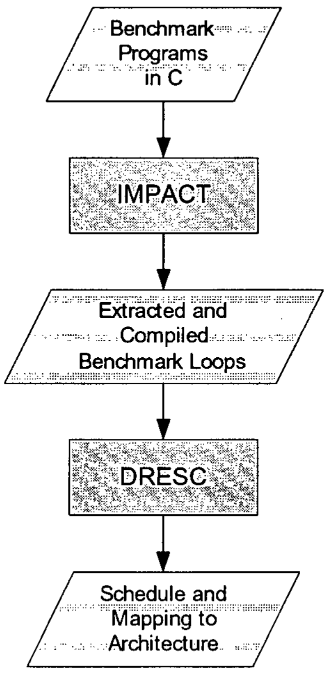

Benchmark Programs in C
IMPACT
Extracted and Compiled Benchmark Loops
DRESC
Schedule and Mapping to Architecture

Figure 3.11: Compilation Flow

3.2.1 IMPACT-I

When a C program is mapped to the ADRES-based architecture, it is first compiled using IMPACT-I. The IMPACT-I C compiler [73] was designed to perform optimizations for a parameterized processor that can execute multiple instructions at a time. This processor must be specified by its instruction set, architecture, and code scheduling model. IMPACT-I separates code into basic blocks that can be executed sequentially and applies code optimization techniques such as function inline expansion, loop unrolling, loop peeling, branch expansion, induction variable expansion, register renaming, global variable register allocation, operation combining, operation folding, memory disambiguation, inline target insertion, and speculative execution of instructions following a branch. Instructions can also be moved around in the code, and even pushed before or after branch instructions. IMPACT-I transforms the C code into an intermediate representation called "Lcode", which resembles a collection of assembly-level instructions to be mapped to the architecture.

3.2.2 DRESC

DRESC first partitions the Lcode into the loops that will be mapped to the reconfigurable array and the rest of the Lcode that will be mapped to the general-purpose VLIW processor [3]. The Lcode for the reconfigurable array is scheduled to exploit loop-level parallelism (LLP) while the Lcode for the general-purpose VLIW processor is optimized for instruction-level parallelism (ILP). DRESC identifies and allocates space in the global register files for the live-in and live-out variables. These are the input and output variables of the loop that constitute the interface between the Lcode for the reconfigurable array and the Lcode for the general-purpose VLIW processor.

Taking the particular basic blocks from IMPACT-I that constitute loops, the DRESC compiler [74] uses a modulo-scheduling algorithm to schedule each loop on the target reconfigurable array, assigning each operation in the loop to a specific FU in a specific time slice and assigning each variable to one of the register files, where temporary storage is necessary. The DRESC tool takes into account the characteristics of the target architecture, including the capabilities of each FU, the size of each register file, and the interconnect between the FUs and the register files. This ensures that the resulting schedule can be implemented on the reconfigurable device. In essence, the DRESC tool performs the tasks of scheduling, placement, and routing simultaneously. DRESC uses techniques from the simulated annealing algorithm [81] and the Pathfinder routing algorithm [82] to perform these tasks.

During mapping, DRESC attempts to minimize the iteration interval (II) of the scheduled loop. The iteration interval is the number of clock cycles between the initiation of successive iterations of the loop, as shown in Figure 3.12. As explained in [74], the II is equal to the number of contexts required in the configuration memory of the reconfigurable fabric. Therefore, the achievable II directly impacts the area of the architecture. Intuitively, if an architecture is more flexible (either because there are more registers or they are better located or connected to the computation units), the DRESC compiler will be able to find a schedule with a smaller value of II. On average, this means that if the fabric was constructed, a smaller number of contexts would be used, meaning less context memory is required, and thus smaller area is required. As explained in [74], the value of II also has a strong influence on the achievable *instructions per cycle* (IPC), which dictates the overall performance of the device.

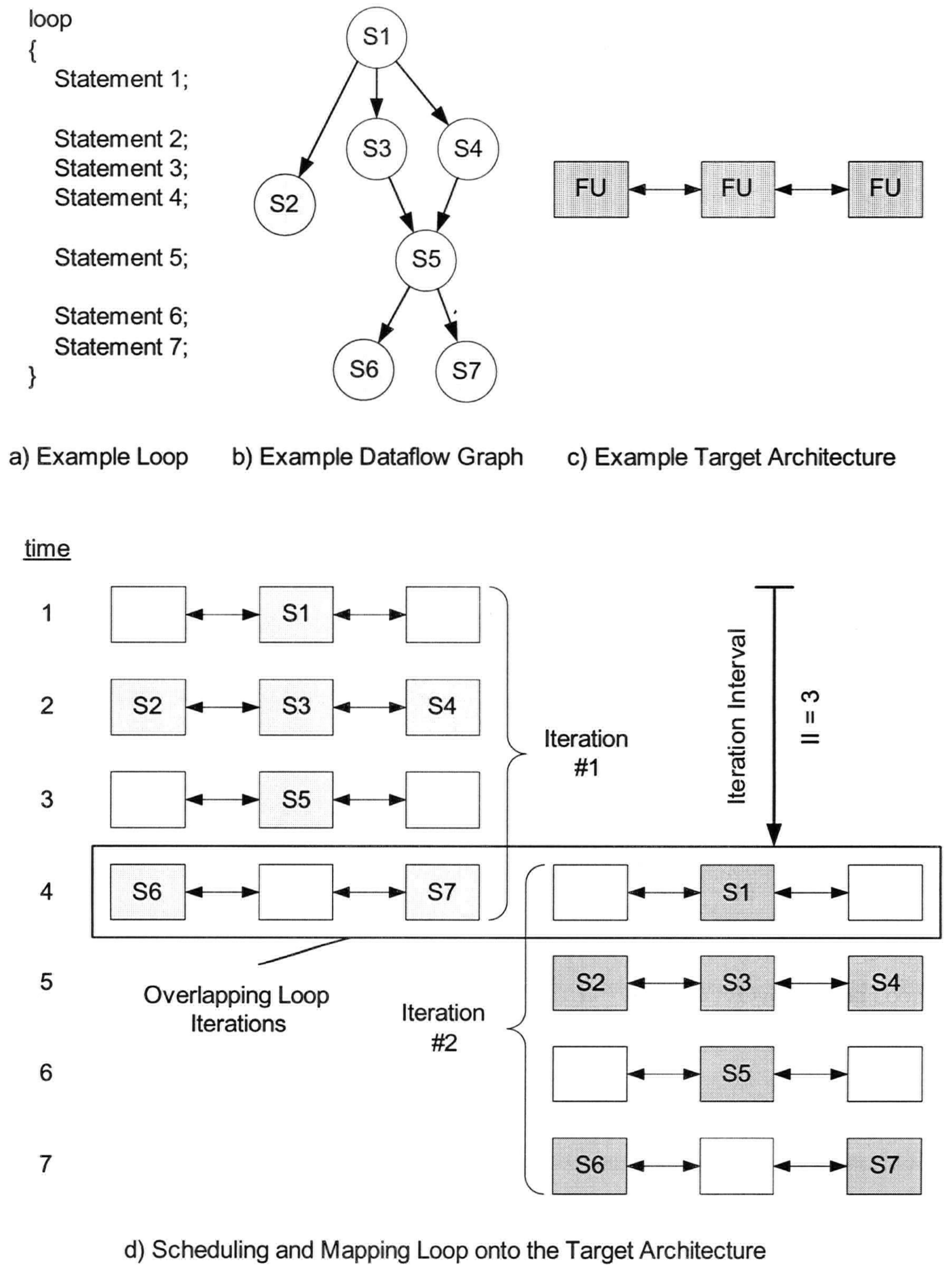


Figure 3.12: Illustration of Iteration Interval when Mapping a Loop to an Architecture

In this way, DRESC takes the parallelized code from IMPACT-I and maps it to the ADRES-based architecture. DRESC produces a schedule for the execution of each accelerated loop, as well as the assignment of each instruction to specific functional units. All of the information required for configuring the ADRES-based architecture, including register file accesses and multiplexer select lines, is generated by DRESC.

3.3 Design Space Explored in This Thesis

This section describes the design space explored in this thesis. In particular, it identifies the parameters that are varied in the experiments.

3.3.1 Parameters

Table 3.1 shows a list of the architectural parameters that describe the register files in the baseline architecture for both a 4x4 array and an 8x8 array. In Chapter 5, one parameter is varied at a time, and the impact on area and performance is measured. The range investigated for each parameter is also shown in Table 3.1.

Besides the "closest" interconnect architecture, the "full" and "empty" interconnect architectures are also used in some experiments. The "full" architecture describes an array of functional units in which each functional unit is connected to every other functional unit. The "empty" architecture describes the opposite; each functional unit has no direct connection to any other functional unit, but only connections from each functional unit to the global register files, which is typical in general-purpose VLIW processors.

Table 3.1: Architectural Parameters Used in This Thesis

Parameter		Value in Baseline Architecture	Range Considered in Chapter 5
Global Register Files	Number of FUs connected to each register file	4 (4x4 array) 8 (8x8 array)	4 to 16 (4x4 array) 8 to 64 (8x8 array)
	Number of read and write ports per register file	4* (4x4 array) 8* (8x8 array)	4* to 16* (4x4 array) 8* to 64* (8x8 array)
	Number of FUs with ports to the register file and to main memory	4 (4x4 array) 8 (8x8 array)	4 to 16 (4x4 array) 8 to 64 (8x8 array)
	Number of registers per register file	16 (4x4 array) 64 (8x8 array)	4 to 48 (4x4 array) 8 to 48 (8x8 array)
Local Register Files	Number of FUs connected to the read and write ports on each register file	5 per read port 1 per write port	1 or 5 per read port 1 or 5 per write port
	Number of read ports per register file	1	1 to 6
	Number of registers per register file	4	0 to 16

* There are twice as many read ports on the Data Global Register File.

3.4 Summary

In this chapter, the ADRES-based architecture has been described in detail. Then the capabilities of the IMPACT-I and DRESC compilers were presented in relation to how they are used to map C programs on the ADRES-based architectures. The chapter concluded with a discussion of the parameter values which will be varied when we explore the design space for the ADRES-based architectures.

Chapter 4
EXPERIMENTAL METHODOLOGY

This chapter describes the flow used to perform experiments on the ADRES-based architectures. The metrics chosen for evaluating the experimental results are also presented.

4.1 Experimental Flow

The design flow for mapping C programs to the ADRES architecture, using IMPACT-I and DRESC, have already been described in Chapter 3. This section will explain how the experimental architectures are incorporated into the overall flow. Then SCRAP, a tool useful for manipulating architectures, will be introduced along with the VHDL model that is included in SCRAP. Lastly, a description of the benchmark kernels will be given.

4.1.1 Overall Flow

In the overall flow shown in Figure 4.1, C benchmark programs are compiled using IMPACT-I and DRESC. DRESC uses an eXtensible Markup Language (XML) architecture description file during compilation. This same architecture description is used to estimate the area using an area model. Note that the area model is affected by the number of contexts, given by the iteration interval (II) produced by DRESC. After the instructions per cycle (IPC) is extracted from DRESC's output files and the area is estimated, we divide the IPC by the area estimate to compute the *number of instructions per cycle per unit area.*

