THE VLIW-SUPERCISC COMPILER: EXPLOITING PARALLELISM FROM C-BASED APPLICATIONS

by

Joshua D. Fazekas

B.S. Computer Engineering, University of Pittsburgh, 2006

Submitted to the Graduate Faculty of
the School of Engineering in partial fulfillment
of the requirements for the degree of
Master of Science

University of Pittsburgh
2006

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Joshua D. Fazekas

It was defended on

April 28th, 2006

and approved by

Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

Raymond R. Hoare, Assistant Professor, Electrical and Computer Engineering Department

J.T. Cain, Professor, Electrical and Computer Engineering Department

Thesis Advisor: Alex K. Jones, Assistant Professor, Electrical and Computer Engineering Department

THE VLIW-SUPERCISC COMPILER: EXPLOITING PARALLELISM FROM C-BASED APPLICATIONS

Joshua D. Fazekas, M.S.

University of Pittsburgh, 2006

A common approach to decreasing embedded application execution time is creating a homogeneous parallel processor architecture. The parallelism of any such architecture is limited to the number of instructions that can be scheduled in the same cycle. This number of instructions scheduled in a cycle, or instruction-level parallelism (ILP), is limited by the ability to extract parallelism from the application. Other techniques attempt to improve performance with hardware acceleration. Often, segments of highly computational extensive code are extracted and custom hardware is created to replace the software execution. This technique requires many resources and still does not address the segments of code outside of the computationally extensive kernel.

To solve this problem, hardware acceleration for computationally intensive segments of code in addition to accelerating the entire application with very long instruction word, VLIW, techniques is proposed. (1) A compilation flow that targets a 4-wide VLIW processor architecture is presented. This system was used to investigate the available speed-up of VLIW architectures. The architecture was modified to combine the VLIW processor with the capability to execute application specific customized instructions. To create the custom instruction hardware, a control and data flow graph (CDFG) framework was created. The CDFG framework was created to provide a framework for compiler transformations and hardware generation. In order to remove control flow from segments of code selected for hardware generation, (2) the technique of hardware predication was developed. Hardware predication allows *if-then* and *if-then-else* control flow constructs to be transformed into strict data flow through the use of multiplexors. From the transformed CDFGs, (3) a VHDL generation pass was created that translates the compiler data structures into synthesizable VHDL. The resulting architecture contains the VLIW processor and tightly coupled application specific hardware. This architecture was analyzed for performance changes compared to the initial VLIW architecture, and a traditional processor. Lastly, (4) the architecture was analyzed for power and energy savings. A post static timing pass was added to the compilation flow for the insertion of hardware to delay early switching of operations.

By measuring only the execution of the hardware function and comparing the performance to the equivalent code executed in software, a performance multiplier of up to 322 times is seen when synthesized onto an Altera Stratix II ES2S180F1508C4 FPGA. The average performance increase seen was 63 times faster. For the entire application, the speedup reached nearly 30X and was on average 12X better than a single processor implementation. The power and energy required by the VLIW processor core and the hardware functions for the computational kernels after 160nm OKI standard cell ASIC synthesis show a maximum power savings of 417 times that of execution on the processor with an average of 133 times savings in power consumption. With the increased execution time and the savings in power the energy savings will see a multiplicative effect. The energy improvement is therefore several orders of magnitude for the hardware functions, the savings range from over 1,000X to approximately 60,000X.

