UNIVERSITY OF CALIFORNIA, SAN DIEGO

Strategies for sharing a Floating Point Unit between SPEs

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Jose E. Lugo Martinez

2010

DEDICATION

This thesis is dedicated to my mother, who supported me unconditionally throughout my academic career. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to thank my mom, my sister and my brother for always supporting my endeavors. I would like to thank my advisors, Steven Swanson and Michael B. Taylor, for their support, guidance, patience, and encouragement throughout my graduate years. It has been a great honor and privilege to work with them.

I would also like to thank the Arsenal students research group, Jack Sampson, Nathan Goulding, Vladyslav Bryksin, Sat Garcia, and Ganesh Venkatesh for all their support in the development of Arsenal's Floating Point infrastructure; their work on Arsenal is the foundation of this thesis. In particular, I thank Jack for his support and guidance in developing and debugging Arsenal's FP infrastructure, and Nathan for his invaluable and uncountable help since the beginning, and without him, I cannot imagine what I would do.

I further would like to thank Professor Chung-Kuan Cheng, my thesis committee member, for his precious time and guidance.

Last but not least, I would like to thank my UCSD family, Sebastián Becerra, Ricardo Fagoaga, Rubén Lorenzo, Josué Pérez, Laura Pina, and Bárbara Zepeda for their friendship and infinite support.

ABSTRACT OF THE THESIS

Strategies for sharing a Floating Point Unit between SPEs

by

Jose E. Lugo Martinez

Master of Science in Computer Science

University of California, San Diego, 2010

Professor Steven Swanson, Chair

Designing special purpose processors and ASICs to execute computer programs requires a methodology that varies greatly from traditional general purpose software programming. The benefits of specialized processor designs and ASICs are: lower power consumption, and greater efficiency, as opposed to general purpose processors. Those benefits are the driven motivation in the Arsenal design that aims to incorporate 10s to 1000s of specialized processing elements (SPEs) into one system. Each one of the SPEs performs a well defined functionality that represents the variety of hardware designs, from general purpose processors to special purpose processors and ASICs.

Among those specialized hardware units, the Floating Point hardware infrastructure (FP) presents an important and interesting challenge reflected by its significant area and power requirements on the system. To one end, it makes the idea of having one FP unit per SPE prohibitively expensive. On the other hand, reducing the number of FP units could potentially create a bottleneck, and hence a negative impact on performance. Therefore, there is a significant tradeoff between area, power, energy and performance aspects for sharing the FP hardware among Arsenal's SPEs.

This thesis focus is designing and analyzing different strategies for sharing FP hardware for the SPEs across Arsenal. Therefore, the main goal is to find a proper balance between area, energy and performance for a set of FP sharing strategies over a sample set of FP applications. Our results show that a shared FP hardware per SPEs complex reduces area, energy and energy delay with negligible performance degradation amongst all designs.

Chapter 1

Introduction

Recently, there has been a wide transition to multicore designs motivated by power and microarchitectural scalability concerns. However, even though multicore designs can deal with such power and scalability issues in the short term, the problem is that utilization of the whole chip at full frequency (i.e. *utilization wall*) will resurfaced in a few generations of process scaling. This is due to a decrease in the percentage of transistors that can be switch at full frequency in a chip cause by threshold voltage scaling. Therefore, a different approach that can overcome aforementioned issues while still targeting general purpose computing is to be able to dictate the parts of the chip that are active at one time.

The Arsenal design is an example of such technique, and it is accomplished by combining an array of massively heterogeneous processors into one system. The Arsenal design aims to incorporate 10s to 1000s of specialized processing elements (SPEs) into one system. Each one of these SPEs performs a well defined functionality that represents the variety of hardware designs, from general purpose processors to special purpose processors and ASICs. This diversification is the key to achieving significant reductions in power consumption, increase performance, and increase efficiency because it allows targeting only specific applications and use a specific portion of the chip at once. The power savings of executing on a SPE rely on its ability to eliminate the overhead that microprocessor interpretation imposes on computation (e.g. register file access, operand bypassing, etc) [1]. Also, SPEs can use significantly less clock power by grouping multiple dependent operations into larger clock cycles and running at a lower clock speed. The architecture of the Arsenal system is described in more detail in Chapter 2.

Overall, the main goal of Arsenal is to provide a diverse set of wide range SPEs and computing resources whose heterogeneity can be map to the wide range of applications that may be run on the system, while still increasing performance, efficiency and energy savings. However, the challenge resides on the complexity of Arsenal systems which success is bounded by finding a good alignment between the diversity of SPEs and computing resources, and the diversity present in the underlying applications.

Therefore, to reduce complexity and design time, Arsenal systems will need to aggressively reuse hardware components and computing resources throughout the system. In particular, among those computing resources, the Floating Point (FP) hardware infrastructure presents an interesting and important characteristic magnified by its significant area and power requirements on the system.

This thesis explores and analyzes different strategies for sharing a FP accelerator infrastructure for the SPEs complexes in Arsenal. While the FP implementation complexity varies across different sharing strategies, the more relevant characteristic is that there is a tradeoff between area, energy and performance aspects for sharing (or reusing) FP hardware between SPEs in Arsenal. To one end, it makes the idea of having one FP unit per SPE prohibitively expensive. On the other hand, reducing the number of FP units could potentially create a bottleneck, and significantly degrade performance. Therefore, the contribution of this thesis is to provide a useful and comprehensive study for finding the best balance among the tradeoff between area, energy and performance aspects for the SPEs in the Arsenal design. In particular, we measure the area, energy and performance tradeoffs among seven different FP infrastructure sharing strategies, describe in more detail in Chapter 4. These FP sharing strategies are implemented as part of the Arsenal toolchain, and we compare the results to a baseline sharing configuration, presented in Chapter 6.

In order to present a meaningful analysis of our designs that reflect the broad range of floating point applications that may be run in Arsenal, we choose a wide sample set of floating point benchmarks which ranges from MP3 encoders, scientific applications to other applications that are amenable to hardware instantiations and spent a large part of execution inside loop bodies. All these applications are either well understood or are drawn from benchmark suites that include Lame MP3 Encoder [12], EEMBC's Basic Floating Point Automotive [4], and SPEC2000's integer and floating point applications [22] such as: equake, mesa and vpr.

To quantify the tradeoff we measure the changes in area, energy, efficiency and performance, according to each FP infrastructure sharing design. Therefore, the focus of this thesis is designing, implementing and analyzing different strategies for sharing a FP accelerator infrastructure for the SPEs across Arsenal. Based on the results, we found that a shared FP hardware per SPEs complex in Arsenal reduces area requirements between 3.6% and 435%, energy consumption between 1% and 102% and energy delay product by up to 35% while decreasing performance between 0.5% and 14% among all tested sharing designs.

In the remainder of this thesis, in Chapter 2, we provide more background on the architecture and motivation of the Arsenal system. In Chapter 3, we describe related work that have influenced our study of sharing hardware resources. In Chapter 4, we present a description of the FP infrastructure, as well as, details on each FP sharing configuration design and implementation. In Chapter 5, we describe the simulation mehodology and set of benchmarks used to gather the results. We then in Chapter 6 describe the results of this thesis. Finally, in Chapter 7 we conclude by reflecting on the area, energy, efficiency and performance tradeoff across the different FP sharing designs.

Chapter 2

Background

Threshold voltage scaling problems cause a decrease in the percentage of transistors that can be switch at full frequency on a chip. This motivates computer architecture researchers to find alternate ways on which to design processors. One approach is to design processors so that we can dynamically control the parts of the chip that are active at once, while still maintaining general purpose computing.

The Arsenal design is an example of such technique, and it is accomplished by combining an array of massively heterogeneous processors into one system. The Arsenal system is comprised of 10s to 1000s of specialized processing elements (SPEs) that are organized into complexes compose of a variable number of SPEs of different sizes and specialization. These SPEs represent a variety of hardware designs, from general purpose processors to specialized processors and ASICs that handle a well defined functionality. This specialization is key to achieving significant reductions in power consumption, increase performance, and increase efficiency because it allows targeting only specific applications and use a specific portion of the chip at once.

2.1 Arsenal system overview

Figure 2.1 illustrates an Arsenal processor composed of twelve SPE complexes that are connected to four banks of shared L2 cache through a grid based onchip interconnect. This collection of complexes, interconnect and caches is similar to previ-

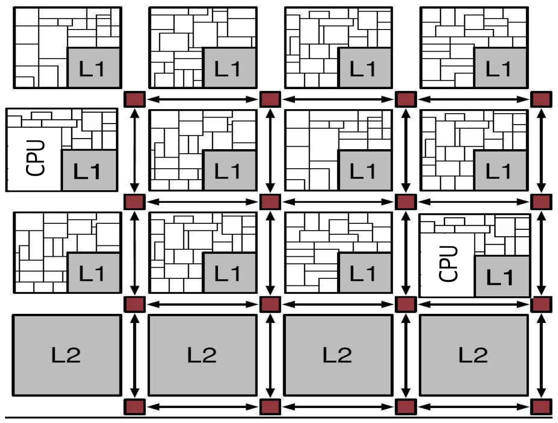


Figure 2.1: **An Arsenal Processor** It shows a collection of complexes comprising a SPE enabled system. Complexes communicate through an intracomplex interconnect connected to a memory system. Different complexes may contain different SPEs.

ously proposed tiled processors like Raw [27, 28], WaveScalar [24, 25], or TRIPS [19]. However, the fundamental difference is that instead of uniform complexes, each complex in an Arsenal processor is comprised of a variant number of SPEs that have different sizes and functionalities, where the size is defined by the architectural properties of the SPE and the available area budget in the complex, while the SPEs grouping is defined by the affinity in specialization, since SPEs that have related functionality are most likely to be used together, which enables them to share the L1 cache.

An Arsenal processor combines one or more general purpose processors, CPU, with 10s to 1000s SPEs that execute specific computations very efficiently. The CPU executes the runtime system and serves as the default when portions of the application cannot be mapped to any of the SPEs. The communication between the CPU and SPEs occurs through a simple interconnect that gives the CPU complete access to all of the SPEs internal state via a set of scan chains. These scan chains are used by the CPU to set up the SPE and transfer initial arguments. However, most of the data is passed via a coherent L1 cache. Figure 2.2 illustrates a zoom in of a typical Arsenal SPEs complex, highlighting the CPU/SPE scan chain interface.

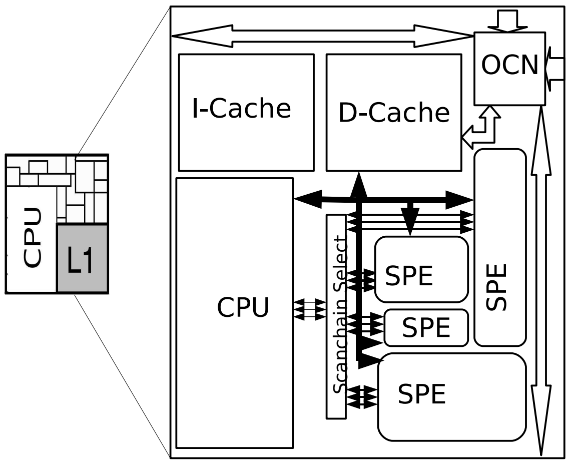


Figure 2.2: **A typical Arsenal SPEs complex** It shows a prototypical complex in an Arsenal system. Intracomplex SPEs communicate over both a simple scanchain based interface and through a coherent memory system.

The memory hierarchy is comprised of L1 caches allocated within each complex, L2 caches allocated to a collection of complexes, main memory, and optional L0 caches that are local to each SPE. Coherency is achieved through a MESI based protocol that ensures coherence between all L1 caches, L2 and main memory. The communication between processors through shared memory follows a release consistency model.

The intracomplex interconnect connects individual SPEs to the intercomplex grid with the property that exactly one SPE within a complex can be active at once, which saves a significant portion of area and power. However, this property is only enforced for intracomplex scheduling, thus, SPEs from different complexes can execute simultaneously, managed by the operating system.

2.2 Arsenal's SPE architecture

SPE selection The preferred code regions to be design into SPEs are code portions that execute frequently and are relatively stable. The former enables significant energy savings, while the latter ensures that subsequent modifications to the source code are infrequent and/or minor. In this thesis, we selected and designed SPEs for a diverse set of floating point workloads that ranges from MP3 encoders, scientific applications to other applications that satisfy aforementioned constraints. More details are provided in Section 5.2.