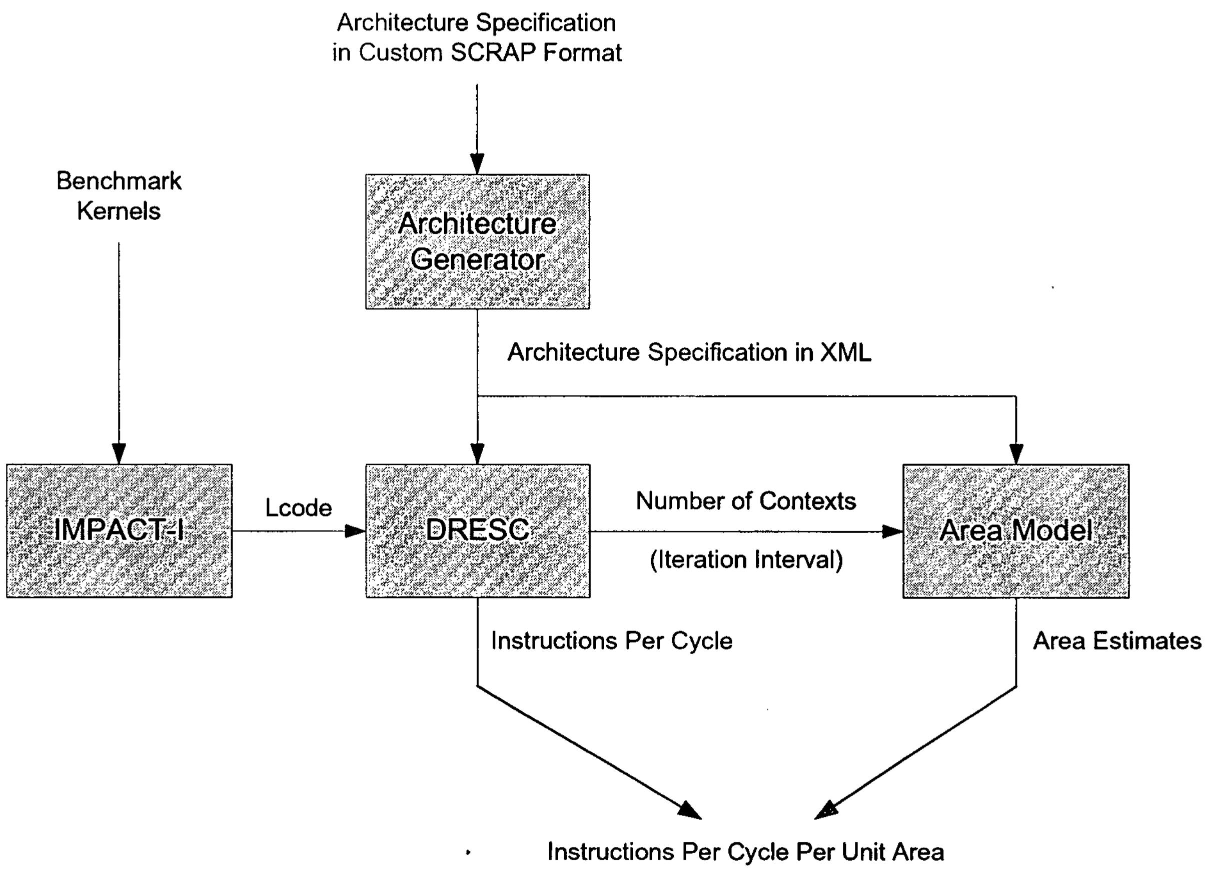


Figure 4.1: Experimental Flow

4.1.2 SCRAP

Steve's Coarse-grained Reconfigurable Architecture Program (SCRAP) is a tool that was developed to ease the specification of new architectures and to provide an area estimate. SCRAP contains an architecture generator and part of an area model. It takes as input a description in a custom SCRAP format and converts it into the standard XML architecture description that is recognizable by DRESC. Manipulating the architecture in its own internal format, SCRAP then produces a VHDL description of the architecture. Enhancements made as part of this work include the generation of the XML architecture description, the addition of global register files to the VHDL description, a transition to a modular VHDL description, and part of the area model.

With the new modular VHDL description, only the top-level entity is customized for each architecture, while the modules representing parameterized functional units, multiplexers, configuration memory, and registers are unchanged.

4.2 VHDL Model

A Very High Speed Integrated Circuits Hardware Description Language (VHDL) model was developed for the ADRES-based architecture and included in SCRAP. The VHDL model was specified at the register-transfer level (RTL), and synthesized using Synopsys using generic standard-cells. This section describes how the functionality of our architectures was specified.

4.2.1 Functional Units

The ALU in each functional unit was implemented using a switch-case block that performed different operations on the input data based on the operation code. While arithmetic, logical, shift, compare, predicate, and multiplication instructions were easy to specify, the division and remainder operations were left unimplemented. There are two reasons for this: (1) there is no standard synthesis of a divider available from Synopsys; and (2) the paper describing ADRES [3] does not mention the presence of dividers. For divisions by powers of two, an arithmetic shift right instruction would suffice. For other types of divisions, subroutines can use shifts, comparisons, and subtractions to perform long division one bit at a time. Note that certain commercial DSP processors do not perform integer division in a single cycle, but have single-cycle instructions that can be used to perform long division one bit at a time [83, 84, 85].

Immediate data for each functional unit was stored in 32 bits of context memory. With several contexts and 16 or 64 functional units, one can see that this quickly becomes a significant memory overhead. In fact, an estimated 12 to 23% of the chip area was occupied by this context memory for immediate data according to Table 1.1.

4.2.2 Multiplexers

Multiplexers were specified behaviourally in VHDL; Synopsys infers multiplexers and implements them. Multiplexers were used frequently as part of larger blocks such as the input multiplexers to functional units or the output multiplexers of register files and context memory. As for control signals that depend on multiplexers, constant 0 and constant 1 inputs were added to allow DRESC to directly specify the value of the control bit, such as the write enable signal for a register file.

4.2.3 Register Files

Register files were implemented using D flip-flops, and the write and read logic using logic gates and multiplexers. Static random-access memory (SRAM) cells were not used, as it was not clear how to specify them in register-tranfer level (RTL). This will have a large impact on the area of the register files, since a D flip-flop may require 60 to 80 μm^2 compared to an SRAM cell which has an area that is closer to 10 μm^2. For register files with multiple write ports, it was assumed that each individual register would only be written by one write port at a time, as this is a restriction that must be enforced at compile time.

4.2.4 Context Memory

Context memory was implemented using D flip-flops arranged in the form of long chains of shift registers, one chain for each context. Each context is selected by a large multiplexer that chooses a set of configuration bits from the shift registers. The select bits for this multiplexer that control the context are modeled as input ports to the chip; control circuitry is presently not implemented on the chip. Alternatively, SRAM cells could have been used to store the configuration bits instead of D flip-flops. Since SRAM cells are smaller than D flip-flops, this could result in area savings. Moreover, the output of the multiplexers are not currently registered, whereas adding a pipeline stage could potentially reduce 1 ns from the critical path at the cost of a small increase in area. The pipeline stage would be a good future enhancement, since the context changes every clock cycle, it would reduce glitches and hence glitch power, and the operation of the context memory is unaffected. However, it does not affect the functionality and so we leave it for now.

4.2.5 Benchmarks

Common DSP functions, including a fast Fourier transform (FFT), an inverse discrete cosine transform (IDCT) solver, and an MPEG-2 decoder were used as benchmark programs. These programs were compiled using IMPACT-I [73] and loops with high amounts of loop-level parallelism were extracted. Twenty such loops were obtained.

4.3 Evaluation Metrics and Techniques

This section describes the three metrics that we will use to evaluate our architectures: area, performance, and instructions per cycle per unit area.

4.3.1 Area Estimation

To estimate the area, we used a parameterized area model. In [79], the VHDL models of potential architectures were synthesized using Synopsys, and the area metric was obtained from this synthesized fabric. In this research, we consider larger architectures, and thus the flat synthesis of these architectures is not feasible. Instead, we use the following approximation. For each component in the reconfigurable fabric (multiplexer, configuration memory, register, etc.) we created a parameterized model based on measurements obtained from Synopsys for a 0.18μm TSMC process. By adding the area contributions of each architectural component, we estimated the overall area of each potential architecture. To verify the accuracy of this approximation, we constructed and synthesized six full architectures as in [79]. In selecting these six architectures, we attempted to "span" the design space as much as possible. We then compared the post-synthesis areas obtained from Synopsysy to the area estimates from our approximate scheme. As shown in Figure 4.2, the area estimates correlate well with the post-synthesis results.

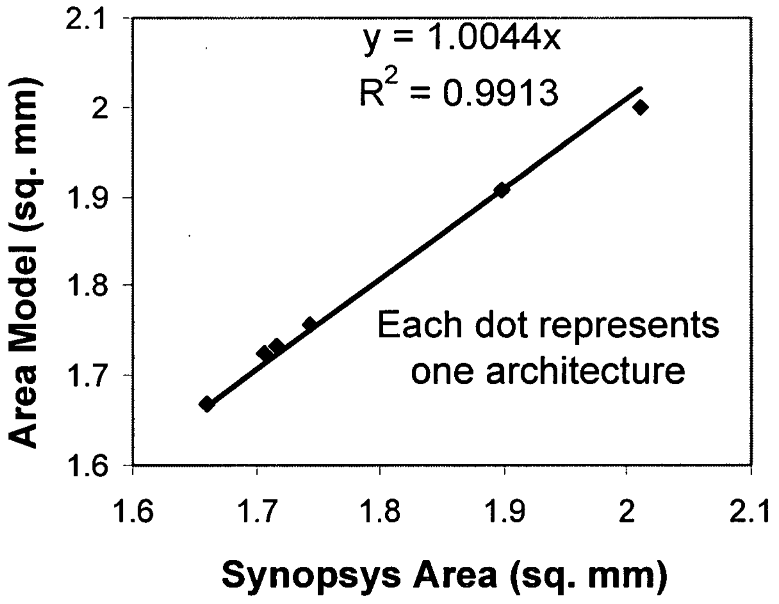


Figure 4.2: Validation of Area Model

There are at least four limitations to the area model. First, since branch, load, store, and other instructions were not implemented in the VHDL model, 0.02 mm^2 was added to the total chip area for each functional unit that required these instructions. Second, the absence of dividers in the VHDL model means the area is underestimated by our area model. However, since this affects all of the architectures that we compare, the relative area efficiency of the different register file architectures can still be evaluated fairly. Third, register files and context memory are typically constructed from SRAM cells, instead of D flip-flops. Fourth, it does not take into consideration the possibility of sharing logic between the read and write ports [86].

An adjusted area model was developed to address the third and fourth limitations. We introduce adjusted areas where the D flip-flops that occupy 60-80 μm^2 are replaced with hypothetical SRAM cells that are approximately the same size as a NAND gate and occupies only 10 μm^2. The adjusted areas will also model the sharing of logic between read and write ports by assuming that read ports do not require extra area when an equivalent number of write ports are already present. These adjustments to the area are intended to err towards lower areas so that the adjusted areas can provide a lower bound for the post-synthesis area of the register files. However, the adjusted areas are not intended for approximating areas of full custom layouts of these architectures, which may be twice as small, but the adjusted areas reflect a smaller percentage contribution due to memory, which is likely in full custom layouts. With the original area model providing an upper bound for the area contributions of register files and the adjusted area providing a lower bound, we can be confident that conclusions, that we find concerning register files, which hold for both area models are: 1) valid for register files constructed using SRAM cells; and 2) robust towards changes to the area model.

4.3.2 Performance Measurements

To quantify the performance of the device, we measured the maximum instructions per cycle (IPC) that could be achieved for each loop on each architecture. In simple pipelined reduced instruction set computer (RISC) processors, the maximum possible instructions per cycle is one. This can be achieved by starting the execution of a new instruction at each successive clock cycle. In superscalar processors and very long instruction word (VLIW) processors, the instructions per cycle can often exceed one because they have several functional units to execute instructions in parallel. The reconfigurable array in ADRES usually achieves much higher instructions per cycle.

This does not take into account differences in the clock cycle time of each architecture (different architectural options might have slightly different cycle times). However, we believe that the delay of the ALU dominates the critical path delay. The multiplication operation, in particular, consumed 7.4 ns of the 10.4 ns in the critical path of the baseline 4x4 array architecture. Since we do not modify the ALU in this thesis, we expect that all architectures investigated here have similar critical path delays.

There are at least three ways to lower this minimum clock cycle time imposed by the multiplier. First, the multiplier can be pipelined. However, pipelining increases the complexity of the compiler, control logic must be added, and conditional execution necessitates the flushing of the pipeline [14]. These difficulties must be overcome to make pipelining possible and are beyond the scope of this thesis, which deals specifically with the architectures based on the ADRES architecture, which has single-cycle multipliers. Second, the multiplier can be implemented using custom layout to reduce the delay. However, custom layout can also be applied to all other parts of the circuit, so that the cycle time decreases by a similar factor. In such a situation, the relative contribution of the multiplier to the critical path delay would remain quite large. Third, multipliers can be removed altogether. However, the experiments in this thesis pertain exclusively to ADRES-based architectures that have multipliers, so this third alternative would not be applicable to the framework chosen for this thesis.

While performance enhancements can be made to the architecture model presented in this chapter by using custom layout, the relative performance is more important for this thesis, where architectural features are compared and studied.

4.3.3 Instructions Per Cycle Per Unit Area

We use the metric, number of instructions per cycle per unit area, to rank our architectures. The same metric, the number of instructions per cycle per mm^2, or IPC/mm^2, was used by Wilton et al. [79] in the evaluation of the interconnect architectures for ADRES-based architectures. It is similar to the functional density (D) metric, 1/(area x time), introduced by Wirthlin et al. [87] and the computational density metric, (number of bit-operations)/(area x time), used for evaluating experiments with PipeRench [2]. Another metric used by Lee et al., performance/cost, not only divides the instruction count by area and time, but also takes cache misses into account [88].

This metric is similar to area-delay product, which is used to determine the best area-speed tradeoff [89]. However, area-delay product considers the minimum clock cycle time but not the number of cycles required to execute a program.

4.4 Summary

In this chapter, the experimental flow was presented in conjunction with a tool called SCRAP and an associated VHDL model. Then an area model used to estimate area, the measure of performance that we are interested in, and the metric that balances performance and area, instructions per cycle per unit area, were discussed.

Chapter 5
RESULTS

In this chapter, we experimentally investigate the effects of changing the number of registers in each register file, the number of ports on each register file, and the manner in which the register files are connected to the functional units.

5.1 Global Register Files

This section presents results for the global register files.