TABLE OF CONTENTS

1.0 INTRODUCTION . . . 1
    1.1 Motivation . . . 3
    1.2 Key Contributions . . . 4
2.0 RELATED WORK . . . 6
3.0 VERY LONG INSTRUCTION WORD PROCESSORS . . . 10
    3.1 A VLIW/SIMD FPGA Processing Architecture . . . 10
    3.2 Compiler Flow for a VLIW/SIMD FPGA Processing Architecture . . . 16
    3.3 Results . . . 18
4.0 CONTROL AND DATA FLOW GRAPHS . . . 21
    4.1 Control Flow . . . 21
    4.2 Data Flow . . . 27
        4.2.1 The *statement* class . . . 27
        4.2.2 The *expression* class . . . 31
        4.2.3 The CDFG node class . . . 31
        4.2.4 A parsing example . . . 34
    4.3 Results . . . 35
5.0 COMPILATION FOR HARDWARE FUNCTIONS . . . 38
    5.1 Profiling . . . 41
    5.2 Preprocessing Compiler Transformations . . . 41
        5.2.1 Definition-Use Analysis . . . 44
        5.2.2 Inlining and Loop Unrolling . . . 46
        5.2.3 Code Motion . . . 49

5.3 Hardware Predication 53
5.4 Results 57
6.0 ASPECTS OF HARDWARE GENERATION 66
6.1 VHDL Generation 66
6.2 Interfacing Software and Hardware 70
6.3 Aspects of Performance Gain 72
6.4 Results 75
7.0 PERFORMANCE RESULTS 79
8.0 REDUCING POWER AND DELAY ELEMENTS 84
8.1 Power Modeling and Analysis 85
8.2 Power Impact of SuperCISC Functions 89
8.3 Delay Element and Latch Generation 92
9.0 CONCLUSIONS 97
9.1 Contributions 97
9.2 Future Research 99
BIBLIOGRAPHY 100

LIST OF TABLES

1 Performance of Instructions (Altera Stratix II FPGA EP2S180F1508C4) . . . 12
2 Performance data of 3 implementations of VLIW instruction decoder on an Altera Stratix II 15
3 The *ExecutionObject* subclasses. 29
4 The CDFG node data structures. 33
5 Hardware function synthesis results. 78
6 Power and energy required by the VLIW processor core and the SuperCISC hardware functions for computational kernels after 160nm OKI standard cell ASIC synthesis. 91

LIST OF FIGURES

1 Very Long Instruction Word Architecture. 11
2 N-element Register File Supporting P-Wide VLIW with P Read Ports and P Write Ports. 13
3 Scalability of a 32-element register file with 2 read and 1 write port per processor for a Stratix II 13
4 Scalability of an N-element register file having N read and N write ports on an Altera Stratix II. 14
5 Scalability of a 32-bit P-to-1 multiplexer on an Altera Stratix II (EP2S180F1508C4). 15
6 Compilation Flow Blue blocks represent added functionality. 17
7 VLIW ROM file format. 17
8 Cycle-by-Cycle visual representation of an FIR filter generated by Trimaran. . 19
9 Average IPL of Banchmarks. 20
10 Example showing the commands used to convert from C source to a SUIF2 file format. 22
11 Example showing the commands used to execute the SUIF control flow graph pass. 23
12 C source and generated CFG for a simple example. 24
13 CFG of basic blocks for a simple example. 25
14 Pseudo-code to call the function to create basic blocks, based on the starting nodes for each basic block. 26
15 Pseudo-code to create the basic blocks, and identify the new starting nodes for basic blocks. 28

16 Pseudo-code for parsing a SUIF statement class. 30
17 Pseudo-code for parsing a SUIF expression class. 32
18 Example data flow graph segment for the statement: $x = a + b$. 34
19 Source code of a simple example to show the CDFG pass. 35
20 Example control and data flow graph for a simple source code. 36
21 The VLIW - SuperCISC Compilation Flow 39
22 The VLIW - SuperCISC Architecture 40
23 Excerpt of the Shark profiling results for the g721 benchmark. 42
24 Code excerpt of results from Figure 23 after insertion of directives to outline computational kernels that are candidates for custom hardware implementation. 43
25 Description of the compilation and synthesis flow for portions of the code selected for custom hardware acceleration. Items on the left side are part of phase 1, which uses standard compiler transformations to prepare the code for synthesis. Items on the right side manipulate the code further using hardware predication to create a DFG for hardware implementation. 45
26 Fmult function from G.721 benchmark. 47
27 G.721 code after function inlining. 48
28 Unrolled inner loop of inlined G.721 hardware kernel. 49
29 G.721 benchmark after inlining, unrolling, and code motion compiler transformations. 50
30 G.721 benchmark example of a static array. 52
31 Software code, CDFG, and DFG with predicated hardware example for control flow in ADPCM encoder. 55
32 Pseudo-code for parsing a SUIF expression class. 57
33 Pseudo-code for the predicating an *if-then* control flow structure. 58
34 Source Code for the G.721 Benchmark Kernel. 60
35 CDFG for the *predictor_zero* Function of the G.721 Benchmark. 61
36 CDFG for the *fmult* Function of the G.721 Benchmark. 62
37 CDFG for the *quan* Function of the G.721 Benchmark. 63
38 CDFG for the Entire G.721 benchmark after Compiler Transformations. . . . 64