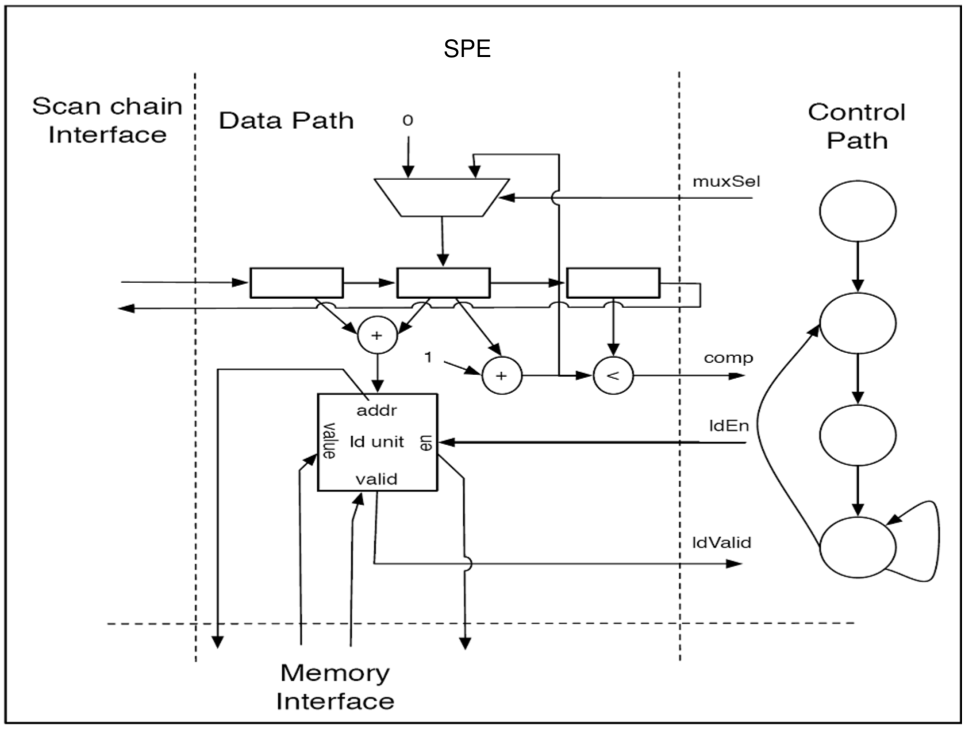


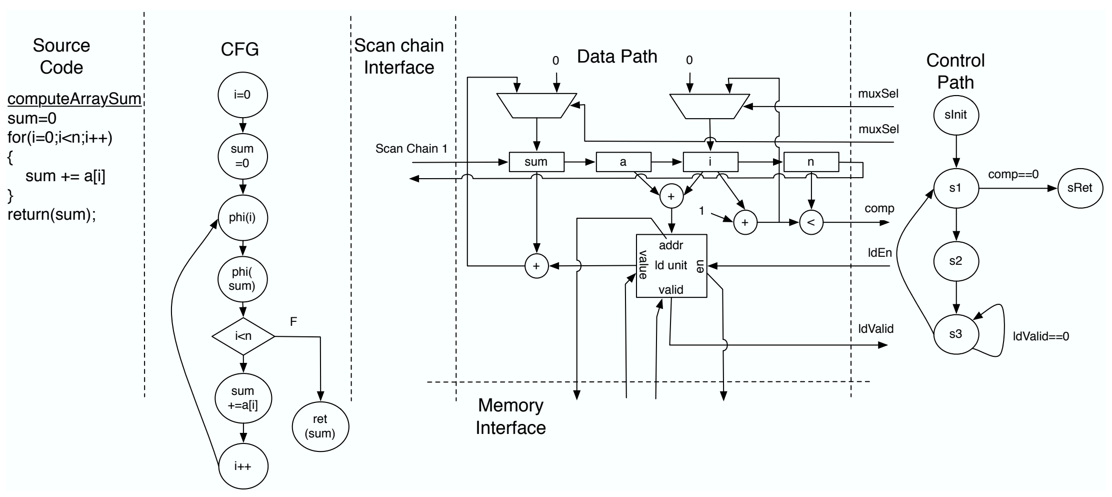
Figure 2.3: **An Arsenal SPE** It illustrates a single SPE organization.

SPE architecture and organization Figure 2.3 shows the architecture of a typical Arsenal SPE. The SPE is comprised of the datapath, the control unit (or control path), the memory interface, and the scan chain interface. The datapath resembles the dataflow graph of the computation and contains the functional units (e.g. adders, shifters, etc), the muxes to implement control decisions, and the registers to hold program values across clock cycles. Every functional unit is always ”on”, and specific components must be enabled and/or disabled with special enable and select lines. These enable and select lines are set by the control path that dictates which parts of the circuit are active at any given time. In particular, the control path implements a finite state machine (FSM) that mimics the control flow graph (CFG) of the underlying code. It tracks branch outcomes that come from the datapath to determine which state to enter on each cycle. The values for the enable and select lines are chosen based on the current state of the FSM and set on the registers and muxes such that the correct basic block is active each cycle.

In Arsenal multicycle instructions (e.g. memory ops, floating point ops, etc) are handled by adding a selfloop to the basic block that contains each operation and exporting a *valid* line to the control path. When a multicycle instruction is completed, it asserts the *valid* signal and control exits the loop and proceeds with the next basic block. The use of the *valid* signal mirrors the memory ordering token used in systems such as Tartan [14] and WaveScalar [25].

Memory ordering constraints are enforced by only permitting exactly one memory operation per basic block, thus, the SPE only enables one basic block at a time, guaranteeing that memory operations execute in the correct order. The load/store units connect to a coherent data cache that ensures that all loads and stores are visible to the rest of the system. Fortunately, the memory ordering enforced through the control path's FSM maps the memory order provided by program counters in general purpose processors. Therefore, to maintain this mapping and to decrease the amount of registers required in the datapath, the registers in the SPE datapaths are in SSA form, (i.e. each static SSA variable has a corresponding register and the registers value only changes when the corresponding static variable comes into scope). This invariant enforces that exactly one register value changes per new value that the program generates, hence, minimizing the number of register updates.

Figure 2.4 shows an example of the translation from C code to the SPE's datapath and control path. The phi operators for the variables i and sum in the CFG are turned into muxes. The datapath has a load unit to access the memory hierarchy for reading array a. The SPE's datapath and control path maps closely to the CFG of the



```
computeArraySum
sum=0
for(i=0;i<n;i++)
{
   sum += a[i]
}
return(sum);
```

Figure 2.4: **C to SPE Translation Example** An illustrative example that shows the translation from C code to hardware. The hardware schematic and state machine corresponds very closely to the control/data flow graph of the C code.

source code. Even further, the FSM of the SPE is almost identical to the CFG with the distinction of the multicycle operations selfloop.

SPEs execution model The execution model for the SPEs in an Arsenal processors is driven by the runtime which maps the application's code regions to be run on the available SPEs, and schedules them dynamically by evaluating the physical location and contention for a given SPE, and runtime behavior. In Arsenal a control register determines which SPE the CPU communicates with. Arsenal SPEs include 43 scan chains broken into two groups: 11 control scan chains and 32 datapath scan chains. The CPU uses control scan chains to pass arguments to the SPE. The scan chains for arguments are only 64 bits to make invocation fast.

A SPE initiates execution when the system loads an application that contains a code region that maps to a particular SPE in the system, then it replaces the CPU implementations of the code region with a stub that invokes the SPE. Subsequently, when the application calls the code region, the stub checks for an available SPE. If it finds one, it uses the scan chain interface to pass the arguments to the SPE. At this point, the SPE starts running and when the executing SPE completes, it raises an exception and transfers control back to the stub, which extracts the return value and passes it back to the caller. In the extreme case that no SPE is available, the CPU can run the application's code region such that it does not blocks.

In general, an application executing on Arsenal migrates between SPEs as its behavior changes [20]. Even further, the mapping of applications to SPEs can change at runtime to account for detected changes in the underlying application behavior.

2.3 Motivation

One of the biggest challenges in Arsenal is that the design space of an Arsenal processor is larger than conventional multiprocessors. Therefore, in order to explore the design space in a more efficient way, our research group ongoing focus is the development of a toolchain for SPE generation, where special purpose processors are synthesized automatically from high level constructs such as C, while SPEs for general purpose processors and established designs can be synthesized from already known solutions. This automatic synthesis approach is based on transformation of the applications into program dependence graph form and merging isomorphic components of the graph. Hence, the common applications are analyzed and semantically similar regions are merged to produce more robust SPEs given the area and power budget, while still reducing the overall amount of hardware for the Arsenal processor. The overall methodology is described in more detail in Section 5.1.

To reduce even further the complexity and design time, Arsenal systems will need to aggressively share and reuse hardware components throughout the system, especially at the SPEs complex level. This is the motivation of this thesis, which focuses on strategies for sharing a very specific hardware component, the Floating Point infrastructure. We believe that our work will provide a useful and comprehensive study for finding the best balance in the tradeoff between area, energy, and performance aspects for the SPEs in the Arsenal design.

Chapter 3

Related Work

Sharing (or reusing) hardware resources is studied in various works, however, the emphasis of such works ranges from sharing I caches and D caches, or sparsely and rarely used functional units to sharing the actual FP operators hardware logic among other FP or integer operators (e.g. sharing the FP adder and multiplier with the FP divider and/or square root, etc). In C.A.S.H. [3] the authors propose a way to selectively share some of the hardware resources on a single chip parallel processor in two ways: *a la SMT* when the sharing can be made noncritical for the implementation, or *a la CMP* whenever resource sharing leads to a superlinear increase of the implementation hardware complexity. In particular, they believe that branch predictors, I caches, D caches and long latency functional units (e.g. integer multipliers and dividers) can be shared among several ”processor” cores while keeping other major parts of the execution cores separated. Their work shows that there exists an intermediate design point between CMP and SMT by presenting a hybrid sharing design. In contrast, our motivation is focused on exploring multiple designs and implementations for sharing the FP hardware among SPEs in Arsenal.

In [21], Soderquist and Leeser provide an analysis of the area and performance tradeoff associated with different implementations of FP divide and square root that ranges from sharing the FP multiplier and adder logic to independent implementations. The main contribution of their work is a comprehensive analysis to help FP unit designers identify the FP divide and square root implementations that most effectively meet their area constraints and performance goals. Even further, instead of recommending novel methods, they focus on exploring the tradeoff inherent to established techniques used in commercial processors. Our work differs significantly from their approach since we are concerned with area, energy and performance tradeoff with sharing the FP hardware between the SPEs on an Arsenal chip and we do not explore sharing the FP logic neither inside the FP unit nor with any integer operator.

The authors of [9] present the design and implementation of a modular and portable FP unit, especially suitable for use in small ASIPs implemented in Architecture Description Language (ADL) [7]. Also, in their paper, they present a summary of area cost, and energy and speed estimates for different FP configurations. While they do explore area, energy and performance tradeoff their FP unit is quite limited (e.g. no hardware support for FP division, etc). Additionally, they target embedded systems while our work is oriented towards the general purpose computing capabilities of the Arsenal system.

Chapter 4

Floating Point Infrastructure

As previously stated, our overall approach consists of exploring and analyzing different sharing strategies for the FP infrastructure in Arsenal by measuring area, energy and performance, and finding the design that best balances the tradeoff among them. In this section, we describe, in detail, each step in the FP infrastructure development, from the actual FP unit logic to the distinct FP sharing configurations designed.

4.1 Floating Point Unit

4.1.1 Overview

One of Arsenal's goal is to provide support for general purpose computing, which frequently includes the need to represent very large and very small values as those observed in scientific applications. This is hard to accomplish using fixed point formats, because maintaining the desired levels of precision and range grows the bitwidth larger. Therefore, in those cases FP formats are used to represent real numbers [5]. The most widely known and used format for FP arithmetic is the IEEE 754 Standard for Binary Floating point Arithmetic [8].

This standard specifies two precision bitwidths formats: *single precision* (32 bit) and *double precision* (64 bit), to which floating point numbers are to be represented, as well as how basic arithmetic operations should be performed on them. Even further,

Table 4.1: IEEE single precision representation.

s	e	.	.	.	e	f	.	.	.	f
0	1	.	.	.	8	9	.	.	.	31

each format has representations for NaNs (Not a Number) and $\pm\infty$ (Infinity).

For simplicity, in this thesis we only focus on the IEEE 754 single precision FP standard. This standard requires a 32 bit word, usually represented as numbered from 0 to 31, left to right. The first bit represents the sign, *s*, the next 8 bits are the exponent, *e*, and the last 23 bits are the fraction, *f*, as shown in Table 4.1. Therefore, it is possible to represent any FP number using the following equation:

$$(-1)^s \times 1.f \times 2^{e-127} \tag{4.1}$$

4.1.2 Design and Implementation

In this section we describe the design and implementation of the IEEE 754 single precision FP format and the corresponding modules to support the basic arithmetic operators.

A system can emulate FP logic with a software implementation, if it is desired, however, FP arithmetic is highly complex which makes this approach undesirable. To complicate things even further if the FP logic implementation does not complies with the IEEE 754 standard, then the behavior of the FP environment becomes significantly unstable. In contrast, another solution is to implement the FP logic in hardware, namely the floating point unit (FPU). This approach greatly accelerates the FP arithmetic and calculations, mainly because a hardware implementation requires a small fraction of the time that a software implementation would require (usually between 50 and 300 times slower than the hardware). Even further, implementing the FP logic in hardware enables sharing this resource across the system at the expense of the initial design and implementation difficulty cost.