5.1.1 Degree of Connectivity

We first consider the manner in which the these register files are connected to the rest of the device. In the baseline architecture, the global register files are connected to the top row of FUs only. Each FU in this top row has dedicated read and write ports to each of the two global register files. Table 5.1 compares the performance and area of this baseline 4x4 array architecture to an architecture in which more than just the top row of FUs are connected to each global register file. The number of read and write ports on the global predicate file were held constant at four, while on the global data register file, the number of read ports was held constant at eight and the number of write ports was held constant at four. Thus, as more FUs are connected to the register file, these FUs need to connect shared signals from the read ports to their input multiplexers and connect their output signals to multiplexers at the write ports. In Table 5.1, if *x* FUs are connected to the register file, there are *x/4* FUs sharing each port.

As the table shows, increasing the reach of the global register file decreases the number of instructions per cycle required to complete the loop (averaged over all benchmark loops). This makes sense; the more paths there are between the global register files and the computing elements, the easier it is for DRESC to find an efficient schedule. The area required to implement the architecture goes up as the number of connections increases. As more connections are available, DRESC is able to find schedules which have a lower iteration interval (II), meaning less context memory is required (on average). This decrease in area is more significant than the small increase in size in the input multiplexers of the FUs and the additional area required for multiplexers at the write ports of the global register files. Combining the area and IPC results, we see that the most significant incremental improvement is achieved when the degree of connectivity is increased from four FUs to eight FUs, but connecting the global register files to as many FUs as possible results in the best IPC per unit area (this ratio is recorded in the fifth column of Table 5.1). This can also be observed from the results obtained using the adjusted area model in the two rightmost columns. Therefore, we recommend that all sixteen FUs in the 4x4 array be connected to the global register files by sharing eight read ports and four write ports on the data global register file and four read ports and four write ports on the predicate global register file.

Table 5.1: Impact of Changing the Number of FUs that can Connect to the Global Register Files (4x4 Array)

Number of FUs	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
4 FUs (Baseline)	7.23	7.2	2.10	3.45	1.44	5.03
8 FUs	9.24	5.9	1.91	4.85	1.24	7.43
12 FUs	9.58	5.8	1.90	5.05	1.23	7.76
16 FUs	9.74	5.7	1.90	5.14	1.23	7.91

Table 5.2: Impact of Changing the Number of FUs that can Connect to the Global Register Files (8x8 Array)

Number of FUs	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
8 FUs (Baseline)	13.59	3.8	6.48	2.10	3.74	3.63
16 FUs	19.68	2.7	5.81	3.39	3.07	6.40
24 FUs	20.40	2.5	5.73	3.56	3.00	6.81
32 FUs	20.38	2.5	5.76	3.54	3.02	6.76
40 FUs	20.40	2.5	5.79	3.53	3.03	6.73
48 FUs	20.88	2.5	5.78	3.61	3.02	6.91
56 FUs	21.90	2.4	5.71	3.84	2.94	7.44
64 FUs	20.85	2.5	5.83	3.58	3.06	6.82

We repeated the experiment for an 8x8 array and obtained the results in

Table 5.2. In this case, the number of read and write ports in the each register file was held constant at eight (except for the data global register which has 16 read ports). As with the 4x4 array, we see that the most significant incremental improvement is achieved when the degree of connectivity is increased from eight FUs to sixteen FUs, but connecting the global register files to 56 FUs results in the best IPC per unit area. This table shows the same trends as Table 5.1, with the exception of the last entry, which corresponds to an architecture in which the global register files are connected to all 64 FUs. In this case, DRESC found schedules that are actually worse than the 56 FU case. We have observed that often the optimization algorithm within DRESC has problems dealing with architectures with too much flexibility; a better understanding of this is on-going work. However, with the exception of this point, the

conclusion is the same: the more connected the global register file, the better. Therefore, we recommend that 56 of the 64 FUs in the 8x8 array be connected to the global register files by sharing sixteen read ports and eight write ports on the data global register file and eight read ports and eight write ports on the predicate global register file.

5.1.2 Number of Ports

In this subsection, we vary the number of ports on the global register files. Note that the number of ports shown in the following tables are the number of predicate read ports, predicate write ports, and data write ports. There are twice as many data read ports as stated in the tables.

Table 5.3: Impact of Changing the Number of Ports on the Global Register Files (4x4 Array)

Number of Ports	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
4* (Baseline)	7.23	7.2	2.10	3.45	1.44	5.03
8*	9.38	5.8	2.06	4.56	1.32	7.09
12*	9.57	5.7	2.21	4.32	1.40	6.81
16*	9.57	5.7	2.40	3.99	1.51	6.33

* There are twice as many read ports on the data global register files.

Table 5.4: Impact of Changing the Number of Ports on the Global Register Files (8x8 Array)

Number of Ports	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
8* (Baseline)	13.59	3.8	6.48	2.10	3.74	3.63
16*	21.40	2.5	6.69	3.20	3.56	6.01
24*	22.29	2.3	7.65	2.91	4.12	5.41
32*	22.56	2.3	8.72	2.59	4.78	4.72
40*	23.35	2.2	9.71	2.41	5.37	4.35
48*	23.35	2.2	10.77	2.17	6.02	3.88
56*	23.35	2.2	11.83	1.97	6.68	3.50
64*	23.35	2.2	12.89	1.81	7.34	3.18

* There are twice as many read ports on the data global register files.

Table 5.3 shows the results for a 4x4 array. In these results, all FUs that are connected to the global register file have a set of dedicated read and write ports. The greatest improvement in the IPC per unit area is observed when the number of ports is doubled compared to the baseline case. However, compared to Table 5.1, we can see that there is very little increase in the achievable IPC (in fact, it is smaller in some cases, likely because of DRESC's difficulty dealing with very flexible architectures as described above). There is a significant area penalty, however, in increasing the number of ports. Therefore, we conclude that increasing the number of read and write ports to match the number of FUs connected to the global register file is not a good idea. Instead, sharing ports between multiple FUs results in a better performance per unit area. Table 5.4 shows the results for an 8x8 array; the same trends apply. The adjusted area model also confirms these conclusions. Therefore, we recommend no increase in the number of read and write ports on the global register files, but to share existing read and write ports in the manner described in Section 5.1.1.

5.1.3 Number of VLIW FUs

The FUs in the top row of the reconfigurable array are connected to the global register files with read and write ports. These FUs also form part of the general-purpose VLIW processor and have access to main memory through load and store instructions. When we increase the number of FUs that have both the global register file ports and main memory ports, we give more FUs the same functionality as the top row FUs. Therefore, we refer to any FU connected to read and write ports on the global register files and main memory ports as "VLIW FUs". Although the implementation of the read and write ports for main memory are not defined in this thesis, the VLIW FUs can read and write from main memory using load and store instructions. The results of increasing the number of VLIW FUs are shown in Table 5.5 for the 4x4 array and in Table 5.6

for the 8x8 array. The "empty" interconnect architecture, as defined in Section 3.3.1, is also studied here because an architecture where every FU is connected to the global register files and to main memory without any other interconnect is just like a typical VLIW processor.

Table 5.5: Impact of Changing the Number of FUs that can Connect to the Global Register Files and Main Memory (4x4 Array)

Interconnect Architecture	Number of VLIW FUs	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
Closest [79]	4 FUs (Baseline)	7.23	7.2	2.10	3.45	1.44	5.03
	8 FUs	9.74	5.7	2.11	4.62	1.38	7.07
	12 FUs	9.90	5.5	2.36	4.21	1.55	6.40
	16 FUs	9.90	5.5	2.62	3.78	1.73	5.71
Empty	16 FUs	9.66	5.9	2.53	3.81	1.66	5.80

Table 5.6: Impact of Changing the Number of FUs that can Connect to the Global Register Files and Main Memory (8x8 Array)

Interconnect Architecture	Number of VLIW FUs	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
Closest [79]	8 FUs (Baseline)	13.59	3.8	6.48	2.10	3.74	3.63
	16 FUs	21.30	2.5	6.85	3.11	3.72	5.73
	24 FUs	24.59	2.2	7.90	3.11	4.37	5.62
	32 FUs	27.75	1.9	8.94	3.10	5.02	5.53
	40 FUs	30.33	1.8	11.08	2.74	6.36	4.77
	48 FUs	30.99	1.8	11.21	2.76	6.49	4.77
	56 FUs	30.99	1.8	12.42	2.49	7.30	4.25
	64 FUs	30.99	1.8	13.63	2.27	8.11	3.82
Empty	64 FUs	22.54	2.4	13.63	1.65	8.15	2.77

Table 5.5 shows the same general trend as Table 5.3. As for FUs which are connected to global register file ports and main memory ports, the IPC of the 4x4 array increases to just below 10. As

well, the area remains fairly constant from four VLIW FUs to eight VLIW FUs. When the number of VLIW FUs is increased even more, the area increases, but the IPC does not increase significantly. Clearly all of the architectures outperform the baseline architecture. While the results in this subsection give better performance than in Subsections 5.1.1 and 5.1.2, comparable IPC can be achieved in these previous subsections with smaller area. This is also confirmed by the results obtained using the adjusted area model.

In the 8x8 array case, Table 5.6 shows the same trend as Table 5.4. The IPC increases as the number of VLIW FUs increases. When the number of VLIW FUs increases to 40 FUs, the IPC begins to saturate. Having more than 40 VLIW FUs increases the area without any significant improvement to the IPC. The most striking difference between Table 5.6, where the number of VLIW FUs increases, and Table 5.4, where the number of FUs connected to global register file ports increases, is the maximum possible IPC. With many VLIW FUs with both global register file ports and main memory ports, architectures can achieve an IPC of 30, while architectures having many FUs with global register file ports only can reach an IPC of 23, which is 25% smaller. This 33% performance boost from adding main memory ports confirms the observation by Ienne et al. [75] that having only one memory port results in 15-20% slower performance than having many memory ports. While there is a corresponding increase in area, making it less favourable than the shared connectivity discussed in Subsection 5.1.1, increasing the number of VLIW FUs is still an option for increasing performance if area is allowed to increase.

Table 5.5 and Table 5.6 show the results for the "empty" interconnect architecture where every FU is a VLIW FU. As shown in the tables, this architecture has a higher IPC than the baseline architecture. In the 4x4 array results in Table 5.5, the architecture with the "empty" interconnect architecture performs worse than having 8, 12, or 16 FUs with the "closest" interconnect architecture. The area overhead is also greater than the 8 and 12 FU cases, but smaller than the 16 FU case. In the 8x8 array in Table 5.6, the architecture with the "empty" interconnect architecture occupies the most area and has the worst IPC per mm^2. Therefore, the "empty" interconnect architecture is inferior to the "closest" interconnect architecture for comparable 8x8 arrays because it requires much more area. This comparison between the "closest" and "empty" interconnect architectures demonstrates how important the programmable interconnect is to achieving high performance, especially for arrays with larger numbers of FUs. This is also confirmed by the results obtained using the adjusted area model.

We recommend no increase to the number of VLIW FUs beyond the baseline case, where the top row of FUs in the reconfigurable array have read and write ports on the global register files and read and write ports for main memory. As well, we recommend that the "empty" interconnect architecture not be used for 4x4 and 8x8 array architectures.

5.1.4 Register File Size

Figure 5.1 shows the impact of the number of registers in each global register file on the performance and area of a 4x4 array. For a very small number of registers, DRESC is unable to find efficient loop implementations, leading to low IPC values and many contexts (and hence large area). As the number of registers increases, the IPC increases; however, beyond 16, the increase is outweighed by the extra area required for the registers. With 12 or 16 registers in each register file, results in terms of IPC per unit area within 2% of the maximum achieved when there are 14 registers.

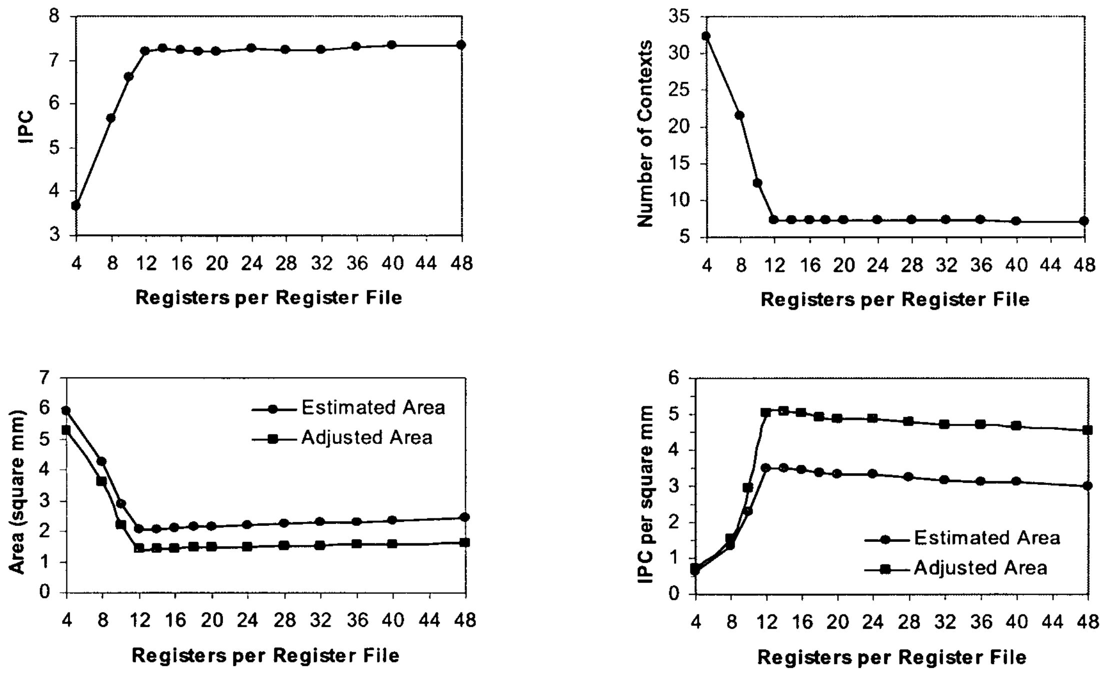


Figure 5.1: Impact of Changing the Number of Registers in Each Global Register File in the 4x4 Array

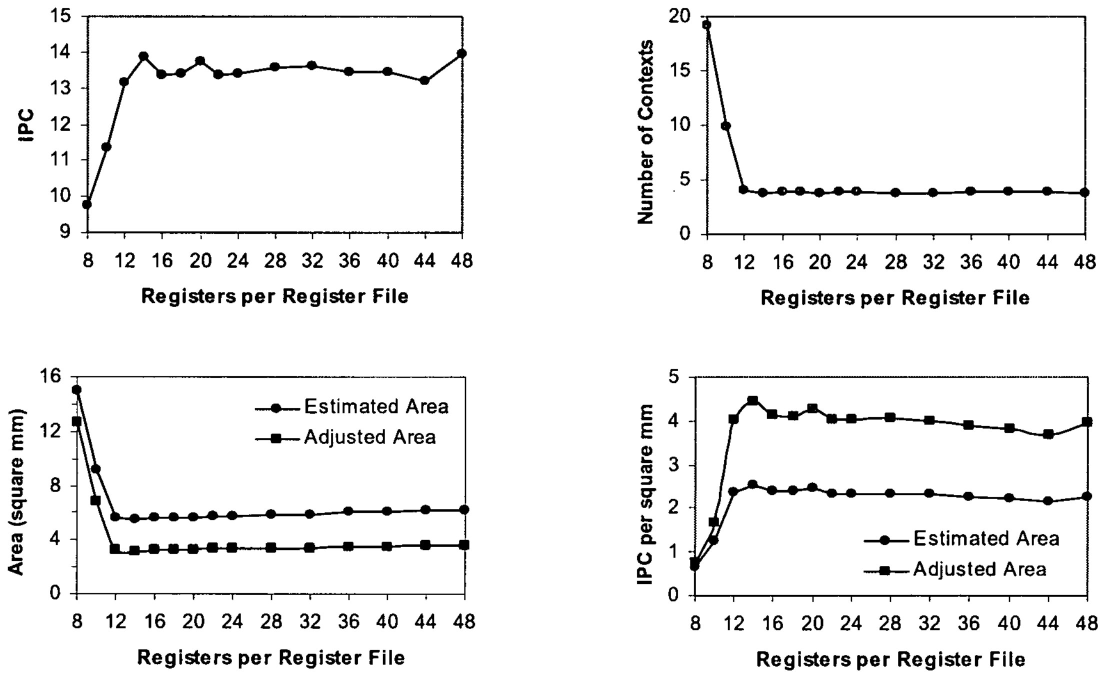


Figure 5.2: Impact of Changing the Number of Registers in Each Global Register File in the 8x8 Array

Figure 5.2 shows the results for an 8x8 array. In this case we see that 14 registers in each global register file are sufficient to achieve peak performance. Intuitively, we might have thought that a larger array would prefer a larger number of global registers than a smaller array, yet the best choice of register file size is the same in both cases. This may be because in the larger array, there are more FUs, and hence more local register files. The larger number of local register files may mean that smaller global register files are suitable.

We recommend having 14 registers in each global register file for both the 4x4 and 8x8 array architectures.

5.2 Local Register Files

This section presents results for the local register files. As shown in Figure 3.2, there are two local register files in each FU: one 32-bit wide register file to store data values; and one 1-bit wide register file to store predicate values.

5.2.1 Degree of Connectivity

As described in Chapter 3, in the baseline architecture, the local register files are connected to the computing resources as follows. The input of each local register file (within a FU) is driven by the unregistered output of the ALU within the local FU. The read port of each local register file, however, is connected to six sinks: the two inputs of the ALU in the local FU, and one input in each of four neighbouring FUs (the four FUs diagonally adjacent to the local FU).

We evaluated the performance and area for this baseline architecture and three other architectures. Architecture 1, shown in Figure 5.3, is less connected than the baseline architecture, shown in Figure 3.7; in Architecture 1, each local register file read port drives *only* the two inputs to the local ALU; it does not drive ALUs in neighbouring FUs. Figure 5.4 shows Architecture 2, in which the write port of each local register file are connected to the output of the local ALU as well as the outputs of four neighbouring ALUs, while the read port of the register file is connected to only the local ALU. Architecture 3 is the most flexible; it contains the flexible register write port connection pattern of Architecture 2 and the flexible register read port connection pattern of the baseline architecture.

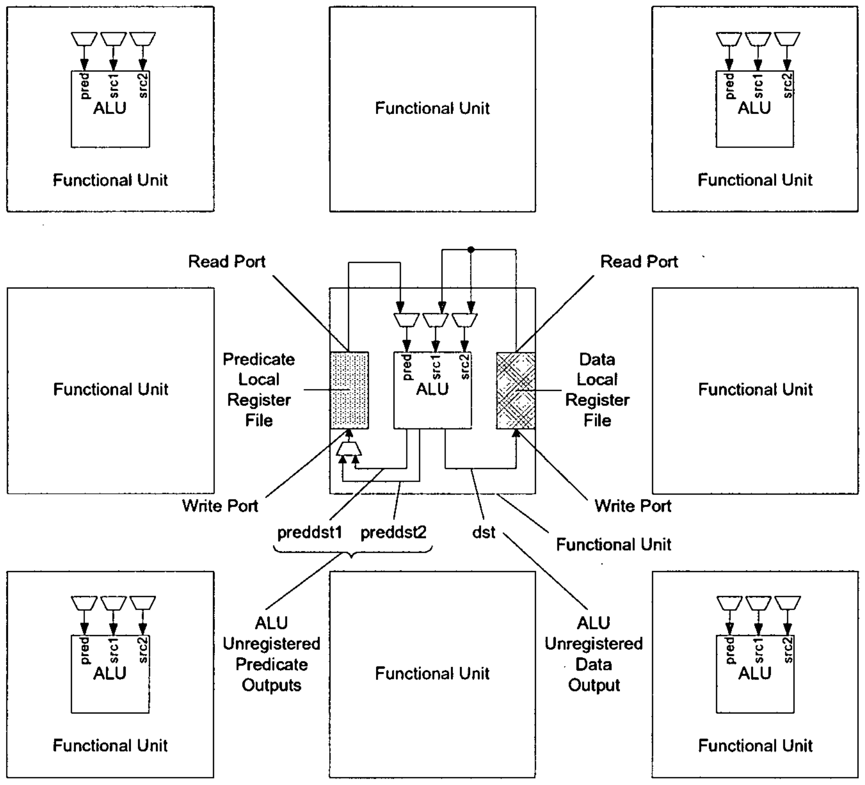


Figure 5.3: Local Register File Connection Pattern for Architecture 1

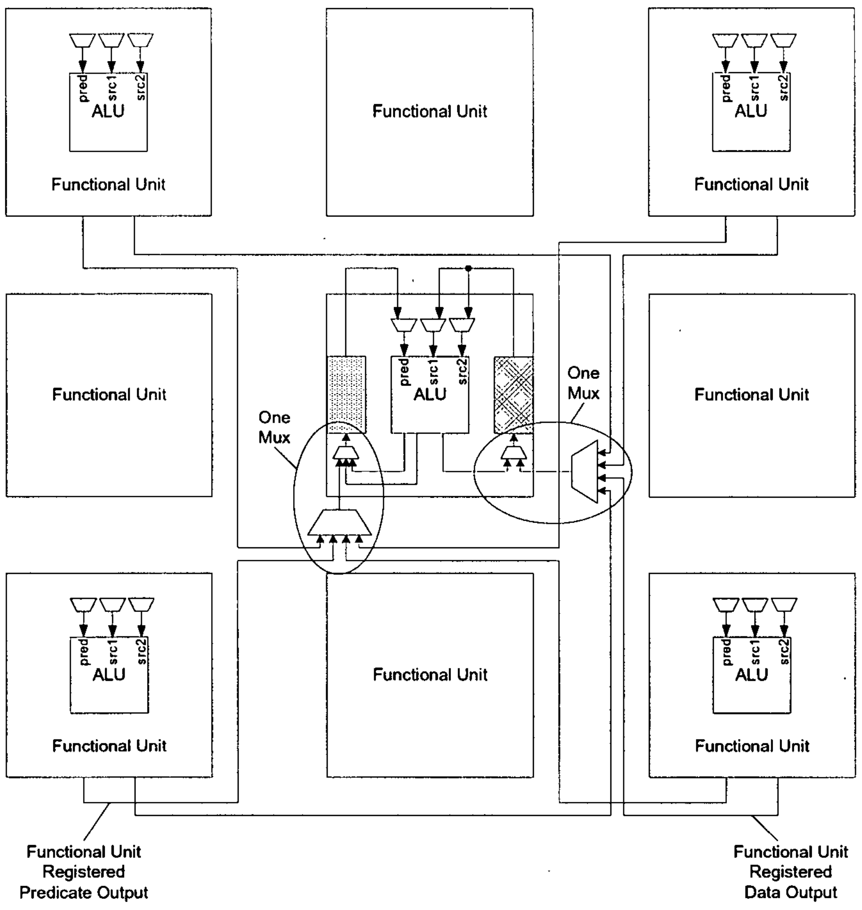


Figure 5.4: Local Register File Connection Pattern for Architecture 2

From Table 5.7, with the exception of Architecture 1, we can see that all architectures provide roughly the same IPC, and there is only a small difference in the area. Results for an 8x8 array are shown in Table 5.8. In this case, there was a larger improvement in performance when both the read port and the write port were connected to multiple FUs. These observations also apply to the results obtained using the adjusted area model.

Table 5.7: Impact of Changing the Local Register File Connection Pattern (4x4 Array)

Local Register File Connection Pattern	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
Architecture 1	6.01	13.3	3.03	1.98	2.36	2.55
Baseline	7.23	7.2	2.10	3.45	1.44	5.03
Architecture 2	7.25	7.2	2.15	3.37	1.47	4.94
Architecture 3	7.21	7.2	2.21	3.26	1.53	4.73

Table 5.8: Impact of Changing the Local Register File Connection Pattern (8x8 Array)

Local Register File Connection Pattern	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
Architecture 1	11.95	4.4	6.67	1.79	3.93	3.04
Baseline	13.59	3.8	6.48	2.10	3.74	3.63
Architecture 2	13.16	4.0	6.68	1.97	3.85	3.42
Architecture 3	14.29	3.6	6.67	2.14	3.84	3.72

Comparing the global register file results in Table 5.1 and

Table 5.2 to the local register file results in Table 5.7 and Table 5.8, we see that the impact of increasing the connectivity of the global register file has a much larger impact on the performance of the array than does increasing the connectivity of the local register files. It is clear that the global register files, which serve as the storage locations for input and output variables for each loop, as well as a central repository, play a critical role in the storage and sharing of data. This is less true of the local register files.

We recommend that the degree of connectivity found the baseline 4x4 array architecture be used, which have the flexible connection pattern for the read ports of the local register files. For the

8x8 array architecture, we recommend using the flexible connection patterns for both read ports and write ports of the local register files, which characterizes Architecture 3.

5.2.2 Number of Ports

In the previous results, although the register file read and write ports may have multiple sources and sinks, there is only one read port and one write port on each register file. Table 5.9 shows the impact of changing the number of read ports in the baseline 4x4 array architecture. Since, in the baseline architecture, the read port drives up to six ALU inputs, the register file could potentially make use of up to six read ports. As Table 9 shows, however, one read port provides sufficient performance. Table 5.10 shows that this also holds for an 8x8 array, but two read ports is also a good choice. These observations also apply to the results obtained using the adjusted area model, except that having two read ports is clearly the better choice for the 8x8 array according to the adjusted area model. Therefore, we recommend having one read port per local register file in the 4x4 array architecture and two read ports per local register file in the 8x8 array architecture.