Arsenal's FPU design is based on the FP logic designed and implemented in Raw [27]. Even though, the original idea was to have a FPU that fully supported

IEEE 754 standard some goals were omitted. In particular, we only support single precision FP operations and we relaxed the compliance of the IEEE 754 standard by not implementing gradual underflow.

The FPU functionality is divided into 7 modules. In Table 4.2 we show the names, functionality and latencies for all the implemented arithmetic operators modules that include *absolute value* and *negation*, *addition* and *subtraction*, *comparisons*, *division*, *format conversions* (between FP and fixed point representations and vice versa), and *multiplication*. Each module is implemented in Verilog, a hardware description language [18]. Together these modules compose the internals of the FPU top level module, which encapsulates all behavior in one location and provides one central interface for calculation of floating point operations. Additionally, after each arithmetic operation the resulting FP value, is renormalized and rounded in the FPU's top level by conditionally adding 1 unit in the last place, as required by IEEE 754.

The FPU top level wraps around all the FP operators, where all basic FP arithmetic operators are fully pipelined, except for the FP divide. The FPU interface is very similar to a memory *LOAD* instruction, where the FPU receives the FP operation request and data in a pipeline fashion, process it, and has it stream out, along with a ”valid” signal. All subsequent FP operations request and data must wait until the active request is finalized. The ready and valid signals capabilities in each module, enables them to be easily assembled into larger or smaller designs. IEEE's specification for handling and raising flags (*inexact, underflow, overflow, division by zero and invalid*) is supported, and any detected flag is propagated to the end of the pipeline and merge into the stream out result value. An additional *unimplemented* flag is added to IEEE's exceptions flags for signaling when the compliance with the IEEE 754 standard is not met.

In detail, at the hardware level, the FPU's input data consist of one or two FP values and an operator selector opcode to determine the calculation to be performed by the FPU. The output consists of the result value of the specified calculation plus the corresponding exception flags (set to 0 if no exceptions are raised).

The next paragraphs describe the particular details for each FP arithmetic operator module.

Table 4.2: Floating Point Hardware Modules description.

Module Name	**Functionality**	**Latency (cycles)**
fp_absneg	Absolute Value/Negation	4
fp_add	Addition/Subtraction	4
fp_cmp	Comparison	4
fp_div	Division	12
fp_f2i	Conversion from FP format to integer	4
fp_i2f	Conversion from integer to FP format	4
fp_mult	Multiplication	4

FP Absolute value/Negation Absolute value and negation are pretty straightforward. The inputs to this module are the FP value and the corresponding operator selector opcode indicating either *abs* or *neg*. The output of the module is the input's value exponent and mantissa concatenated with the corresponding manipulation of the sign bit, plus the invalid exception flag.

The internal implementation is composed of one always block that sets the per stage invalid register such that both operations take the desired number of cycles.

FP Addition/Subtraction Addition and subtraction are one of the most computationally complex arithmetic FP operations. The inputs to this module are two FP values and the corresponding operator selector opcode indicating either *add* or *sub*. The output of the module is the normalized form of the result value, in addition to the inexact, invalid, underflow and unimplemented exception flags. The algorithm for the addition/subtraction operation is performed in a three steps pipeline as follows:

1. Select the larger and smaller operands in magnitude
2. 26-bit logical right shift to normalize the smaller operand, and perform the addition/subtraction on the normalized operands
3. 28-bit logical left shift to normalize result

The internal implementation is comprised of a series of always blocks, each of which corresponds to a step in the above algorithm.

FP Compare FP compare is implemented exactly like MIPS with the distinction that the result value is returned, as opposed to set on a flags register. The module takes two FP values and the corresponding operator selector opcode that contains a 4 bit comparison condition code, as describe in Table 4.3, which indicates the desired comparison. The output of the module is the result value of the relation given the selected comparison. Table 4.3 shows all the possible outputs for each relation, along with the invalid exception flag.

The module is implemented internally with one always block that sets the per stage invalid flag and result value registers such that it takes the desired number of cycles. The compare module also checks for corner cases such as NaN, zero, and $\pm\infty$.

Table 4.3: Floating Point Comparison Condition Codes.

Predicate		**Relations(Results)**				**Invalid flag if unordered**
Cond	Definition	Greater Than	Less Than	Equal	Unordered	
0	False	F	F	F	F	No
1	Unordered	F	F	F	T	No
2	Equal	F	F	T	F	No
3	Unordered or Equal	F	F	T	T	No
4	Ordered Less Than	F	T	F	F	No
5	Unordered or Less Than	F	T	F	T	No
6	Ordered Less Than or Equal	F	T	T	F	No
7	Unordered or Less Than or Equal	F	T	T	T	No
8	Signaling False	F	F	F	F	Yes
9	Not Greater Than or Less Than or Equal	F	F	F	T	Yes
10	Signaling Equal	F	F	T	F	Yes
11	Not Greater Than or Less Than	F	F	T	T	Yes
12	Less Than	F	T	F	F	Yes
13	Not Greater Than or Equal	F	T	F	T	Yes
14	Less Than or Equal	F	T	T	F	Yes
15	Not Greater Than	F	T	T	T	Yes

FP Division FP division is implemented as a non pipelined 11 cycle operation that uses MIPS like decoupled HI/LO interface. The *fp_div* module inputs are two FP values and the output is the normalized and rounded approximation of the quotient value, in addition to the division by zero, inexact, invalid, overflow, underflow and unimplemented exception flags. The division algorithm implemented is functional iteration (also known as multiplication based or multiplicative division algorithm) [16], which rely on the fact that division can be written as the product of the dividend and the reciprocal of the divisor. This approach tries to take advantage of highspeed multipliers combine with the accuracy of the initial reciprocal approximation. However, one disadvantage of functional iteration is the lack of a remainder.

In particular, the *fp_div* module uses a 8 bit ROM lookup table for initial reciprocal approximation and the Synopsys Module Compiler (Y2006.06SP5) [26] for generating technology specific custom functional unit for multiplication that is optimized for speed and power usage.

Format Conversion (FP $\rightarrow$ Int/Int $\rightarrow$ FP) Our FPU design would not be complete without modules for converting between the floating point and integer (i.e. fixed point) representations. The module *fp_f2i* implements the conversion from the IEEE's FP representation to its integer representation. The module takes a FP value and the corresponding operator selector opcode indicating if the output should be rounded or not. The output is the integer representation of the FP representation, plus the inexact and invalid exception flags.

Module *fp_i2f*, on the other hand, implements the inverse function of the *fp_f2i* module. It converts from an integer representation to its IEEE's FP representation. This module input is the operand's value, either signed or unsigned, integer representation. The output of the module is the FP representation of the value along with the inexact exception flag.

FP Multiplication The FP multiplication implementation is fairly straightforward (i.e. significantly easier than addition), mainly due to the nature of the IEEE754 FP format. The inputs to this module are two FP values and the output of the module is the normalized form of the result value plus the inexact, invalid, overflow, underflow and unimplemented exception flags. The multiplication algorithm is described as follows:

- the sign of the product is the *XOR* of the operands signs

- the exponent of the product is the sum of the operands exponents
- the mantissa of the product is the product of the operands mantissas

The multiplier internally contains a set of stages which implement the multiplication algorithm presented above. Like FP division, we also make use of the Synopsys Module Compiler, in order to generate technology specific custom functional unit for multiplication that is also optimized for speed and power usage.

FP arithmetic logic extensions In this thesis, in order to allow for a more comprehensive and broad study of FP designs and configurations, we expand the FPU design by implementing each one of the FPU's supported operators as an independent Verilog module. Therefore, instead of having an entire FPU in the presence of one or more FP operations, we allow the ability to only have an independent and isolated instantiation of the FP logic for each particular FP operator that would be needed in the SPE. We accomplish this extension of the FP logic by replacing the FPU top level with a per operation ”top level” that only wraps the corresponding FP operator logic, where the interface is exactly the same as before: The active FP operator logic receives the FP operation request and data, executes it, and has it stream out, along with the ”valid” signal. All subsequent FP operations that require the same operator must wait until the active request is finalized. This smaller per operation FP logic design enables design flexibility and a decrease in area and power requirements, as well as permits more efficient independent FP arithmetic logic implementations that could have smaller cycle latencies for some FP operations. In particular, by having independent FP operator logic, we are able to redesign some FP operators to have a lower latency than the one achieved in the full FPU design as shown in Table 4.4.

Area, energy and performance measurements for the FP logic in hardware The ability of having the FP logic in hardware comes at a cost in area, energy and performance(depending on the selected design). Area and performance costs for the different FP logic designs are shown in Table 4.4, this table does not accounts for energy costs,

Table 4.4: Area and latency numbers for the full FPU and each FP operation logic.

FP Logic Configuration	**Area** (mm^2)	**Latency (cycles)**
FPU	0.11	see Table 4.2
FP ABS/NEG	0.0068	1
FP ADD/SUB	0.017	3
FP CMP	0.00071	1
FP DIV	0.048	11
FP F2I	0.0093	3
FP I2F	0.012	3
FP MULT	0.036	3

since our simulation tool is not yet calibrated to measure per operation energy. The area cost of a FP operation vary from just $0.00071mm^2$ for the ”smallest” per operator FP logic design (i.e. FP CMP) to $0.11mm^2$ for a full FPU. Similarly, our different FP logic configurations can have an impact in performance as shown in Table 4.4, where performance benefits could come from the ability to improve the cycle latencies of the FP logic for most FP operators when designed in isolation.

Testing and Validation Testing and validation of all FP arithmetic operations in the FPU are performed at three levels [15]. At the first level, we develop a Verilog testbench that drives each one of the modules for testing. Each test module consists of a series of operand values and an expected result. The module simply simulates certain signals (such as the ”clock”, ”request” and ”reset” signals) that would otherwise come from the active SPE, and then executes each FP arithmetic operation, reporting errors (if any) and successes using the ”display” function. This very simple strategy worked extremely well for testing.

The second testing level is performed using *Softfloat* [6], a high quality software implementation of the IEEE Standard for Binary Floating point Arithmetic, to generate a file with FP traces from a set of applications to be used as input to feed our FPU testbench.

Finally, as part of our simulation infrastructure, we generate a cycle accurate system simulator for the SPEs which generates traces that we use to validate and drive the synthesized SPEs, describe in detail in Section 5.1.

4.2 Floating Point Hardware Sharing Strategies

4.2.1 Overview

In Section 4.1, we described the underlying logic for each FP arithmetic operation in the FPU. In this section, we describe the design and implementation details for each sharing strategy of the FPU between the SPEs in an Arsenal system. In the thesis, we explore the area, power and performance tradeoff for seven FP logic sharing strategies: a shared FPU between all SPEs in a complex, a private FPU per SPE, a private FPU per SPE but with a shared FP divider across the SPEs complex, a private OnDemand FP logic per SPE, a private OnDemand nondivide FP logic per SPE along with a complex wide shared FP divider, private FP gadgets per SPE, and private non divide FP gadgets per SPE sharing the FP divider between the SPEs complex. In the next subsections we describe each one of them.

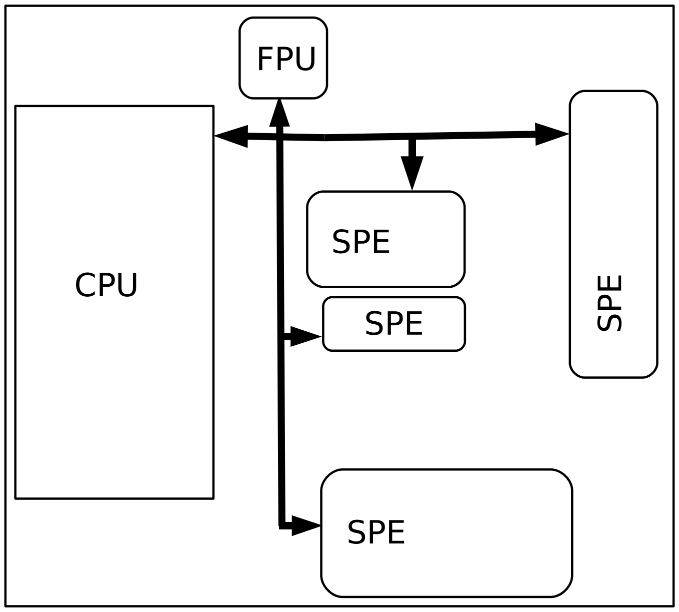


Figure 4.1: **Shared FPU in SPE Complex** It shows an Arsenal complex where a FPU is shared among all SPEs in the complex.

4.2.2 Shared FPU per SPE complex

Design The first FPU sharing strategy considered is a FPU shared between exactly one complex of SPEs, as shown in Figure 4.1. This sharing configuration is also used as the baseline design for area, energy and performance comparisons in Chapter 6. Sharing the FPU across only one SPEs complex becomes prohibitively expensive as the number of SPEs in the complex grows due to potential FPU contention and arbitration among the active SPEs in the complex. Even though, this contention scenario limitation is not currently a concern in Arsenal because only one SPE in the complex is active at once. However, it could significantly degrade performance in subsequent Arsenal designs that may relax the constraint of exactly one active SPE per complex. Additionally, the wire delay's negative impact on performance increases as the number of SPEs sharing a FPU increases, thus exceeding the sharing benefits. We claim that sharing a FPU between more than two SPEs complexes will cause such effect.