Table 5.9: Impact of Changing the Number of Read Ports on the Local Register Files (4x4 Array)

Number of Read Ports	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
1 (Baseline)	7.23	7.2	2.10	3.45	1.44	5.03
2	7.20	7.2	2.18	3.31	1.44	4.99
3	7.16	7.3	2.26	3.18	1.51	4.75
4	7.28	7.2	2.30	3.17	1.54	4.74
5	7.36	7.0	2.35	3.13	1.57	4.68
6	7.22	7.2	2.43	2.97	1.64	4.40

Table 5.10: Impact of Changing the Number of Read Ports on the Local Register Files (8x8 Array)

Number of Read Ports	IPC	No. of Contexts	Area (mm^2)	IPC / mm^2	Adjusted Area (mm^2)	IPC / mm^2
1 (Baseline)	13.59	3.8	6.48	2.10	3.74	3.63
2	13.90	3.7	6.61	2.10	3.68	3.78
3	13.76	3.8	6.84	2.01	3.84	3.58
4	13.59	3.8	7.07	1.92	4.00	3.39
5	13.73	3.8	7.23	1.90	4.10	3.35
6	13.55	3.8	7.42	1.83	4.26	3.18

5.2.3 Register File Size

Figure 5.5 and Figure 5.6 show the impact of the number of registers in the local register files on IPC, area, and IPC per unit area for the 4x4 and 8x8 arrays, respectively. As the figures show, only a small number of registers are required in each local register file. In the 4x4 array, two registers are sufficient according to area estimates obtained using the area model; using the adjusted area model, we find that having two, three, or four registers produce results within 1% of the maximum IPC per unit area. In the 8x8 array, having one or two registers per local register file both produces results within 1% of the maximum IPC per unit area according to the area estimates obtained using the area model; using the adjusted area model, having one to five registers per local register file produces results within 3% of the maximum IPC per unit area.

We recommend having two registers per local register file for both the 4x4 and 8x8 array architectures.

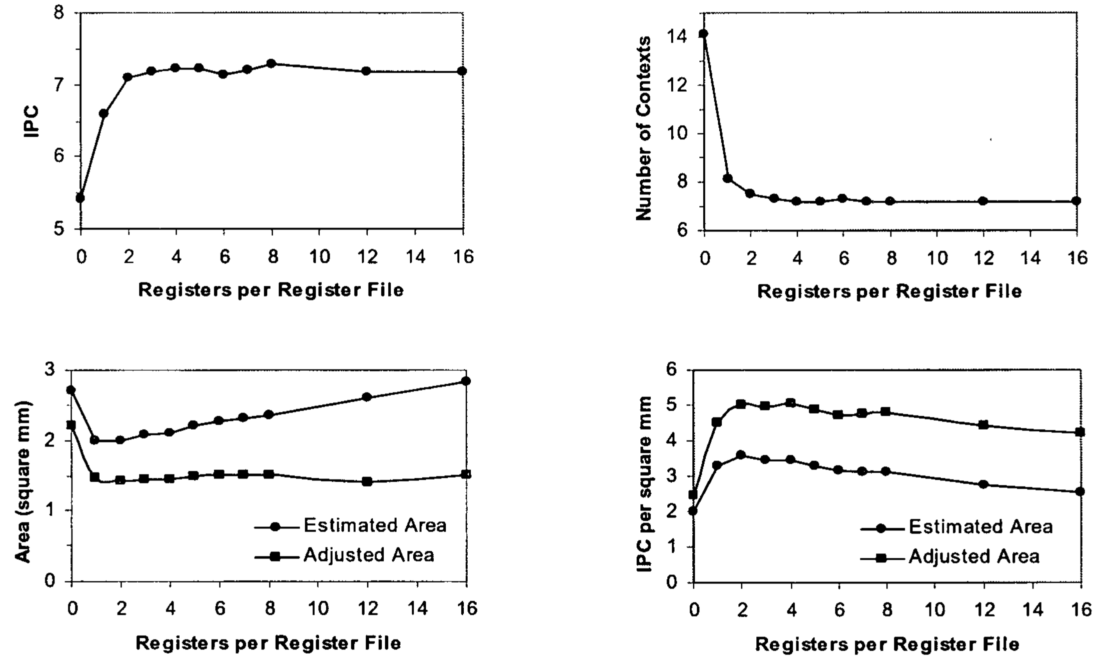


Figure 5.5: Impact of Changing the Number of Registers in Each Local Register File in the 4x4 Array

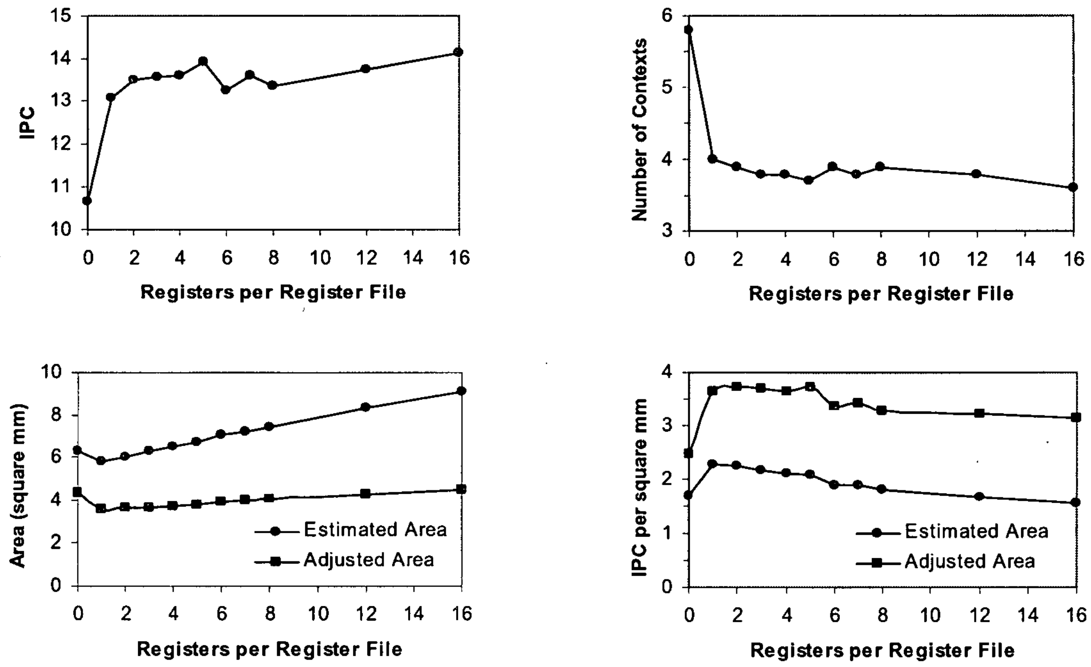


Figure 5.6: Impact of Changing the Number of Registers in Each Local Register File in the 8x8 Array

5.2.3.1 Interconnect and Register File Size

In this subsection, we will consider how the local register file size interacts with the interconnect architecture and the size of the reconfigurable array.

Two different interconnect architectures are studied: "closest" and "full". Since the previous experiment suggested that only two registers per local register file were required to achieve the maximum IPC, the architectures tested had 0, 1, or 2 registers per data local register file and 0, 1, or 2 registers per predicate local register file. We tested the following combinations: 2 registers per data local register file and 2 registers per predicate local register file; 2 and 1; 2 and 0; 1 and 1; 1 and 0; and 0 and 0. The number of rows and columns of functional units in the reconfigurable array was also varied. The sizes studied were 4x4, 4x6, 4x8, 6x4, 6x6, 6x8, 8x4, 8x6, and 8x8. All of these architectures had four read ports for each local data register file and local predicate register file. While these different array sizes provided a large family of architectures, the focus is on the interaction between the interconnect architecture and the size of the local register files.

Figure 5.7 shows the results. This figure was obtained by plotting the architectures in a scatterplot, where the x-axis represents the area and the y-axis represents the average achievable IPC. Two data points have been removed because the results for the two architectures were incomplete due to DRESC's inability to produce the output file. Figure 5.7 shows that the choice between the "closest" and "full" interconnect architectures affects IPC and area in a way that applies to different sizes of the local register files and the size of the reconfigurable array. From the scatterplot, it appears that "full" architecture achieves higher IPC than the "closest" architecture. This confirms results published by Wilton et al. in [79].

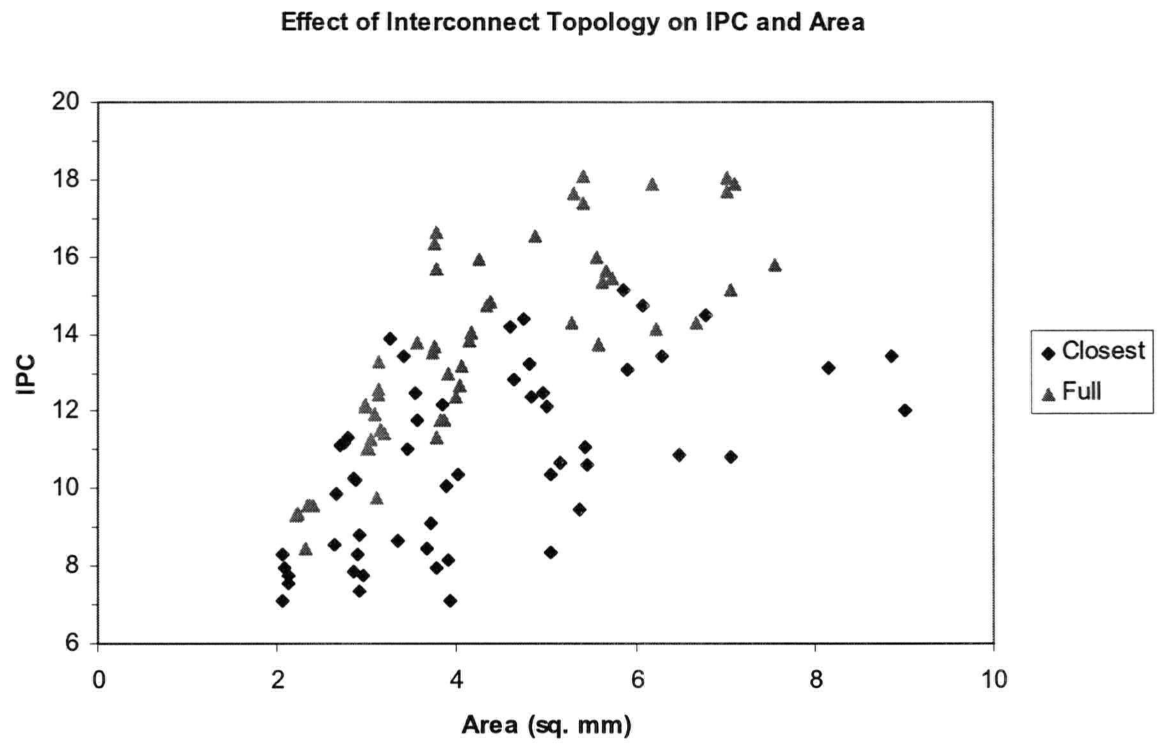


Figure 5.7: Effect of Interconnect Architecture on IPC and Area

Figure 5.8 shows results for a superset of the architectures shown in Figure 5.7 with the same two data points omitted. The architectures represented in the scatterplot are classified by the size of the data and predicate local register files. Similar to the previous set of architectures, these architectures have array sizes of 4x4, 4x6, 4x8, 6x4, 6x6, 6x8, 8x4, 8x6, and 8x8. Some of these architectures have the full interconnect architecture and some have the closest interconnect architecture. As well, each local register file has four read ports. Unlike the architectures in

Figure 5.7, some of these architectures have write ports on the global data register file and the global predicate register file. Like the baseline architecture, some of these architectures have one row of VLIW FUs, while others have more VLIW FUs, as discussed in Subsection 5.1.3. The scatterplot shows that the area and performance are governed by factors such as array size and interconnect to a greater degree than by the sizes of the local register files. It also suggests that the best sizes of local register files vary depending on these other factors.