The interface between the executing SPE and the FPU is as follows, when the SPE finds a FP operation, it sends the request and data to the FPU which, if not busy, performs the multicycle operation and sends the result back to the SPE to continue execution. Otherwise the SPE request must wait. One of the caveats of such SPE/FPU interface is that it incurs in small performance overhead due to the latency of transferring control and data between the SPE and FPU.

Implementation In section 4.1, we already described the FPU design and implementation details which leaves only the implementation details of what occurs when the SPE encounters a FP operation during the C to Verilog translation stage. In section 5.1.1, we describe Arsenal's toolchain which creates SPEs from an input application source code written in C. As part of the toolchain's C to Verilog compilation stage, a shared FPU is instantiated for the entire SPE's complex where for each FP operation found in the underlying SPE a *fp_local_unit* that connects and transfers control and data to the shared FPU is instantiated. Therefore, when the FP operation is enable the request and input data are sent to the shared FPU by the corresponding *fp_local_unit*. Later on, when the

FPU is done, it sends a *valid* signal, as well as, the result value back to the active SPE so that execution can resume.

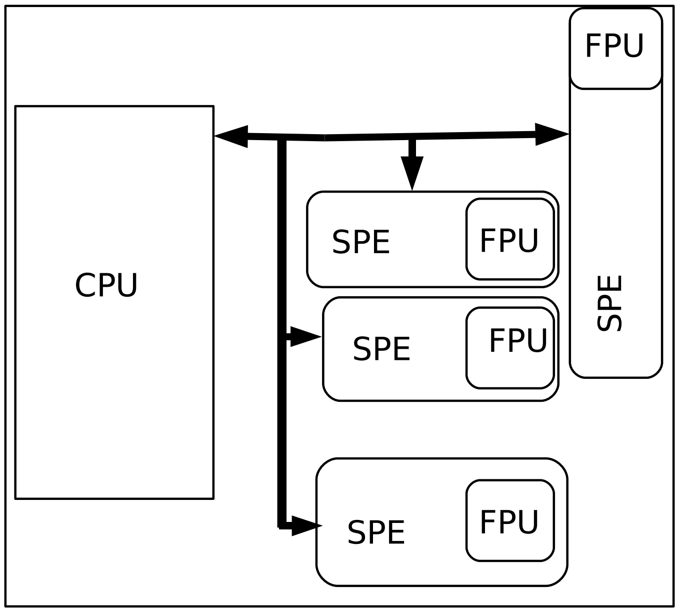


Figure 4.2: **Private FPU per SPE** It shows an Arsenal complex where there is a private FPU for each SPE in the complex.

4.2.3 Private FPU per SPE

Design The second proposed FPU sharing strategy is having a FPU per SPE in the complex, as shown in Figure 4.2. In this sharing design the FPU is private to each SPE in the complex which overcomes the FPU contention limitation of the previous sharing model. However, having a private FPU per SPE requires a larger area per SPE ($0.11mm^2$ as shown in Table 4.4) and increase power requirements and energy consumption, hence, limiting the chip's area and power budget for SPEs, ultimately reducing the number of SPEs in an Arsenal processor.

The interface between the running SPE and the FPU is very similar to the previous one, with the sole distinction that the FPU is private to the SPE, thus, when the SPE finds a FP operation, it transfers the request and data to its private FPU which performs the operation and sends the result back to the SPE to continue execution. As opposed to the baseline, this SPE/FPU interface incurs in a smaller performance overhead because the FPU is actually private to that SPE, and therefore the transferring latency between the SPE and FPU should be less than the overhead of communicating outside the SPE, especially when FPU contention and arbitration are introduced.

Implementation The SPE/FPU interface implementation for this design is exactly the same as before. The difference is that when the SPE encounters a FP operation it first instantiates one private FPU for the SPE such that for each FP operation encountered in the SPE a *fp_local_unit* connecting and transferring control and data with the private FPU is instantiated. Later on, when a particular FP operation is enable the request and input data are transferred to the private FPU by the corresponding FP operation's *fp_local_unit*. Finally, when the FPU is done, it sends a *valid* signal to the SPE along with the result value to continue execution.

4.2.4 Private nondivide FPU per SPE with shared FP divider

Design The third FPU sharing strategy is based on a well documented observation about instruction mix which stipulates that FP division is one of the two least frequently occurring operations in an application [16]. Therefore, in this design, we consider a nondivide FPU per SPE, where the FP divider logic is separated and shared across the SPEs in the complex, as shown in Figure 4.3. In this sharing design the ”partial” FPU (or nondivide FPU) is still private to each SPE, but without all the FP divider logic, that is usually infrequently used. The FP divider is still accessible to all the SPEs in the complex. This sharing mechanism reduces per SPE area and power requirements, and energy consumption on the chip by eliminating the FP divider logic per SPE which may enable an increase in the number of SPEs that could not be allocated by having a completely private FPU per SPE in the complex.

In this case, the interface between the active SPE in the complex and the FPU

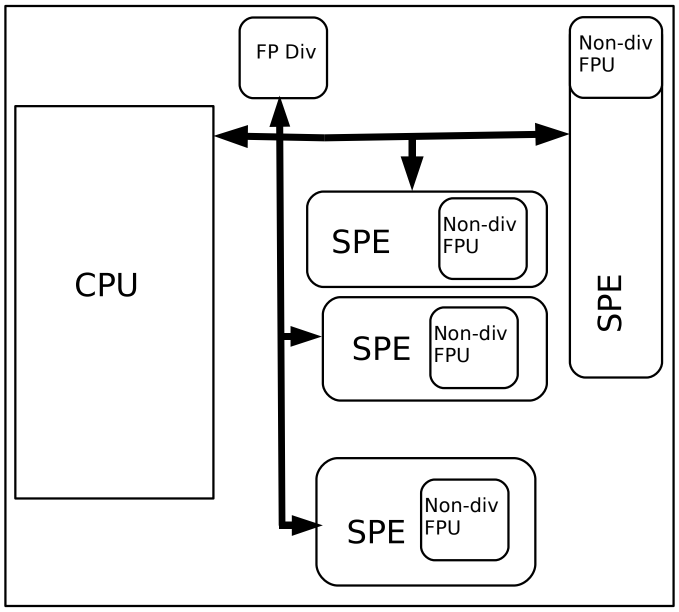


Figure 4.3: **Private nondivide FPU per SPE with shared FP divider per complex** It shows an Arsenal complex where there is a private nondivide FPU for each SPE in the complex and a FP divider shared among all SPEs in the complex.

differs from the previous ones. When the SPE finds a nondivide FP operation, it transfers the request and data to its private FPU which performs the operation and sends the result value back to the SPE to continue execution. However, when the SPE finds a FP divide operation, it transfers the request and data to the shared FP divider logic for the SPEs complex which performs the operation and sends the result back to the SPE to continue execution.

This sharing design differs from the shared and private SPE/FPU interfaces since a very small performance overhead is tradeoff to the control and data transfer between the shared FP divider outside the SPE, whereas the fully private partial FPU stills maintains a reduced transfer overhead between the SPE and internal nondivide FPU for the most frequently occurring operations in an application, and thus enabling an area, power and energy savings.

Implementation The implementation for the SPE/FPU interface for this design is very similar to the ones defined before. For nondivide operations one nondivide private FPU is instantiated and for each nondivide FP operation encountered in the SPE a *nondiv_fp_local_unit* connecting and transferring control and data with the nondivide FPU is instantiated. Similarly, for each FP divide operation in the SPE a similar *fp_div_local_unit* connecting and communicating with the shared FP divider is instantiated. Later on, when a FP operation is enable, if it is a nondivide FP operation the request and input values are transferred to the nondivide FPU by the matching FP operation's *nondiv_fp_local_unit*. Otherwise, if it is a FP divide operation the request and operand values are transferred to the shared FP divider by the operation's *fp_div_local_unit*. When either, the nondivide FPU and FP divider, are done they send a *valid* signal and the respective result value back to the SPE to continue execution.

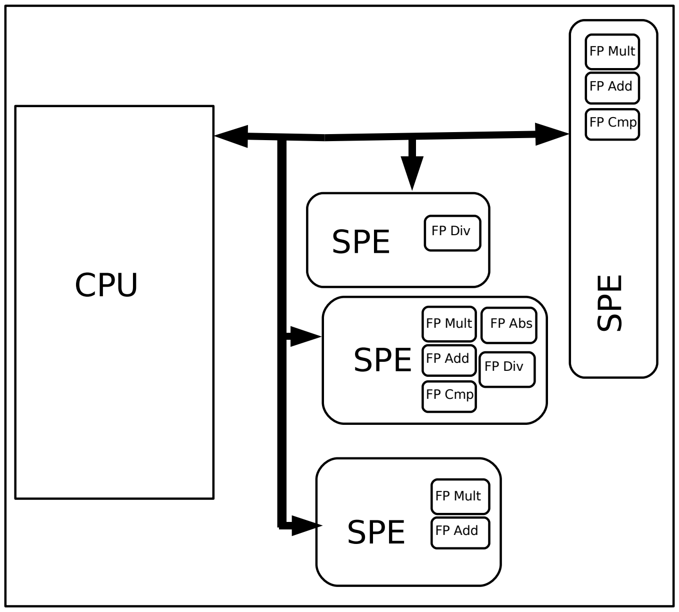


Figure 4.4: **Private OnDemand FP Operator Logic per SPE** It shows an Arsenal complex where each particular FP Operator Logic needed per SPE is privately and independently instantiated for each SPE in the complex.

4.2.5 Private OnDemand FP Operator Logic per SPE

Design The fourth studied FP logic sharing strategy goes one step further than the second FP logic sharing configuration, in this case, it is based on previous work that reports that FP addition and FP multiplication are the two most frequently occurring operations in an application [16]. Therefore, the proposed FP logic configuration strategy proposes that instead of instantiating one full FPU if there are one or more FP operations, we could only independently instantiate the logic for each particular FP operator that is needed in the SPE as shown in Figure 4.4. In this FP configuration design some parts of the FPU are partially dissolved as only the required logic is created and instantiated on an ondemand basis. This sharing mechanism enables a reduction in per SPE area, power and energy demands, as well as, allows for more efficient independent FP arithmetic logic implementations that could have smaller cycle latencies for some FP operations as previously shown in Table 4.4.

The interface between the running SPE and the separate FP operators logic is very similar to the second FP sharing design. In this case, however, each FP operator logic is not only private but independent of other FP operators in the SPE, hence, when the SPE finds a FP operation, it transfers the control and data to its private and "stand alone" instantiation of the logic for that FP operation. After the FP operation is performed, the result value is send back to the SPE to continue execution. This configuration allows for parallelization of different FP operations. Like the SPE/private FPU interface there is a small performance overhead due to transferring control and data between the SPE and the independent FP operators logic.

Implementation The SPE/FP logic interface implementation for this design mirrors the private FPU one. When the SPE encounters a FP operation it first instantiates one private and independent FP operator logic for that particular operator in the SPE so that for each similar FP operation encountered in the SPE a fp_{op}*_local_unit* connecting and communicating with that private FP operator logic gets instantiated. After a particular FP operation is enable, the request and input data are transferred to the independent FP operator logic by the corresponding FP operation's $fp_{op}\_local\_unit$. Finally, when the FP operator logic is done, it sends a *valid* signal to the SPE along with the result value to continue execution.

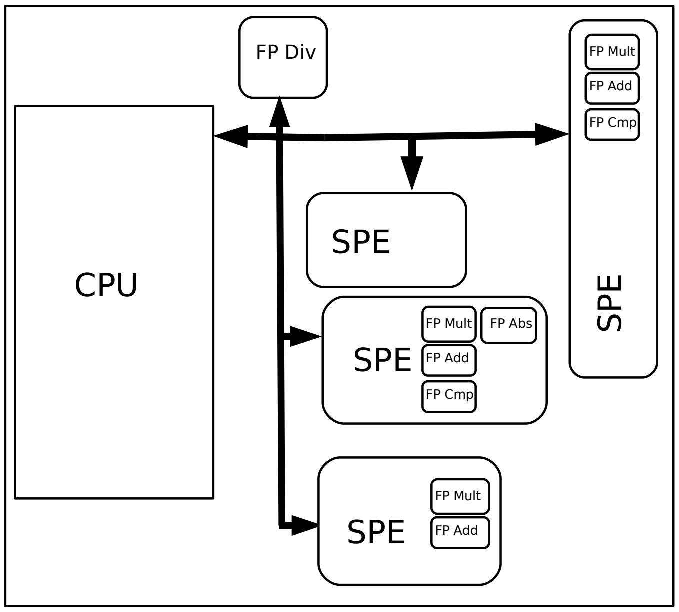


Figure 4.5: **Private OnDemand nondivide FP Operator Logic per SPE with shared FP divider per SPEs complex** It shows an Arsenal complex where each particular nondivide FP Operator Logic needed per SPE is privately and independently instantiated for each SPE in the complex and a FP divider shared among all SPEs in the complex.

4.2.6 Private OnDemand nondivide FP Operator Logic per SPE with shared FP divider

Design The fifth sharing strategy combines the previous FP configuration design with a SPEs complex shared FP divider, as shown in Figure 4.5. In this FP configuration design the FP divider logic is completely removed as a possible private and independent FP operator in a SPE, while the other operators are created and instantiated on an ondemand basis as the SPE requires them. This sharing mechanism enables a reduction in per SPE area, power and energy that comes from removing the FP divider logic.

The interface between the running SPE and the separate FP operators logic is very similar to the ones describe before. If the SPE finds a nondivide FP operation it transfers the request and data to its private and stand alone instantiation of the nondivide FP logic for that FP operation. After the nondivide FP operation is performed, the result value is send back to the SPE to continue execution. On the other hand, if the SPE finds a FP division operation, it transfers the request and data to the shared FP divider logic for the SPEs complex which performs the operation and sends the result back to the SPE to continue execution.

Like the SPE/private FPU with shared divider interface, this sharing configuration tries to balance the tradeoff between area, power, energy and performance. To one end, it plays with the increase in the performance overhead of transferring control and data with an outside shared FP divider for each FP divide operation, and, at the other end, balancing it with a more efficient and area conscious but power hungry configuration for each nondivide FP operation.

Implementation The SPE/FP logic interface implementation for this design resembles the ones defined before. When the SPE encounters a nondivide FP operation it first instantiates one private and independent FP operator logic for that particular operator in the SPE so that for each similar nondivide FP operation encountered in the SPE a *nondiv_fp$_{op}$_local_unit* connecting and communicating with the private FP operator logic gets instantiated. Similarly, for each FP divide operation in the SPE a similar *fp_div_local_unit* connecting and communicating with the shared FP divider is instantiated. Later on, when a FP operation is enable, if it is a nondivide FP operation the request and input data are transferred to the independent FP operator logic by the corresponding FP operation's *nondiv_fp$_{op}$_local_unit*. Otherwise, if it is a FP divide operation the request and operand values are transferred to the shared FP divider by the operation's *fp_div_local_unit*. Finally, when either the FP operator logic or the shared FP divider is done, a *valid* signal and the result value are sent back to the SPE to continue execution.

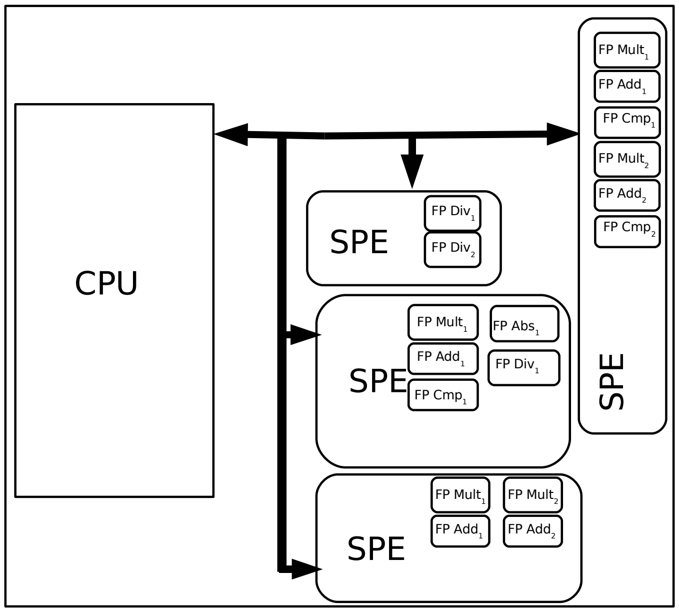


Figure 4.6: **Private FP Gadgets per SPE** It shows an Arsenal complex where for each FP operation the corresponding logic is privately and independently instantiated per SPE in the complex.

4.2.7 Private FP Gadgets per SPE

Design The sixth proposed FPU sharing strategy builds upon and extends the private OnDemand FP operator Logic configuration by taking it to the extreme case. It augments the SPE with private FP gadgets such that the SPE does not have transfer control and data, at all, with the FPU. A FP gadget is the replacement of the need for an FPU by inserting the corresponding logic for each one of the FP operators in the basic blocks of the SPE, hence, enabling the underlying SPE to instantiate the FP gadget at runtime without the overhead of switching or transferring the data to and from the FPU. In this configuration the FPU is completely dissolved and only private FP gadgets are created as the SPE requires them, as show in Figure 4.6. This sharing mechanism reduces even further the performance overhead of communication and arbitration with the FPU. Nevertheless, FP gadgets on the critical path of the execution may increase cycle time.

Also, the FP gadgets significantly increase the SPE area and power requirements, since for each FP operation in the basic blocks of the SPE we have to insert the actual FP gadget operator logic, as well as instantiate it. This sharing approach stresses the available SPE area and power budget to a point which could easily become impractical.

The interface between the running SPE and the FPU is completely removed, since at runtime when the SPE finds a FP operation, it can instantiate its private FP gadget for it and continue execution.

Implementation The implementation of FP gadgets requires that during the C to Verilog compilation phase, when a SPE FP operation is found the corresponding FP gadget operator logic must be inserted and instantiated. In this case, the interface completely differs with previous ones because the active SPE never transfers control to the FPU. Therefore, the SPE can execute a FP operation using the corresponding gadget without interrupting execution.

4.2.8 Private nondivide FP Gadgets per SPE with shared FP divider

The seventh and final FPU sharing strategy builds upon trying to reduce the area and power demands that FP gadgets have on the SPE area and power requirements. Hence, it combines FP gadgets with a SPEs complex shared FP divider. In this case, we are still augmenting the SPE with private FP gadgets but since FP divide operations do not occur very often, instead of having a FP divide gadget for each FP divide operation, we have a shared FP divider across the complex, as illustrated in Figure 4.7. In this sharing design the FPU is almost completely removed except for the FP divider. As explained before, the private nondivide FP gadgets are created as the SPE requires them. This sharing mechanism reduces the performance overhead of transferring control to and from the FPU for all but the FP divide unit, that even though usually infrequent, incurs in a larger performance overhead. As explained before, the FP divider is accessible to all the SPEs in the complex. This sharing configurations goal is to slightly relax the SPE area and power requirements of gadgets, and increase the available SPE area and power

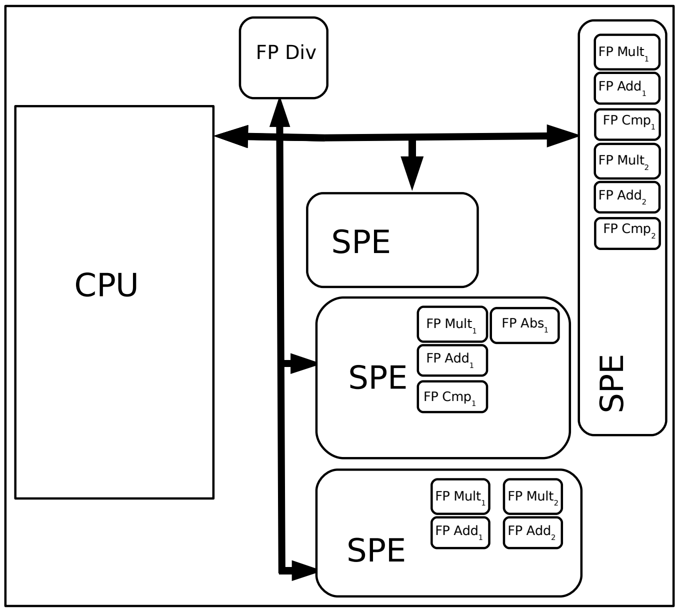


Figure 4.7: **Private nondivide FP Gadgets per SPE with shared FP divider per SPEs complex** It shows an Arsenal complex where for each nondivide FP operation the corresponding logic is privately and independently instantiated per SPE in the complex and a FP divider shared among all SPEs in the complex.

budget such that FP gadgets do not become impractical.

The interface between the running SPE and the FPU is remove for all nondivide FP operations. When the SPE finds a nondivide FP operation, it can instantiate its corresponding private FP gadget logic and continue execution. However, if a divide operation is encountered in the SPE, the request and data are transferred to the shared FP divider for the SPEs complex which performs the operation and sends the result value back to the SPE to continue execution.

Implementation Like the sixth sharing design the implementation requires that during the C to Verilog compilation, when a SPE nondivide FP operation is found the particular FP gadget must be inserted and instantiated. This enables the SPE to execute nondivide FP operations without interrupting execution. On the other hand, when, for each FP divide operation in the SPE a *fp_div_local_unit* connecting and communicating with the shared FP divider is instantiated. Later on, when a FP division operation is enable, the request and operand values are transferred to the shared FP divider by the operation's *fp_div_local_unit*. Finally, when the shared FP divider is done, a *valid* signal and the result value are sent back to the SPE to continue execution.

Chapter 5

Evaluation Methodology

This thesis goal is to explore and analyze the tradeoff between area, energy and performance of distinct designs for sharing FP accelerators under a sample set of FP applications that use a different number of SPEs. Although, specialized processors and ASICs are known to accelerate and be well suited for certain types of applications (e.g. signal processing, multimedia), the SPEs design space for other, more irregular, FP applications is not well understood. Thus, our motivation is to explore the design of SPEs in Arsenal for several FP applications, and, more importantly, measure the suitability of our proposed FP accelerator sharing models between the applications.

The next two sections describe in detail Arsenal's system modeling infrastructure, as well as, measurement tools, and the benchmarks set studied.

5.1 Methodology

Our SPE synthesis toolchain takes C programs as input, splits them into datapath and control path segments, and then uses a state of the art electronic design automation (EDA) tool flow to generate a circuit fully realizable in silicon. In addition to that it concurrently and automatically generates a cycle accurate system simulator for the new hardware. This simulator is used to generate traces that drive the placed and routed netlist in Synopsys VCS and PrimeTime for power measurement.

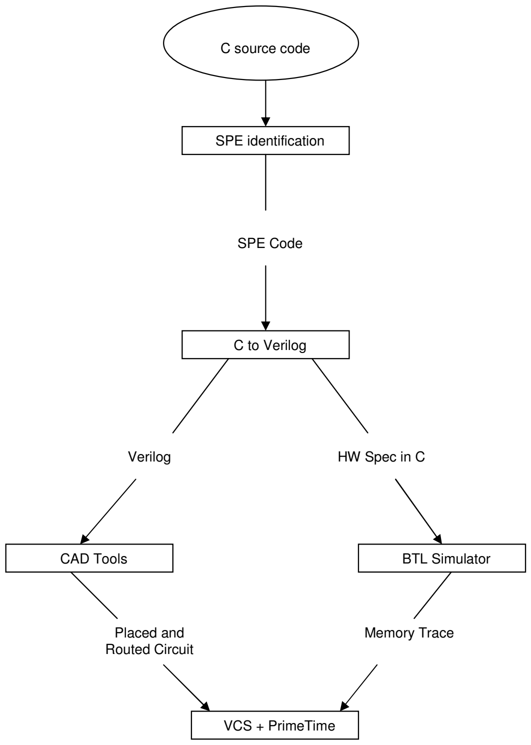


Figure 5.1: **Arsenal C to Verilog Toolchain** Flows for C to hardware, simulation, and power measurement infrastructure.

5.1.1 Arsenal Toolchain

Figure 5.1 summarizes the toolchain we developed to generate SPEs. Arsenal's toolchain is based on mature research and commercial infrastructures such as OpenImpact(1.0rc4) [17], LLVM (2.4) [13] and CodeSurfer(2.1p1) [23]. The toolchain takes a subset of C applications that includes arbitrary pointer references, switch statements, and loops with complex conditions. It does not supports function pointers.

At the SPE identification phase we use profile information to identify code regions that can be turn into SPEs (or *SPEized*). Converting the underlying code region usually involves a series of steps. First, outlining is used to separate the code region we are interested in. Later, during a second step, function calls are removed using inlining. Finally, global variables are added as additional input arguments and pass by reference.

The C to Verilog phase translates the modified input C code (labeled SPE code) into a set of synthesizable Verilog modules using the generated control and dataflow graphs for each identified code region in SSA form [2] to be *SPEized.* This phase also adds basic blocks and control states for each memory operation and multicycle instruction, as described in 2.2. The final product of the toolchain is synthesizable Verilog code for the SPE which requires converting operators into muxes, inserting registers at the definition of each value, and adding self loops to the CFG for the multicycle operations. After completing these passes, the toolchain generates Verilog to instantiate each of the necessary operators, registers, and wires to create the dataflow graph in silicon. Then, it generates the control unit with a FSM that resembles the input's CFG. This phase of the toolchain also generates a C++ cycle accurate module for our architectural simulator describe next.

5.1.2 Simulation Infrastructure

The simulation infrastructure is based on btl, the Raw simulator [28]. Btl is modified so that it can model cache coherent memory among the multiple SPEs complexes, include a scanchain interface between the complex CPU and SPEs (described in detail in Section 5.2), and to simulate the generated SPEs. The SPEs operate at lower clock frequencies than the core complex clock, and even further at different frequencies from each other. The ratio of processor clock to SPE clocks is restricted to integer multiples.