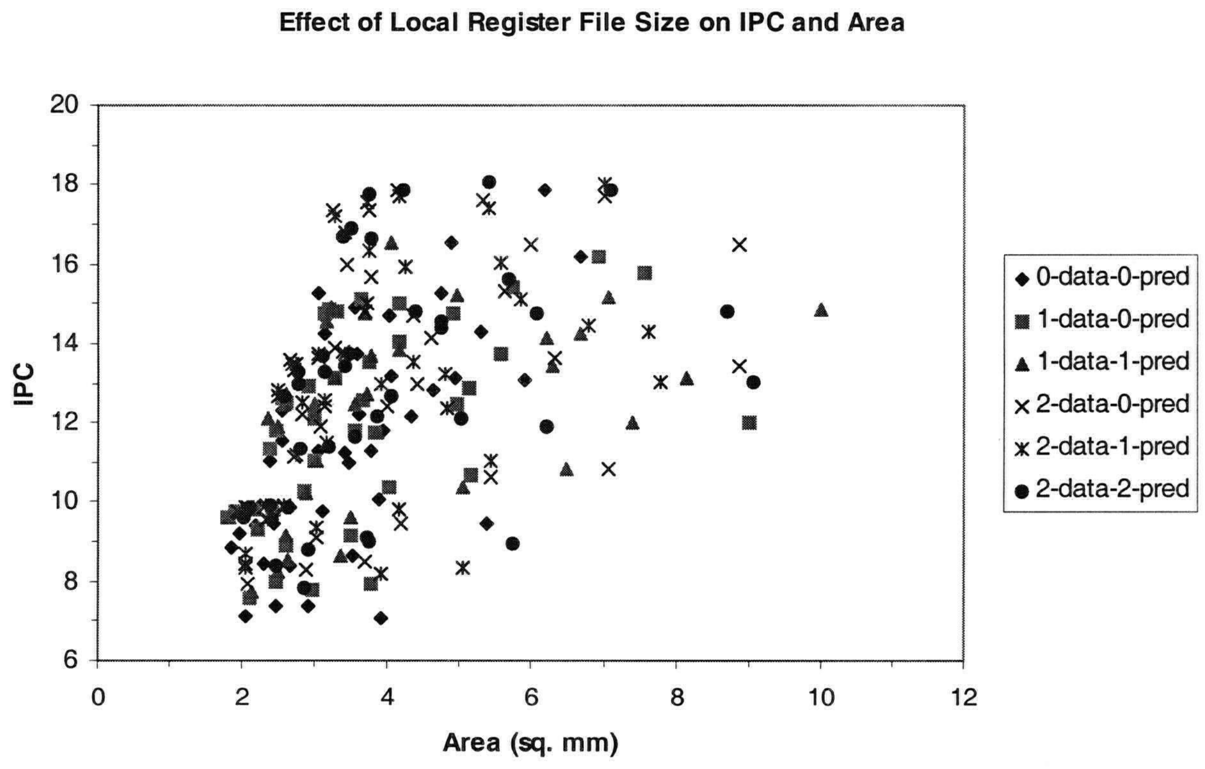


Figure 5.8: Effect of Local Register File Size on IPC and Area

5.2.3.2 Register Utilization

It is surprising that such a small number of local registers should be included within each register file, especially considering the larger number of local registers (4 to 128) reported for other architectures (see Table 5.11). To gather insight into this, the register utilization of the local register files was extracted from the loop scheduling and mapping information, and recorded in Table 5.12 and Table 5.13. These tables show the total number of variables stored in all of the local register files, the average number of variables stored in each local register file, and the peak number of variables stored in any single local register file in the 4x4 and 8x8 arrays. These results were gathered for an architecture with 128 registers per local register file, so as to maximize the local register file utilization. DRESC failed to produce one of the output files for benchmarks one, three, and five for the 4x4 array architecture, so the corresponding slots in Table 5.12 indicate n/a.

Table 5.11: Number of Local Registers in Each Configurable Functional Unit in Various Published Architectures

Published Architectures	Local Registers Per FU
MorphoSys [1]	4
REMARC [5]	12
PADDI [6]	12
PADDI-II [10]	12
DReAM [15]	16
Montium [14]	16
MaRS [9]	16
CRISP [13]	32
MATRIX [11]	128

Table 5.12 shows that the peak number of local data registers used from a single register file (the most highly utilized register file in the 4x4 array) is eight registers. It also shows that the local predicate register file usage is much lower, with at most four registers being used from a single register file. In this set of benchmarks, 0 to 53 data variables and 0 to 12 predicate variables are stored in local register files. The peak usage of a single register file ranges from 0 to 8 for the data register files and 0 to 4 for predicate register files. These low utilizations explain the why

increasing the size of the local register files does not significantly improve performance. Table 5.13 shows similar results for the 8x8 array, except that total number of variables used in the array is greater.

Table 5.12: Per Benchmark Register Utilization (4x4 Array)

Benchmark Kernels			Total Number of Variables in All Local RFs		Peak Number of Variables in a Local RF	
			data	pred	data	pred
LDL	ldl	1	n/a	n/a	n/a	n/a
	fnorm	2	2	0	1	0
FFT	radix4	3	n/a	n/a	n/a	n/a
IDCT 8x8	vertical	4	53	0	8	0
	horizontal	5	n/a	n/a	n/a	n/a
MPEG-2 Decoder	dequantize intra	6	2	1	1	1
	dequantize non-intra	7	3	1	1	1
	saturate	8	24	12	6	4
	add block non-intra	9	17	4	4	1
	add block intra	10	6	4	2	1
	clear block	11	0	0	0	0
	fast idct	12	27	0	4	0
	form component prediction	13	17	0	6	0
		14	4	1	2	1
		15	15	0	2	0
		16	8	0	3	0
		17	13	0	3	0
		18	6	0	2	0
		19	34	0	6	0
		20	15	0	3	0

Table 5.13: Per Benchmark Register Utilization (8x8 Array)

Benchmark Kernels			Total Number of Variables in All Local RFs		Peak Number of Variables in a Local RF	
			data	pred	data	pred
LDL	ldl	1	105	0	6	0
	fnorm	2	2	0	1	0
FFT	radix4	3	26	0	4	0
IDCT 8x8	vertical	4	80	0	6	0
	horizontal	5	58	10	4	2
MPEG-2 Decoder	dequantize intra	6	3	1	1	1
	dequantize non-intra	7	3	0	1	0
	saturate	8	22	12	2	3
	add block non-intra	9	12	4	2	2
	add block intra	10	10	3	2	2
	clear block	11	0	0	0	0
	fast idct	12	28	0	2	0
	form component prediction	13	13	0	2	0
		14	0	0	0	0
		15	10	0	2	0
		16	10	0	2	0
		17	8	0	2	0
		18	4	0	1	0
		19	24	0	3	0
		20	6	0	2	0

With the clear difference in usage between data and predicate register files, further area savings could possibly be obtained by having fewer predicate registers than data registers. On the other hand, since their area cost is relatively small, having larger predicate register files may be an inexpensive way to improve performance. Initial results have shown that varying the size of the predicate register files without varying the size of the data register files has a small impact on both area and performance.

We may select the peak number of registers used by the benchmarks, found in columns four and five of Table 5.12 and Table 5.13, as the number of registers to have in each local register file to ensure that the register file size does not restrict the achievable IPC. In the 4x4 array, there would be eight registers per data local register file and four registers per predicate local register file. In the 8x8 array, there would be six registers per data local register file and three registers per predicate local register file. Since this represents the maximum usage of the local register files, local register files in these architectures do not need to be any larger. However, this is usually unnecessary because DRESC can assign the variables to available local registers in other functional units or to available global registers without reducing the performance in terms of IPC.

We may also choose to have enough local registers across the reconfigurable array to store the total number of variables listed in columns two and three of results in Table 5.12 and Table 5.13. In the 4x4 array, there would be four registers per data local register file and one register per predicate local register file. In the 8x8 array, there would be two registers per data local register file and one register per predicate local register file. However, this does not guarantee that there are sufficient local registers to achieve the maximum IPC, as there may be particular local register files that need more registers than others.

It is possible to further reduce the size of the local register files. Having insufficient local register files will not directly cause DRESC to fail to schedule the loop for an architecture. If there is no place to store the variables, DRESC could find another schedule with a larger iteration interval (II) and a lower IPC. However, the maximum number of iteration intervals is restricted by the number of contexts that can be stored in the limited context memory.

We have shown that the maximum potential utilization of local register files is small. However, these results do not indicate the best choice for the local register file size, but the peak register file usage establishes an upper bound. The best choice must be found by experiments such as those performed in Section 5.2.3.

5.3 Summary

In this chapter, we experimentally measured the impact of architecture parameters related to the global and local register files on the area and performance of our coarse-grained reconfigurable array. We found that increasing the number of shared connections to the global register files improved performance significantly and even reduced the area. Adding more VLIW FUs boosts performance by up to 33%. As well, low utilization of the local register files explain why one or two registers in each local register file are sufficient to achieve good performance.

Our recommendations concerning the register file architecture are summarized in Table 5.14. While the optimal value of each parameter may be sensitive to interactions with other modifications to the baseline architecture, we do not conduct a new set of experiments to tune the parameters starting from these recommendations because of three reasons: 1) we are more interested in general trends rather than absolute parameter values; 2) absolute parameter values for this specific ADRES-based architecture may not be relevant to other coarse-grained reconfigurable architectures or different implementations of the ADRES architecture; and 3) the variation is expected to be small.

Table 5.14: Recommendations for Architectural Parameters

Parameter		Value in Baseline Architecture	Recommended Value
Global Register Files	Number of FUs connected to each register file	4 (4x4 array) 8 (8x8 array)	16 (4x4 array) 56 (8x8 array)
	Number of read and write ports per register file	4* (4x4 array) 8* (8x8 array)	4* (4x4 array) 8* (8x8 array)
	Number of FUs with ports to the register file and to main memory	4 (4x4 array) 8 (8x8 array)	4 (4x4 array) 8 (8x8 array)
	Number of registers per register file	16 (4x4 array) 64 (8x8 array)	14
Local Register Files	Number of FUs connected to the read and write ports on each register file	5 per read port 1 per write port	5/read port, 1/write port (4x4 array) 5/read port, 5/write port (8x8 array)
	Number of read ports per register file	1	1 (4x4 array) 2 (8x8 array)
	Number of registers per register file	4	2

* There are twice as many read ports on the Data Global Register File.

The results have shown that similar trends are noted with both the area estimated directly with our area model and the area determined after adjusting for the size of SRAM cells and read ports. This demonstrates the robustness of our findings. We also found that the adjusted area was significantly smaller but followed the same general pattern. This highlights the fact that memory is a key aspect of our area model that needs to be refined, while indicating that there is good fidelity between the two area models.

Chapter 6

ENHANCED ARCHITECTURES

In this chapter, we use the recommendations from Chapter 5 to propose two new architectures, one containing a 4x4 array and one containing an 8x8 array. These architectures are optimized to have a good performance-area tradeoff. We also want to construct high-quality architectures for comparison in future research.

We use the unadjusted area model in this chapter. Although it may be inaccurate in terms of SRAM cell size or read port sizes, it is based on numbers obtained from Synopsys, unlike the adjustments that were made to test for robustness. As well, the results from Chapter 5 show that the unadjusted area model gives more conservative results and has good fidelity with the adjusted area model.

6.1 New 4x4 Array Architecture

In this section, we propose a new 4x4 array architecture and compare it against the baseline architecture.

6.1.1 Selected Parameters

Our new 4x4 array architecture has the following parameters:

1. All sixteen functional units are connected to the global register files using shared read and write ports.
2. The predicate global register file has four read ports and four write ports; the data global register file has eight read ports and four write ports.
3. The global register files each have 14 registers.
4. The local register files each have one read port and one write port.
5. The local register file's read port is connected to five neighbouring functional units including its local functional unit.
6. The local register files each have two registers.

6.1.2 Results

Table 6.1 shows a comparison between the enhanced 4x4 array (new) architecture and the baseline (old) architecture broken down by benchmark loop. As the table shows, the enhanced architecture achieves -6% to 100% higher IPC for individual benchmarks and 32% higher IPC on average, over the set of benchmarks, than the baseline architecture. The enhanced architecture also requires -1% to 29% less area for individual benchmarks and 16% less area on average, over the set of benchmarks, than the baseline architecture. The IPC per mm^2 is -6% to 152% higher for individual benchmarks and 56% higher on average, over the set of benchmarks, in the enhanced architecture than in the baseline.

The average IPC per mm^2 for the enhanced architecture is higher than that of all other ADRES-based 4x4 array architectures reported in Chapter 5 for the unadjusted area model. The highest average IPC per mm^2 reported in Chapter 5 was 5.14 for an architecture in Section 5.1.1. In this architecture, all sixteen functional units were connected to the global register files using shared using shared read and write ports. This indicates that sharing the global register file ports with all sixteen functional units generated 49% of the average 56% improvement, while reducing the size of the global and local register files generated the remaining 7% improvement. Therefore, the degree of connectivity to the global register files is of critical importance to the register file architecture.