The CPU is an 660MHz, 8 stage, single issue RISC pipeline that implements a MIPS like instruction set that is based on the Raw [28] ISA. It has a 32KB L1 cache and 16KB I/D cache.

5.1.3 Synthesis

The Synopsys Design Compiler (Y2006.06SP6) [26] and Astro (Z2007.03SP10) CAD flow tools are used for SPEs synthesis where we target a TSMC 90nm G process technology. As part of our toolchain, the generated synthesizable Verilog design for each one of the SPEs is then automatically pass and process into the Synopsys CAD tool flow, starting with netlist generation and continuing through placement, clock tree synthesis, and routing, before performing post route optimizations (oriented towards speed and power, not area). This tools measure area and frequency for each one of the SPEs and their associated hardware resources, if any. The Synopsys Module Compiler is use to generate technology specific custom functional units for some basic arithmetic (e.g. addition) and bit shifting. Our target ratio between processor frequency and SPE frequency is 2:1 (i.e. 330:660 MHz). However, the circuits for most SPEs designs are currently synthesizing to between 8:1 and 4:1 ratios. We believe that the 330 MHz frequency for SPEs would be met with some additional tuning and retiming, thus, performance and energy values in Chapter 6 are presented assuming a 2:1 ratio is achieved for all SPEs.

5.1.4 Power measurements

Power measurements for each SPE come from periodical execution samples that the simulator stores in traces of all offchip activity. Each sample starts with a snapshot recording the entire register state of the SPE and continues for 10,000 cycles. In thesis, the sampling policy is to sample 10,000 out of every 50,000 cycles, and in order to capture typical periods of execution, we only process the middle 30 samples.

Each sample is the feed from btl into the Synopsys VCS (B2008.12) [26] logic simulator. The Arsenal toolchain also automatically generates a Verilog testbench module, which initiates the simulation of each sample by scanning in the register values from each btl snapshot. The VCS simulation generates a VCD activity file, which we pipe as input into Synopsys PrimeTime (B2008.12SP2) [26]. PrimeTime computes both the static and dynamic power for each sampling period. Since the majority of the registers and logic in our SPE are inactive at any given time, we reduce the reported internal power by the average activity factor, which we measure during btl simulation. It is assume that when registers are inactive, they consume 10% of their normal operating power.

The power numbers for the I cache and D cache, and derive processor are measure with Cacti 5.3 [29]. Additionally, power numbers for clock power values are model using specs for a 660MHz MIPS 24KE processor in TSMC 90nm G. Finally, for component ratios for Raw the power model uses power numbers reported in [10].

5.2 Benchmarks

To characterize and validate our FP sharing designs study such that they reflect the wide range of FP applications that may be run in Arsenal, we choose a sample set of FP applications that ranges from encoding and decoding, scientific applications to other applications that are amenable to hardware designs. These applications are either well known or are drawn from benchmarks, and their respective source code is written in C. The set of applications includes *Lame MP3 Encoder* [12], EMBC's *Basic Floating Point Automotive* [4], and SPEC2000's floating point applications [22] *equake* and *mesa* and integer application *vpr*.

For each FP application, we target and transform code regions from the most intensively executed functions such that the SPEs can be easily and automatically created from Arsenal's toolchain, as described in Section 5.1.1. Those SPEs are later synthesized, enabling substantial benefits in an Arsenal system. The following subsections, describe in detail each one of the applications studied and the underlying SPEs for each application.

5.2.1 Lame

Lame is an application used for MP3 encoding. In this thesis, version 3.95 of Lame is used with the default parameters. We compiled Lame with profiling enabled to identify the set of code regions that comprise a large portion of execution time; from that set we choose the most intensively executed function in Lame 3.95, namely, `AnalyzeSamples` to be design and synthesize into a SPE.

After outlining a large region of `AnalyzeSamples` the resulting region is rename to `AnalyzeSamples_Outlined` which accounts for 24.62% of execution time on native execution and is called 10,473 times for a relatively large *.wav* input file. Then a SPE is created and synthesized using the Arsenal toolchain, the underlying SPE accelerates a large region of the function that analyzes all samples to calculate the recommended decibels (dB) level change. `AnalyzeSamples_Outlined` contains 156 FP operations, which breakdown is: 20 Abs/Neg, 66 Add/Sub and 70 Mult.

5.2.2 EEMBC

The second application comes from the AutoBench suite of the Embedded Microprocessor Benchmark Consortium (EEMBC). These benchmarks help predict the performance of embedded processors in automotive, industrial, and general purpose applications. In this thesis we are only interested on the Basic Floating Point Automotive Algorithm. It was compiled and profile using the default parameters for multiple consecutive iterations, and we identified the set of functions that comprise the largest region of execution time.

In this case, we *SPEized* the most intensively executed function in Basic Floating Point Automotive, named `t_run_test`. As part of Arsenal's toolchain the desire SPE is converted from an outlined code region of `t_run_test`, rename to `t_run_test_Outlined` which accounts for 99% of execution time on native execution and is called exactly once. The synthesized SPE accelerates the actual main algorithm, and contains 48 FP operations distributed as: 9 Add/Sub, 18 Cmp, 6 Div, and 15 Mult.

5.2.3 SPEC2000 FP and INT

183.equake The first SPEC floating point benchmark is *equake* which simulates the propagation of seismic waves in large, highly heterogeneous valleys. In particular, we use the *equake* version in SPEC2000 along with the default parameters. Functions that dominate execution time are determined via profiling, and of those, we selected `smvp` which is the most heavily executed function in *equake* to be designed into a SPE.

We *SPEized* the entire `smvp` function which accounts for 37.51% of execution time and is called 34 times for the test input *test.in*. The generated SPE has 93 FP operations divided as: 33 Add/Sub, 18 Cmp and 42 Mult.

177.mesa In this thesis, we use SPEC2000's version of the 3D graphics library *mesa* under the MinneSPEC [11] recommended parameters. This benchmark is an OpenGL like library. Like the previously describe benchmarks, we compiled *mesa* with profiling enabled to identify the set of functions that accounts for the largest piece of execution time and selected `gl_depth_test_span_less` and `gl_color_shade_vertices_fast` subroutines as the targets to be design and synthesize into SPEs.

Both target subroutines were entirely *SPEized* and together they comprise 7.48% of execution time and `gl_depth_test_span_less` is called 126,287 times and `gl_color_shade_vertices_fast` is called 100 times when run with the input test file *mesa.in*. There are 45 total FP operations between both SPEs that break down as follows: 13 Add/Sub, 7 Cmp, 8 F2I and 17 Mult.

175.vpr (place) The *vpr* SPEC integer benchmark is a FPGA circuit place and route application. SPEC2000's version of *vpr* is used with the default parameters. Execution of vpr (place) almost entirely comprises repeated calls within `try_place` to the subroutine `try_swap`. Therefore the subroutine `try_swap` is our target code region to be turned into a SPE.

In `try_swap`, we outline and rename a large portion to `try_swap_Outlined` to be *SPEized.* `try_swap_Outlined` accounts for 69.23% of execution time and is called 499,999 times for the test *.arch* and *.net* input files. The resulting SPE contains only 12 FP operations separated as: 4 Add/Sub, 3 I2F and 5 Mult.

Table 5.1: Summary of SPEs execution coverage and Floating Point (FP) operations breakdown by application.

Application	**SPE Execution Coverage (%)**	**Breakdown of FP Operations in SPE**
Lame MP3 Encoder	24.62	20 Abs/Neg
		66 Ad/Sub
		70 Mult
		156 Total FP Ops
Basic FP Automotive	99.0	9 Add/Sub
		18 Cmp
		6 Div
		15 Mult
		48 Total FP Ops
Equake	37.51	33 Add/Sub
		18 Cmp
		42 Mult
		93 Total FP Ops
Mesa	7.48	13 Add/Sub
		7 Cmp
		8 F2I
		17 Mult
		45 Total FP Ops
Vpr	69.23	4 ADD/SUB
		3 I2F
		5 Mult
		12 Total FP Ops

5.2.4 Summary

In Table 5.1 we summarize the set of applications along with their respective SPE execution coverage and breakdown of the number of FP operations per SPE. The percentage of SPE coverage ranges from 7.48% for *mesa* to 99% for *basic FP Automotive*, while the number of FP operations varies from 12 operations in *vpr* to 156 operations in *lame*. It is noteworthy that the FP instruction mix for our sample set of applications adheres and is consistent with previous work reports that claim that FP Add/Sub and FP Mult are the most frequently occurring operations in an application while FP Div is one of the least frequently occurring operations in an application. In the next chapter we present the area, energy and performance simulation results for our proposed FP hardware sharing designs.

Chapter 6

Results

This chapter presents the area, energy and performance of the Arsenal system under the seven described FP hardware sharing mechanisms. First, a characterization of the area, energy, performance and efficiency measurements for simple SPEs exhibiting each FP hardware designs is shown. Then, we explore the area, energy, performance and efficiency of the proposed FP hardware sharing configurations on a set of floating point applications.

6.1 Costs and benefits of FP hardware support for SPEs

In Section 4.1.2 we described the area costs of the FP logic, in this section we evaluate the area, energy, performance and efficiency tradeoff of FP hardware support on a simple SPE for each proposed FP sharing configuration. To quantify these measurements, we synthesized a simple SPE that executed a subset of supported FP operations (ADD, MULT, DIV), and then extracted the area, frequency, energy, performance and efficiency numbers of each design. It is possible that in larger SPE designs performance overhead could come by potential increases on the critical path length of an SPE which ultimately may increase the achievable clock speed. Table 6.1 summarizes the area and frequency of each simple SPE under each FPU configuration design (i.e. shared FPU, private FPU, etc). The data shows that SPE area increases between 0.016 mm^2 to 0.11 mm^2 when compared to running on a shared FPU configuration. However, this table could be misleading since the FP gadgets area increases as a function of the number of FP operations.

Table 6.1: Area and frequency values for simple SPEs under each FPU configuration design.

SPE Design	**Area (mm^2)**	**Frequency (MHz)**
Shared FPU	0.023	192
Private FPU	0.134	206
Private FPU w/ shared FP divider	0.087	195
Private OnDemand FPU	0.076	283
Private OnDemandFPU w/Shared Divider	0.039	279
Private FP Gadgets	0.076	280
Private FP Gadgets w/Shared Divider	0.039	277

Additionally, Figure 6.1 presents the performance gains of these simple SPEs under each FP hardware designs when compared to running on software. As expected, the proposed FP hardware configurations reduced execution time by 5%, however, the individual benefits of each configuration cannot be separated among these simple SPEs, but will become clear as the number of FP operations in the SPE increases.

Figure 6.2 explores the energy breakdown for each one of the simple SPEs under each of the FP configurations compared to running entirely on software. The graph shows energy savings of 4% for each configuration. As with Figure 6.1, energy impact per configuration would become distinguishable as the number of FP operations in the SPE increases.

Figure 6.3 summarizes the impact of the FP hardware designs on energy delay product. Again, each FP configuration reduced the energy delay product by 8% for the simple SPE, but the change variation between configurations cannot be appreciated.

6.2 Benchmarks results

The overall goal of this thesis is to present a comprehensive study of the trade off between area, energy, performance and efficiency, as well as find a proper balance point between them. Table 6.2 summarizes the area, frequency and execution time coverage numbers of each FP hardware sharing design in Arsenal for a set of real world

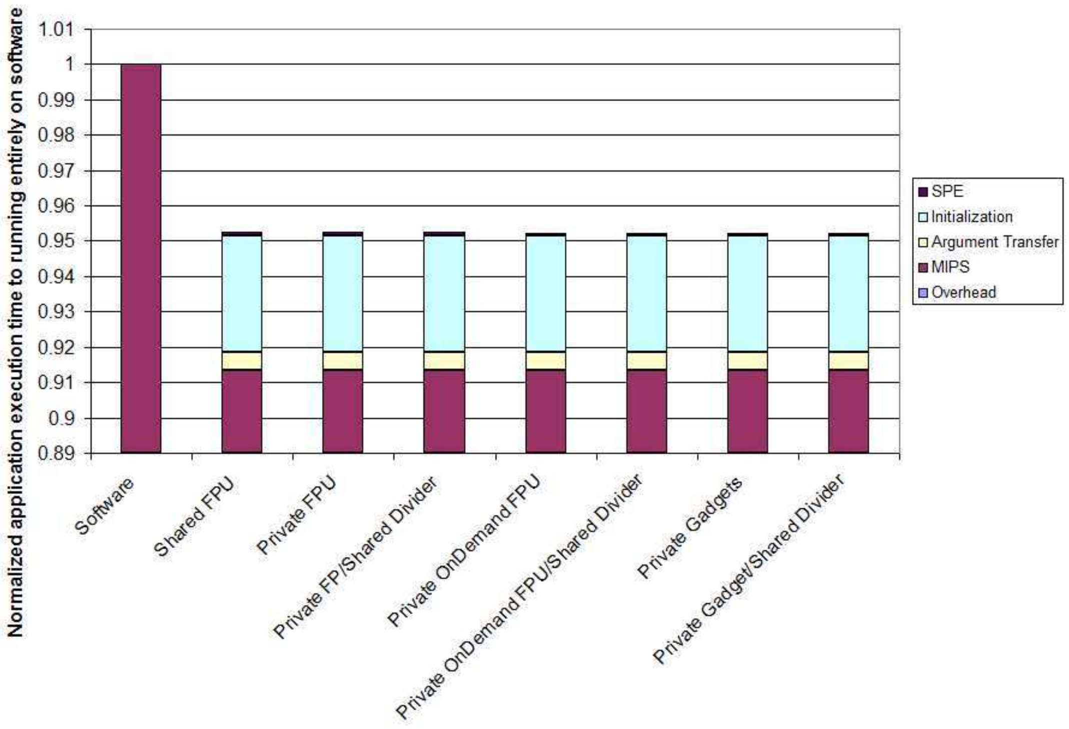


Figure 6.1: **Execution time for simple SPEs under each floating point hardware configuration** Results are normalized to running entirely on software.

floating point applications.