Table 6.1: Per Benchmark Comparison Between Results for Enhanced and Baseline Architectures with a 4x4 Array

Benchmark Kernels			Performance (IPC)		Number of Contexts		Area (mm^2)		IPC / mm^2	
			new	old	new	old	new	old	new	old
LDL	ldl	1	8.94	9.47	18	17	3.72	3.69	2.41	2.57
	fnorm	2	5.00	4.38	7	8	1.95	2.23	2.56	1.97
FFT	radix4	3	9.88	7.18	8	11	2.11	2.71	4.67	2.65
IDCT 8x8	vertical	4	10.33	9.30	9	10	2.27	2.55	4.54	3.65
	horizontal	5	11.08	11.08	13	13	2.92	3.04	3.80	3.65
MPEG-2 Decoder	dequantize intra	6	8.50	5.67	2	3	1.15	1.41	7.37	4.01
	dequantize non-intra	7	9.50	6.33	2	3	1.15	1.41	8.24	4.47
	saturate	8	11.14	8.67	7	9	1.95	2.39	5.70	3.63
	add block non-intra	9	11.00	7.33	4	6	1.47	1.90	7.46	3.85
	add block intra	10	10.00	8.00	4	5	1.47	1.74	6.78	4.60
	clear block	11	8.00	4.00	1	2	0.99	1.25	8.05	3.19
	fast idct	12	9.88	9.88	8	8	2.11	2.23	4.67	4.44
	form component prediction	13	10.25	6.83	4	6	1.47	1.90	6.95	3.59
		14	6.50	4.33	2	3	1.15	1.41	5.63	3.06
		15	9.50	8.14	6	7	1.79	2.06	5.30	3.94
		16	8.25	5.50	4	6	1.47	1.90	5.60	2.89
		17	10.80	9.00	5	6	1.63	1.90	6.61	4.73
		18	10.00	6.00	3	5	1.31	1.74	7.61	3.45
		19	11.17	7.44	6	9	1.79	2.39	6.23	3.11
		20	10.75	6.14	4	7	1.47	2.06	7.29	2.97
Arithmetic Mean			9.52	7.23	5.9	7.2	1.77	2.10	5.38	3.45

While improvements were reported for most benchmarks, benchmark one experienced a decrease in IPC and an increase in area, due to its need for one more context. This is probably caused by the reduction in the sizes of the global and local register files, which may make it more difficult for DRESC to find an efficient schedule.

6.2 New 8x8 Array Architecture

In this section, we propose a new architecture based on an 8x8 array of functional units.

6.2.1 Selected Parameters

This architecture has the following parameters:

1. Fifty-six of the 64 functional units are connected to the global register files using shared read and write ports.
2. The predicate global register file has eight read ports and eight write ports; the data global register file has sixteen read ports and eight write ports.
3. The global register files each have 14 registers.
4. The local register files each have two read ports and one write port.
5. The local register file's read ports and write port are connected to five neighbouring functional units including its local functional unit.
6. The local register files each have two registers.

6.2.2 Results

Table 6.2 shows a comparison between the enhanced 8x8 array (new) architecture and the baseline (old) architecture broken down by benchmark loop. As the table shows, the enhanced architecture achieves -29% to 200% higher IPC for individual benchmarks and 42% higher IPC on average, over the set of benchmarks, than the baseline architecture. The enhanced architecture also requires -7% to 38% less area for individual benchmarks and 23% less area on average, over the set of benchmarks, than the baseline architecture. The IPC per mm^2 is -33% to 383% higher for individual benchmarks and 88% higher on average, over the set of benchmarks, in the enhanced architecture than in the baseline.

Table 6.2: Per Benchmark Comparison Between Results for Enhanced and Baseline Architectures with an 8x8 Array

Benchmark Kernels			Performance (IPC)		Number of Contexts		Area (mm^2)		IPC / mm^2	
			new	old	new	old	new	old	new	old
LDL	ldl	1	20.12	20.12	8	8	8.37	9.14	2.40	2.20
	fnorm	2	17.50	8.75	2	4	4.37	6.60	4.00	1.33
FFT	radix4	3	19.75	13.17	4	6	5.71	7.87	3.46	1.67
IDCT 8x8	vertical	4	15.50	18.60	6	5	7.04	7.24	2.20	2.57
	horizontal	5	20.57	28.80	7	5	7.71	7.24	2.67	3.98
MPEG-2 Decoder	dequantize intra	6	17.00	8.50	1	2	3.71	5.33	4.58	1.59
	dequantize non-intra	7	9.50	9.50	2	2	4.37	5.33	2.17	1.78
	saturate	8	26.00	15.60	3	5	5.04	7.24	5.16	2.16
	add block non-intra	9	22.00	14.67	2	3	4.37	5.97	5.03	2.46
	add block intra	10	40.00	13.33	1	3	3.71	5.97	10.79	2.23
	clear block	11	8.00	8.00	1	1	3.71	4.70	2.16	1.70
	fast idct	12	19.75	19.75	4	4	5.71	6.60	3.46	2.99
	form component prediction	13	20.50	10.25	2	4	4.37	6.60	4.69	1.55
		14	13.00	6.50	1	2	3.71	5.33	3.51	1.22
		15	28.50	14.25	2	4	4.37	6.60	6.51	2.16
		16	16.5	11.00	2	3	4.37	5.97	3.77	1.84
		17	27.00	13.50	2	4	4.37	6.60	6.17	2.04
		18	15.00	10.00	2	3	4.37	5.97	3.43	1.68
		19	22.33	16.75	3	4	5.04	6.60	4.43	2.54
		20	14.33	10.75	3	4	5.04	6.60	2.84	1.63
Arithmetic Mean			19.64	13.59	2.9	3.8	4.97	6.48	3.95	2.10

The average IPC per mm^2 for the enhanced architecture is higher than that of all other ADRES-based 8x8 array architectures reported in Chapter 5 for the unadjusted area model. The highest average IPC per mm^2 reported in Chapter 5 was 3.84 for an architecture in Section 5.1.1. In this architecture, 56 of the 64 functional units were connected to the global register files using shared read and write ports. This indicates that sharing the global register file ports with 56 of the 64 functional units generated 83% of the average 88% improvement, while reducing the size of the global and local register files, adding a flexible connection pattern for the write ports of the local register file, and adding a second read port to each local register file combined to generate the remaining 5% improvement. Therefore, the degree of connectivity to the global register files is of critical importance to the register file architecture.

While improvements were reported for most benchmarks, benchmarks four and five experienced decreases in IPC and increases in area, due to their need for one or two more contexts. This is most likely caused by the reduction in the sizes of the global and local register files, which may make it more difficult for DRESC to find an efficient schedule.

6.3 Comparison Between the New 4x4 and 8x8 Array Architectures

From Table 6.1 and Table 6.2, we see that the 8x8 array has both higher performance and a larger chip area than the 4x4 array. Since the 8x8 array has four times as many functional units and register files as the 4x4 array, it might have been expected to have four times the performance and occupy an area that is four times as large. Instead, the performance and the area of the 8x8 array are roughly double and triple that of the 4x4 array, respectively. The performance is most probably limited by the parallelism available in the benchmark loops. It is possible that loop unrolling would expose more parallelism and lead to greater increases in performance. The area, on the other hand, shows an interesting side effect of increasing the IPC. Even though more area was required for additional functional units and other architectural elements that increase the IPC, this was partially offset by the corresponding decrease in the iteration interval (II), which is equal to the number of contexts. Because fewer contexts are required, less area is needed for configuration memory. The IPC per mm^2 was -8% to 152% higher, and 36% higher on average, in the 4x4 array than in the 8x8 array for the benchmark loops tested.

6.4 Summary

In this chapter, two enhanced architectures were presented, which combined the best aspects of the ADRES-based architectures reported in Chapter 5. The enhanced 4x4 array architecture has an IPC per unit area that is -6% to 152% higher than the baseline 4x4 array architecture and 56% higher on average. The enhanced 8x8 array has an IPC per unit area that is -33% to 383% higher than the baseline 8x8 array architecture and 88% higher on average. The degree of connectivity to the global register files was also shown to be the greatest contributor to the improvements and is therefore of critical importance to register file architectures. When they were compared against each other, the enhanced 8x8 array architecture had better performance, but the enhanced 4x4 array architecture demonstrated -8% to 152% higher IPC per unit area and 36% higher IPC per unit area on average.

Chapter 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we have investigated the impact of the global and local register file architecture on a reconfigurable system based on the ADRES architecture from the Interuniversity Microelectronics Centre (IMEC). The register files consume a significant amount of area on the reconfigurable device, and their architecture has a strong impact on the performance. Starting with a baseline architecture, we considered the global and local register files separately, and found that:

- The global register files should be tightly connected to as many functional units as possible. Not all functional units need dedicated read and write ports, however.
- A global register file depth of 14 provides the best performance per unit area, for the assumptions made for this architecture.
- The local register files should be connected to several neighbours, although the importance of this is small compared to the importance of providing wide access to the global register files.
- One read port and one write port are sufficient for each local register file, but it is also good to have two read ports.

- The local register files should be small; two registers per register file is sufficient.

Using these results, we developed two new register file architectures that have on average 56% higher performance per unit area for a 4x4 array and on average 88% higher performance per unit area for an 8x8 array compared to baseline 4x4 and 8x8 array architectures. On individual benchmarks, the new 4x4 array had -6% to 152% higher performance per unit area than the baseline 4x4 array and the new 8x8 array had -33% to 383% higher performance per unit area than the baseline 8x8 array. The sharing of the global register file ports with all or most of the functional units contributed most of the improvements to the instructions per cycle per unit area. Although the numerical results are specific to this processor, the trends that we observe may apply to many other reconfigurable systems that have global or local register files.

7.2 Future Work

This section describes some further work that could enhance the understanding of the structure and connectivity of register files in coarse-grained reconfigurable architectures.

First, the interaction between the register files and the main memory interface and the interconnections between functional units in the array can be investigated in more detail. The results in this thesis have shown that adding more functional units with both global register file ports and main memory ports can boost performance up to 33%, though at the cost of larger area.

Second, the "closest" and "full" interconnect architectures can be investigated to see how the optimal number of local register files is dependent on the interconnect architecture. As well, the register utilization for different interconnect architectures can be studied.

Third, the value of register files versus the value of other architectural elements can be compared. The number of functional units, the interconnect architecture, and the number of main memory ports have all been shown to significantly increase performance. In this thesis, we have tried to optimize the performance per area ratio. Having done that, we can change the problem to achieving a certain level of performance with the minimum area, or achieving the maximum performance with a specified area.

7.3 Contributions of This Work

The contributions of this thesis are as follows:

1. We presented a methodology for evaluating ADRES-based reconfigurable architectures. This included the use of the IMPACT-I [73] and DRESC [74] compilers used for ADRES [3]. As well, the SCRAP tool used in [79] was enhanced to generate an XML architecture file, modularized VHDL code that included global register files, and a parameterized area model that could produce area estimates that correlate well with Synopsys post-synthesis area estimates, but runs in less than a second.
2. We performed experiments on the degree of connectivity of global and local register files to the functional units and found that shared connectivity of every functional unit to the global register files is vital to obtain high performance.
3. We verified that increasing the number of read and write ports on global and local register files improves performance, but at the cost of significant area.
4. We performed sweeps of the size of global and local register files and discovered in particular that only very few registers are needed in each local register file. This was attributed to the low utilization of local register files by the benchmark kernels.

5. We constructed enhanced 4x4 array and 8x8 array architectures that applied this new knowledge about register files to achieve on average 56% and 88% higher instructions per cycle per unit area, respectively, compared to the baseline architectures. For individual benchmarks, the enhanced 4x4 array architecture increased the instructions per cycle per unit area by -6% to 152% and the enhanced 8x8 array architecture increased the instructions per cycle per unit area by -33% to 383%.

A portion of this work has been published in [16].

REFERENCES

[1] H. Singh, M-H Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Compute Intensive Applications", IEEE Transactions on Computers, Vol. 49, No. 5, May 2000, pp. 465-481.

[2] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", Computer, Vol. 33, No. 4, 2000, pp. 70-77.

[3] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix", International Conference on Field-Programmable Logic and Applications (FPL), Sept. 2003, pp. 61-70.

[4] J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk, "Fast Communication Mechanisms in Coarse-grained Dynamically Reconfigurable Array Architectures", Proceedings of the Second International. Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE´00, in conjunction with PDPTA 2000), Las Vegas, USA, June 2000.

[5] J. M. Rabaey, "Reconfigurable Processing: The Solution to Low-Power Programmable DSP", IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 1, Munich, Germany, April 1997, pp. 275-278.

[6] D. Chen and J. Rabaey, "Reconfigurable Multi-Processor IC for Rapid Prototyping of Algorithmic-Specific High-Speed Datapaths," IEEE Journal of Solid-State Circuits, Vol. 27, No. 12, Dec. 1992.

[7] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications", ACM/SIGDA International Symposium on FPGAs (FPGA), Feb. 1999, pp. 135-143.

[8] M. Frank, S. Amarasinghe, and A. Agarwal, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs", IEEE Micro, Vol. 22, No. 2, Mar/Apr 2002, pp. 25-35.

[9] N. Tabrizi, N. Bagherzadeh, A. H. Kamalizad, and H. Du, "MaRS: A Macro-pipelined Reconfigurable System", Proc. of the First Conference on Computing Frontiers, Ischia, Italy, April 2004, pp. 343-349.

[10] A. K. W. Yeung and J. M. Rabaey, "A Reconfigurable Data-Driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms", Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences, Vol. 1, January 1993, pp. 169-178.

[11] E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", IEEE Symposium on Field-Programmable Custom Computing Machines, April 1996.

[12] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "On Reconfigurable Co-Processing Units", Proceedings of the Reconfigurable Architectures Workshop (RAW98), Orlando, Florida, March 1998.

[13] F. Barat, M. Jayapala, T. Vander Aa, R. Lauwereins, G. Deconinck, and H. Corporaal, "Low Power Coarse-Grained Reconfigurable Instruction Set Processor", Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL), September 2003, pp. 230-239.

[14] P. M. Heysters, G. J. M. Smit, and E. Molenkamp, "A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems", The Journal of Supercomputing, Vol. 26, No. 3, Kluwer Academic Publishers, Boston, U.S.A., November 2003, ISSN 0920-8542.

[15] J. Becker, M. Glesner, A. Alsolaim, and J. Starzyk, "Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, April 2000, pp. 205-215.

[16] Z. Kwok and S. J. E. Wilton, "Register File Architecture Optimization in a Coarse-Grained Reconfigurable Architecture", to appear in the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, April 2005.

[17] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171-210.

[18] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", Proceedings of the Conference on Design, Automation and Test in Europe (DATE), Munich Germany, 2001, pp. 642-649.

[19] B. K. Lee and L. K. John, "Implications of Programmable General Purpose Processors for Compression/Encryption Applications", Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), July 2002, pp. 233-242.

[20] P. S. Zuchowski, C. B. Reynolds, R. J. Grupp, S. G. Davis, B. Cremen, and B. Troxel, "A Hybrid ASIC and FPGA Architecture", Proceedings of the International Conference on Computer-Aided Design (ICCAD), November 2002, pp. 187-194.

[21] Xilinx Inc., The Programmable Logic Data Book, 1994.

[22] AT & T Inc., ORCA Datasheet, 1994.

[23] S. K. Nag and R. A. Rutenbar, "Performance-driven Simultaneous Place and Route for Island-Style FPGAs", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, December 1995, pp. 332-338.

[24] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", International Workshop on Field Programmable Logic and Applications, 1997.

[25] J. S. Swartz, V. Betz, and J. Rose, "A Fast Routability-driven Router for FPGAs", Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, March 1998, pp. 140-149.

[26] A. Marquardt, V. Betz, and J. Rose, "Timing-driven Placement for FPGAs", Proceedings of the Eighth International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, California, USA, February 2000, pp. 203-213.

[27] S. J. E. Wilton, "A Crosstalk-aware Timing-driven Router for FPGAs", Proceedings of the ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, February 2001, pp. 21-28.

[28] G. Lemieux and D. Lewis, "Using Sparse Crossbars Within LUT Clusters", Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, February 2001, pp. 59-68.

[29] P. Kannan, S. Balachandran, and D. Bhatia, "fGREP - Fast Generic Routing Demand Estimation for Placed FPGA Circuits", Lecture Notes in Computer Science, Vol. 2147, Springer-Verlag, London, U.K., Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL), 2001, pp. 37-47.

[30] K. Poon, A. Yan, and S. J. E. Wilton, "A Flexible Power Model for FPGAs", Proceedings of the Reconfigurable Computing is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications (FPL), Montpellier, France, September 2002, pp. 312-321, Lecture Notes in Computer Science, Vol. 2438, 2002, Springer-Verlag, pp. 48-58.

[31] J. Y. Lin, A. Jagannathan, and J. Cong, "Placement-driven Technology Mapping for LUT-based FPGAs", Proceedings of the ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, February 2003, pp. 121-126.

[32] P. Maidee, C. Ababei, and K. Bazargan, "Fast Timing-driven Bartitioning-based Placement for Island Style FPGAs", Proceedings of the 40th Conference on Design Automation (DAC), Anaheim, CA, USA, June 2003, pp. 598-603.

[33] D. Chen, J. Cong, and Y. Fan, "Low-Power High-Level Synthesis for FPGA Architectures", Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), Seoul, Korea, August 2003, pp. 134-139.

[34] J. Lamoureux and S. J. E. Wilton, "On the Interaction Between Power-Aware FPGA CAD Algorithms", International Conference on Computer Aided Design (ICCAD), November 2003, pp. 701-708.

[35] F. Li, Y. Lin, L. He, and J. Cong, "Low-power FPGA Using Pre-defined Dual-Vdd/Dual-Vt Fabrics", Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, Feb. 2004, pp. 42-50.

[36] D. Lewis, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, E. Ahmed, K. Stevens, R. Yuan, R. Cliff, J. Rose, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, and C. Lane, "The Stratix II Logic and Routing Architecture", Proceedings of the ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, CA, USA, Feb. 2004, pp. 14-20.

[37] Xilinx Inc., Virtex-4 Brochure, http://www.xilinx.com, 2004.

[38] Leopard Logic Inc., "HyperBlox MP Embedded Structured ASIC Cores", Product Brief, http://www.leopardlogic.com, 2004.

[39] L. Pileggi, A. Strojwas, P. Gopalakrishnand, V. Kheterpal, A. Koorapaty, C. Patel, V. Rovner, and K. Tong, "Exploring Regular Fabrics to Optimize the Performance-Cost Trade-off", Proceedings of the 40th ACM/IEEE Design Automation Conference (DAC), 2003, pp. 782-787.

[40] Altera Corp., HardCopy Series Handbook, Vol. 1, Section II. HardCopy Stratix Device Family Data Sheet, Chapter 6. Description, Architecture & Features, Version 2.0, http://www.altera.com, January 2005.

[41] Altera Corp., HardCopy Series Handbook, Vol. 1, Section I. HardCopy II Device Family Data Sheet, Chapter 2. Description, Architecture & Features, Version 1.0, http://www.altera.com, January 2005.

[42] F. Lien, J. Feng, E. Huang, C. Sun, T. Liu, N. Liao, and D. Hightower, "A Hardware / Software Solution for Embeddable FPGA", Proceedings of the Custom Integrated Circuits Conference (CICC), May 2001, pp. 71-74.

[43] T. Vaida, "Reprogrammable Processing Capabilities of Embedded FPGA Blocks", Proceedings of the IEEE International ASIC/SOC Conference, September 2001, pp. 180-184.

[44] IBM Microelectronics, Blue Logic Cu-08 ASIC Product Brief, April 2002.

[45] eASIC Corp., "eASICore Overview (0.13μm Process)", http://www.easic.com, 2005.

[46] M2000, Inc., "M2000 FlexEOS Embedded FPGA Macros", Product Brief, http://www.m2000.fr/products.htm, 2004.

[47] Leopard Logic Inc., "HyperBlox FP Embedded FPGA Cores", Product Brief, http://www.leopardlogic.com, 2004.

[48] N. Kafafi, K. Bozman, and S. J. E. Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", Proceedings of the ACM

International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, CA, February 2003, pp. 1-9.

[49] A. Yan and S. J. E. Wilton, "Product Term Embedded Synthesizable Logic Cores", Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT), Tokyo, Japan, December 2003, pp. 162-169.

[50] S. J. E. Wilton and R. Saleh, "Programmable Logic IP Cores in SoC Design: Opportunities and Challenges", Proceedings of the Custom Integrated Circuits Conference (CICC), May 2001, pp. 63-66.

[51] R. Razdan and M. Smith, "A High Performance Microarchitecture with Hardware Programmable Functional Units, In MICRO27, November 1994, pp. 172-180.

[52] M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1995, pp. 99-107.

[53] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transactions on VLSI Systems, Vol. 4, No. 1, 1996, pp. 56-69.

[54] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1996, pp. 126-135.

[55] C. Iseli and E. Sanchez, "Beyond Superscalar Using FPGAs", Proceedings of the IEEE International Conference on Computer Design: VLIW in Computers and Processors, Cambridge, MA, USA, October 1993, pp. 486-490.

[56] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), April 1997, pp. 87–96.

[57] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), April 1998, pp. 28-37.

[58] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array", Computer, Vol. 24, No. 1, January 1991, pp. 81-89.

[59] P. Athanas and H. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis", Computer, Vol. 26, No. 3, March 1993, pp. 11-18.

[60] P. Athanas and A. Abbot, "Real-Time Image Processing on a Custom Computing Platform", Computer, Vol. 28, No. 2, February 1995.

[61] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Processor", Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 1997.

[62] O. Colavin and D. Rizzo, "A Scalable Wide-Issue Clustered VLIW with a Reconfigurable Interconnect", Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, San Jose CA, USA, 2003, pp. 148-158.

[63] J. R. Anderson, S. Sheth, and K. Roy, "A Coarse-Grained FPGA Architecture for High-Performance FIR Filtering", Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, USA, 1998, pp. 234-244.

[64] S.-S. Chin, W. Wu, and S. Hong, "Rapidly Reconfigurable Coarse-Grained FPGA Architecture for Digital Filtering Applications", The 45th Midwest Symposium on Circuits and Systems, Vol. 3, August 2002, pp. 203-206.

[65] W. Wu, S.-S. Chin, and S. Hong, "A Coarse-Grained FPGA Architecture for Reconfigurable Baseband Modulator/Demodulator", Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems and Computers, Vol. 2, November 2002.

[66] M. Sadasivam and S. Hong, "Application Specific Coarse-Grained FPGA for Processing Element in Real Time Parallel Particle Filters", Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, June 2003, pp. 116-119.

[67] N. Bansal, S. Gupta, N. Dutt, and A. Nicolau, "Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with Different Processing Element Configurations", WASP03.

[68] G. M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," Proceedings of AFIPS Conference, 1967, pp. 483-485.

[69] Elixent Ltd., "D-Fabrix Array", Datasheet, http://www.elixent.com, 2005.

[70] C. Ebeling, D. Conquist, and P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath", International Workshop on Field-Programmable Logic and Applications, August 1996.

[71] K. Bondalapati, Chameleon Systems Inc., "Parallelizing DSP Nested Loops on Reconfigurable Architectures using Data Context Switching", Proceedings of the Design Automation Conference (DAC), Las Vegas, Nevada, USA, June 2001, pp. 273-276.

[72] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, "Architecture and Synthesis for On-Chip Multi-Cycle Communication", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 4, April 2004, pp. 550-564.

[73] P. Chang, S. Mahike, W. Chen, N. Warter, and W. Hwu, "IMPACT: An Architectural Framework For Multiple-Instruction-Issue Processors", International Symposium on Computer Architecture, May 1991, pp, 266-275.

[74] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting Loop-Level Parallelism On Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling", IEE Proceedings – Computers and Digital Techniques, Vol. 150, No. 5, September 2003, pp. 255-261.

[75] P. Ienne, L. Pozzi, and M. Vuletic, "On the Limits of Processor Specialisation By Mapping Dataflow Sections on Ad-Hoc Functional Units", Technical Report CS 01/376, Swiss Federal Institute of Technology Lausanne (EPFL), December 2001, Updated June 2002.

[76] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 2001.

[77] K. Compton and S. Hauck, "Track Placement: Orchestrating Routing Structures to Maximize Routability", UWEE Technical Report #UWEETR-2002-0013, University of Washington, October 2002.

[78] N. Bansal, S. Gupta, N. Dutt, A. Nicolau, and R. Gupta, "Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures", Proceedings of the Design, Automation and Test In Europe Conference and Exhibition (DATE), Vol. 1, February 2004, pp. 474-479.

[79] S. J. E. Wilton, N. Kafafi, B. Mei, and S. Vernalde, "Interconnect Architectures for Modulo-Scheduled Coarse-Grained Reconfigurable Arrays", Proceedings of the International Conference on Field-Programmable Technology (FPT), Brisbane, Australia, December 2004, pp. 33-40.

[80] J. Lee, K. Choi, and N. D. Dutt, "Evaluating Memory Architectures For Media Applications On Coarse-Grained Reconfigurable Architectures", Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2003 (ASAP) pp.172-182, June 2003.

[81] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," Science, May 1983, pp. 671-680.

[82] C. Ebeling, L. McMurchie, S. Hauck, and S. Burns, "Placement and Routing Tools for the Triptych FPGA," IEEE Transactions on VLSI, Vol. 3, Dec. 1995, pp. 473-482.

[83] Y.-T. Cheng, Texas Instruments, "TMS320C6000 Integer Division", Application Report SPRA707, http://www.ti.com, October 2000.

[84] Agere Systems, "Division Math Function on the DSP16000 Core Digital Signal Processor", Application Note, http://www.agere.com, March 2002.

[85] Freescale Semiconductor, Inc., "Fractional and Integer Arithmetic Using the DSP56000 Family of General-Purpose Digital Signal Processors", Application Note APR3, Revision 1, http://www.freescale.com, January 2000.

[86] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A Fully Bypassed Six-Issue Integer Datapath and Register File on the Itanium-2 Microprocessor", IEEE Journal of Solid-State Circuits, Vol. 37, No. 11, Nov. 2002, pp. 1433-1440.

[87] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration", IEEE Transactions on Very Large Scale Integration Systems (VLSI), Vol. 6, No. 2, June 1998, pp. 247-256.

[88] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool For Evaluating and Synthesizing Multimedia and Communications Systems", Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, Research Triangle Park, NC, USA, December 1997, pp. 330-335.

[89] V. Betz, J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, Boston, 1999, p. 136.