Figure 6.4 shows the area for all the proposed FP hardware configurations for the floating point applications. The data shows that the proposed FP designs increase SPE overall area between 3.6% and 435% when compared to the shared FPU design. As expected, the FP gadgets configuration becomes impractical as the number of FP operations increases. The private FPU and private FPU w/shared divider increase area per SPE between 6.5% and 13.5%. Additionally, the OnDemand FPU w/shared divider configuration delivers minimal increases in SPE area for private FP hardware support. In general, these area results are of particular interest to our group because we are proposing SPEs with facilities that allow them to be patched so they can adapt to new versions of software that become available [30].

Figure 6.5 presents the impact of the FP hardware designs on runtime. The

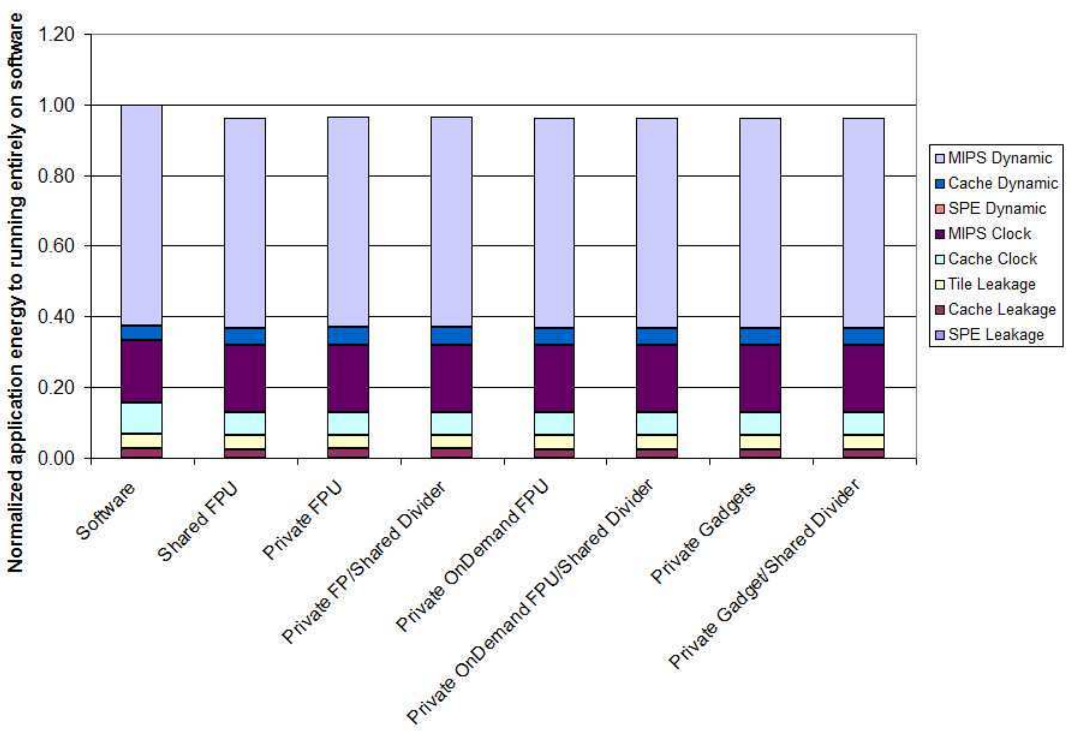


Figure 6.2: **Energy breakdown for simple SPEs under each floating point hardware configuration** Results are normalized to running entirely on software.

graph shows that SPEs under the FP gadgets configuration provides the largest performance gains between 0.5% and 14% when compared to a shared FPU configuration. However, this configuration strategy was shown to be impractical earlier. The OnDemand FPU delivers performance gains that range from 0.5% to 7% when compared to a shared FPU configuration while still managing to keep low area requirements. Observe that private FPU and shared FPU configuration provide same performance, this is a direct result from our assumption that only one SPE could be active per SPEs complex, and, hence, no FPU contention was modeled in the shared configuration. Therefore, one should expect the shared FPU performance to degrade when contention is introduced. Finally, *mesa*'s SPEs only provide between 0.1% and 0.5% performance gains due to the small execution time coverage of the SPE.

Figure 6.6 explores the energy breakdown for each one of the SPEs under each

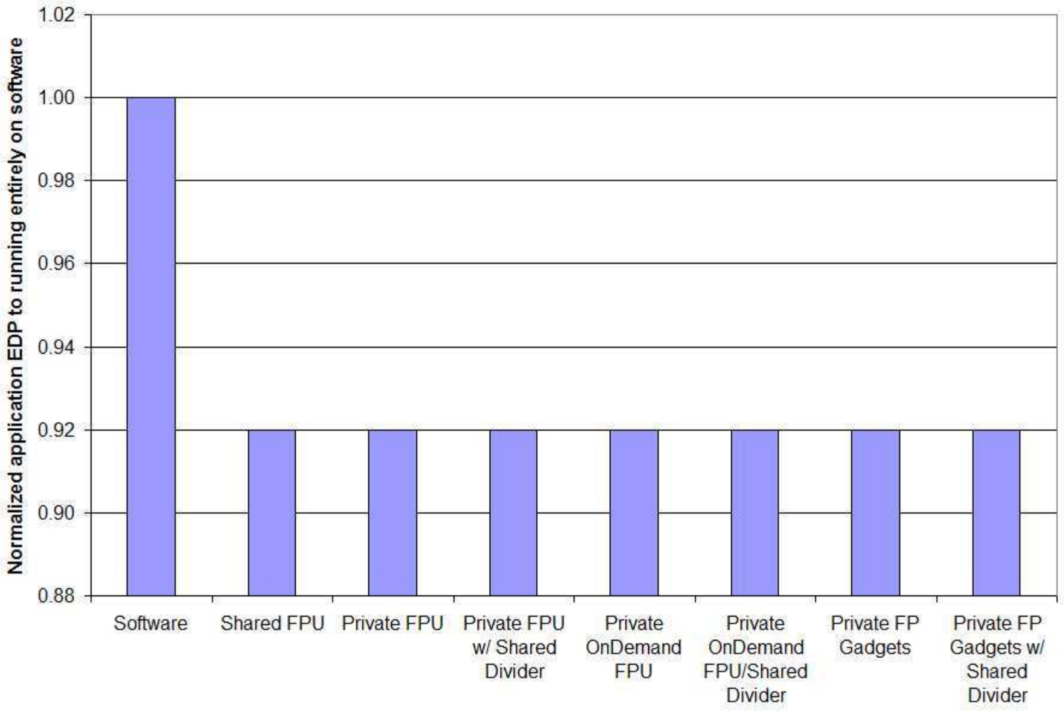


Figure 6.3: **Energy delay product for simple SPEs under each floating point hardware configuration** Results are normalized to running entirely on software.

of the FP configurations running the applications normalized to the energy breakdown of the application running on a shared FPU configuration. The data shows that SPEs energy consumption can increase between 0.5% and 6% for the OnDemand FPU configuration, and up to 1% and 102% for the FP gadgets configuration when compared to running on a shared FPU configuration. This dramatic increase in energy consumption for the FP gadgets relies on the fact that it depends on the number of FP operations, therefore, as the number of FP operations increases so does the energy consumption.

Finally, Figure 6.7 summarizes the impact of the FP hardware designs on the applications energy delay product. The results show that only the Ondemand FPU and OnDemand FPU w/shared divider deliver maximum energy delay product reductions of 4% when compared to the shared FPU configuration. Additionally, the impracticality of FP gadgets is reaffirmed with an energy delay product ranging from 2% to 35%.

Table 6.2: Breakdown of area, frequency and SPE execution time coverage for the applications SPEs under each FP sharing strategy.

Benchmark	**SPEized Application**	**FP Sharing Strategy**	**Area** (mm^2)	**Freq. (MHz)**	**% SPE Cycles**
MP3 Encoder	lame version 3.95	Shared FPU	1.53	135	18
		Private FPU	1.72	151	18
		Private FPU w/Shared Divider	1.68	144	18
		Private OnDemand FPU	1.58	159	16
		Private OnDemandFPU w/Shared Divider	1.58	153	16
		Private FP Gadgets	5.36	159	13
		Private FP Gadgets w/Shared Divider	5.36	157	13
EEMBC	basic fp automotive	Shared FPU	0.82	164	99
		Private FPU	0.94	191	99
		Private FPU w/Shared Divider	0.88	168	99
		Private OnDemand FPU	0.92	225	99
		Private OnDemandFPU w/Shared Divider	0.88	203	99
		Private FP Gadgets	1.81	223	99
		Private FP Gadgets w/Shared Divider	1.48	205	99
SPEC2000	equake	Shared FPU	0.63	163	42
		Private FPU	0.75	178	42
		Private FPU w/Shared Divider	0.70	168	42
		Private OnDemand FPU	0.69	217	39
		Private OnDemandFPU w/Shared Divider	0.69	209	39
		Private FP Gadgets	2.77	191	36
		Private FP Gadgets w/Shared Divider	2.77	185	36
	mesa	Shared FPU	0.47	143	4
		Private FPU	0.58	162	4
		Private FPU w/Shared Divider	0.53	153	4
		Private OnDemand FPU	0.54	171	4
		Private OnDemandFPU w/Shared Divider	0.54	166	4
		Private FP Gadgets	1.36	174	4
		Private FP Gadgets w/Shared Divider	1.36	170	4
	vpr	Shared FPU	1.63	121	61
		Private FPU	1.74	134	61
		Private FPU w/Shared Divider	1.69	124	61
		Private OnDemand FPU	1.71	136	60
		Private OnDemandFPU w/Shared Divider	1.71	133	60
		Private FP Gadgets	1.91	137	59
		Private FP Gadgets w/Shared Divider	1.91	133	59

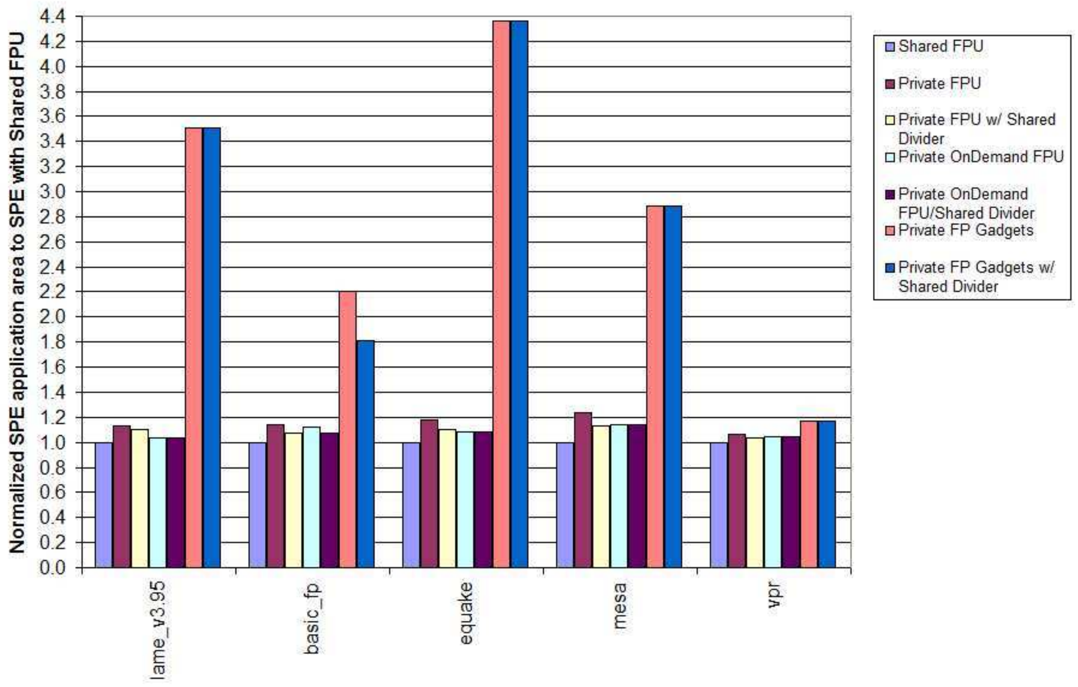


Figure 6.4: **Application SPE area for all five benchmarks** Each subgroup of bars represents a specific floating point hardware configuration. Results are normalized to running on a shared FPU configuration.

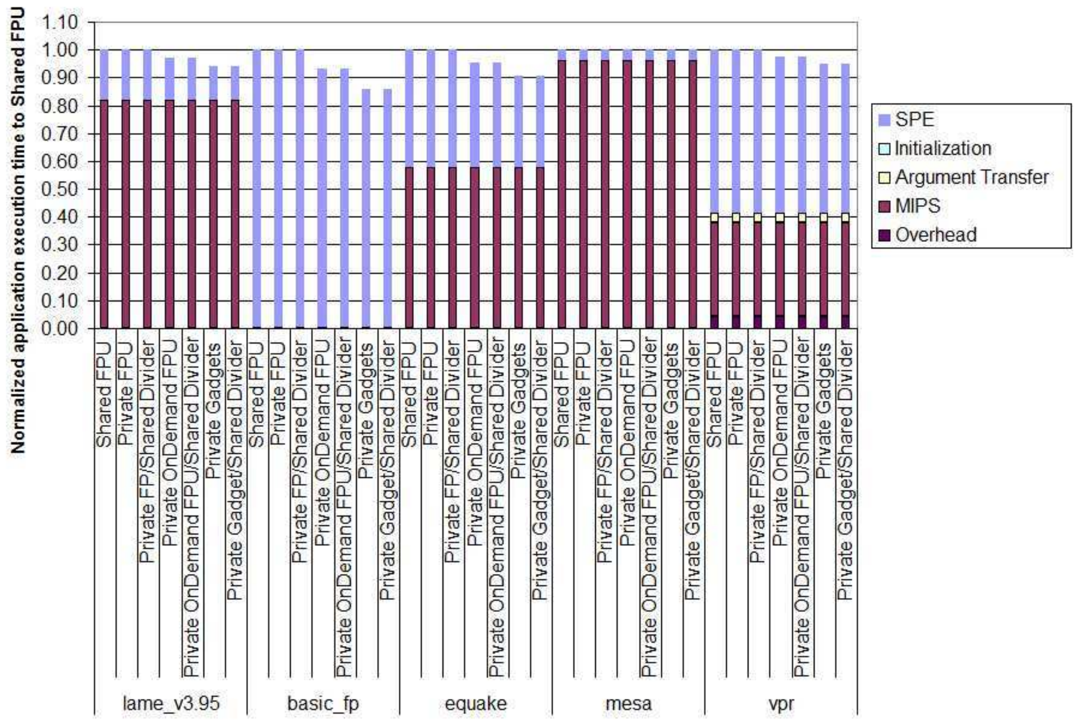


Figure 6.5: **Application execution time for all five benchmarks** Each subgroup of bars represents a specific floating point hardware configuration. Results are normalized to shared FPU configuration execution time.

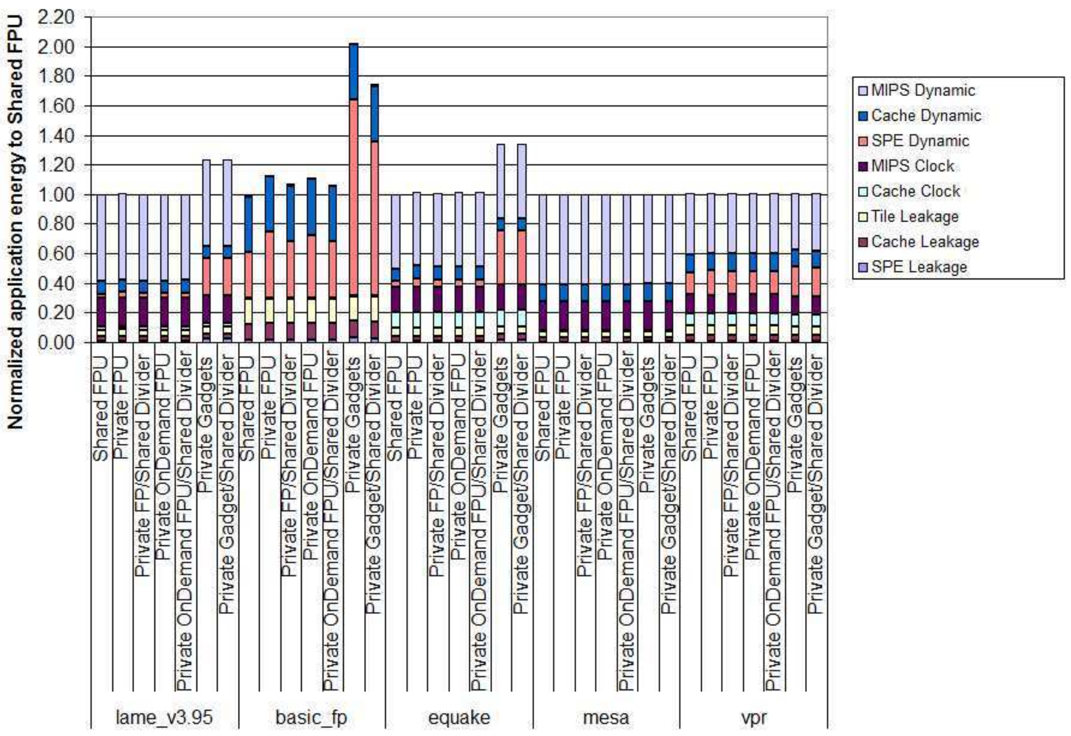


Figure 6.6: **Application energy breakdown for all five benchmarks** Each subgroup of bars represents a specific floating point hardware configuration. Results are normalized to shared FPU configuration execution time.

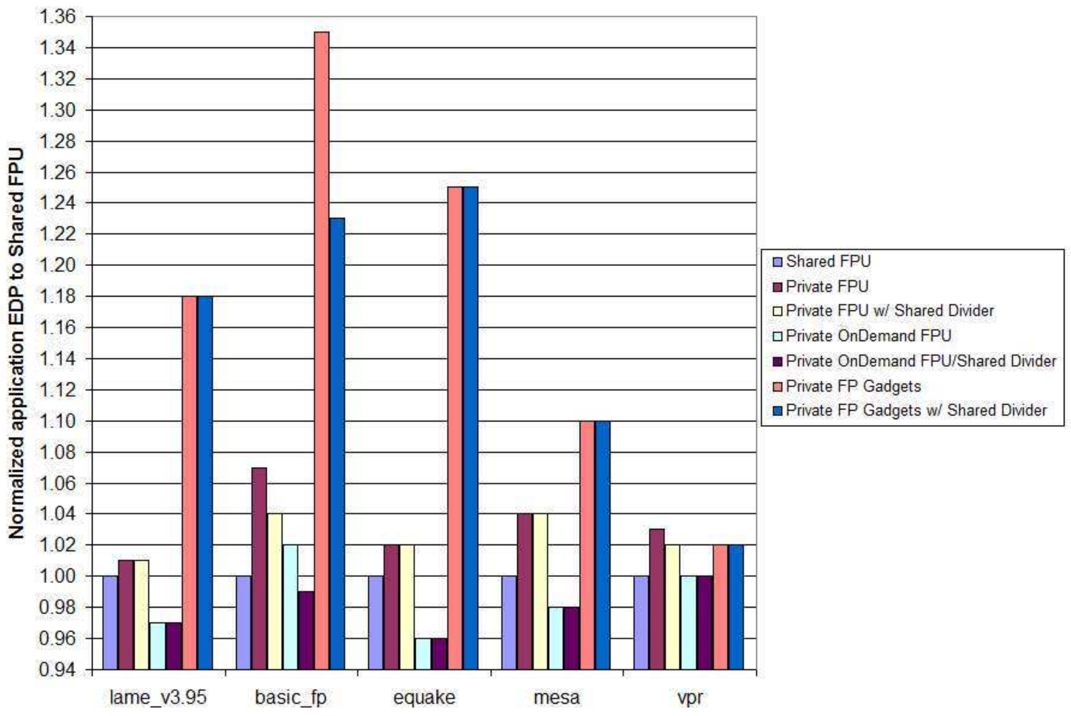


Figure 6.7: **Application energy delay product for all five benchmarks** Each subgroup of bars represents a specific floating point hardware configuration. Results are normalized to shared FPU configuration execution time.

Chapter 7

Conclusion

In this thesis, we have described the design and implementation of seven different FP hardware sharing designs between SPEs in an Arsenal system. Even more, we have explored the tradeoff between area, energy, performance and efficiency across those seven FP hardware sharing configurations. Our data for five fully placed and routed SPEs shows that sharing the FPU between the SPEs in a complex can reduce area requirements between 3.6% and 435%, energy consumption between 1% and 102% and energy delay by up to 35% while imposing minor performance degradation between 0.5% and 14% when compared to others designs. The results also show that even though, private FP hardware support configurations can improve applications performance such benefit comes at significant and, in some cases, impractical area and energy requirements.

More importantly, we show a comprehensive study of the tradeoff among area, energy, performance and efficiency of several FP hardware configurations that provide various degrees of flexibility when designing SPEs for Arsenal processors. In particular our study provided insights for the feasibility that sharing a FPU per SPEs complex in Arsenal would allow floating point intensive SPEs to be patched such that they can adapt to new versions of software that become available without significant impacts on performance.

Further improvements to our toolchain simulation capabilities, like more than one active SPE in an Arsenal complex and having concurrently active SPEs across distinct complexes, would provide additional insights into an optimal sharing scenario of a FPU across SPEs complexes such that we can reduce area and power requirements even further with minor performance degradation and wire delay effects.

Bibliography

[1] J. Babb. *"High level compilation for gate reconfigurable architectures"*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "An efficient method of computing static single assignment form". In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM.

[3] R. Dolbeau and A. Seznec. CASH: Revisiting hardware sharing in single-chip parallel processor. Research Report RR4660, INRIA, 2002.

[4] EEMBC Benchmark. http://www.eembc.org.

[5] D. Goldberg. "What every computer scientist should know about floating point arithmetic". *ACM Comput. Surv.*, 23(1):5–48, 1991.

[6] J. Hauser. SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html.

[7] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr. A methodology for the design of application specific instruction set processors (asip) using the machine description language lisa. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 625–630, Piscataway, NJ, USA, 2001. IEEE Press.

[8] IEEE Standards Board. "IEEE standard for binary floating point arithmetic", 1985.

[9] K. Karuri, R. Leupers, G. Ascheid, H. Meyr, and M. Kedia. "Design and implementation of a modular and portable IEEE 754 compliant floating point unit". In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 221–226, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[10] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. "Energy characterization of a tiled architecture processor with on-chip networks". In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 424–427, New York, NY, USA, 2003. ACM.

[11] A. J. Kleinosowski and D. J. Lilja. ”MinneSPEC: A New SPEC Benchmark Workload for Simulation Based Computer Architecture Research”. *Computer Architecture Letters*, 1(1):7, 2002.

[12] LAME MP3 Encoder. http://lame.sourceforge.net.

[13] C. Lattner and V. Adve. ”LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In *CGO ’04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[14] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. ”Tartan: evaluating spatial computation for whole program execution”. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 163–174, New York, NY, USA, 2006. ACM.

[15] D. Monniaux. ”The pitfalls of verifying floating point computations”. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[16] S. Oberman and M. Flynn. ”Design issues in division and other floating point operations”. *Computers, IEEE Transactions on*, 46(2):154–161, Feb 1997.

[17] OpenImpact Website. http://gelato.uiuc.edu/.

[18] S. Palnitkar. *”Verilog HDL: a guide to digital design and synthesis”*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. ”TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP”. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, 2004.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. ”Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems”*, pages 45–57, New York, NY, USA, 2002. ACM.

[21] P. Soderquist and M. Leeser. ”Area and performance tradeoffs in floating point divide and square root implementations”. *ACM Comput. Surv.*, 28(3):518–564, 1996.

[22] SPEC2000 Benchmark. http://www.spec.org/cpu2000.

[23] CodeSurfer by GrammaTech, Inc. http://www.grammatech.com/products/codesurfer/.

[24] S. Swanson. *”The WaveScalar Architecture”*. PhD thesis, University of Washington, Seattle, WA, USA, June 2006.

[25] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. "The Wave Scalar Architecture". *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[26] Synopsys Inc. http://www.synopsys.com.

[27] M. Taylor. *"Tiled Microprocessors"*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, February 2007.

[28] M. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim. "Evaluation of the Raw microprocessor: an exposed wire delay architecture for ILP and streams". In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2–13, June 2004.

[29] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. "Cacti 5.1". *Tech. Rep. HPL-2008-20, HP Labs, Palo Alto*, 2008.

[30] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, S. Bryksin, J. Lugo Martinez, S. Swanson, and M. Taylor. "Conservation Cores: Reducing the Energy of Mature Computations". *ASPLOS (accepted for publication)*, 2010.