39 SuperCISC compilation flow - Blue blocks represent added functionality. . . . 65

40 Pseudo-code for top level hardware generation. 67

41 Pseudo-code for Entity Declaration Parsing. 68

42 Pseudo-code for Architecture Declaration Parsing. 69

43 Algorithm execution profile including potential overheads. In the VLIW processor with hardware functions, the hardware functions require no hardware specific setup overhead. 71

44 Software code and data flow graph (DFG) showing control flow in ADPCM encoder. 72

45 Latency for software equivalent of 2:1 multiplexer and 16-entry priority encoder relative to hardware implementation. 73

46 Cycle time utilization of hardware function arithmetic nodes as normalized to the cycle time of a 167 MHz processor on an Altera EP2S180. 75

47 Simple Source Code Marked for Hardware Creation. 76

48 CDFG for the Simple Source Code from Figure 47. 76

49 Generated VHDL for the Simple Source Code from Figure 47. 77

50 Performance improvement from hardware acceleration of computational portion of the hardware kernel. 81

51 Kernel speedups several architectures when considering required loads and stores to maintain the register file. 82

52 Overall performance speedup of the entire application for several architectures including overheads associated with function calls. 83

53 4-D plots of p, d, and s versus power for several ALU operations synthesized with standard cells for a 160nm OKI ASIC process. Power is indicated as a color between red and blue where solid blue represents the least power consumed by the device and red indicates the most power consumed by the device. Measurements taken at 0.1 intervals in each dimension p, d, and s. 86

54 Power profile of several different functional unit implementation techniques. Results for a standard cell 160nm OKI ASIC process. 88

55 Power compression of various classes of ALU operations. 90

56 Power reduction shown with performance improvement for benchmark suite. . 91
57 Energy improvement of SuperCISC over VLIW for benchmark suite. 92
58 Proposed Delay Element. 93
59 Proposed Delay Element's Delay Function. 94
60 Proposed Delay Element's Power and Energy Cost. 95

1.0 INTRODUCTION

As demand for processing power increases, and fabircation technologies continue to increase transistor density, it is important to develop architectures that can leverage these transistor densities and provide increasingly high performance capability while satisfying progressively stricter power budgets.

Today, most current processor improvements revolve around increasing the size of the processing device. For example, a SIMD, Single Instruction Multiple Data, machine takes a single prcoessing element and replicates the physical device many times. Then, the same instruction is executed on every logic unit. The operations remain independant by scheduling each processing unit to operate on different sets of data. Thus, complex data dependant operations can be completed with relative ease. This technique has been around for a long time and is used almost solely in vector processing calculations. Another technique, MIMD or Multiple Instruction Multiple Data, also takes the same core processing element and replicates the logic units. In this technique each of the instructions can be operated on independently. A MIMD machine behaves essentially like a multiprocessor system on a chip. The difficulty in creating these types of processor lies in the network connecting the logic units, and the ability to schedule instuctions that are independant of each other for every processor.

The problem with SIMD and MIMD solutions is that up until now all of the thinking and development for the past forty years has been on software that will execute sequentially. Several large scale projects, such as the Intel Itanium 2 VLIW processor, have failed in the market due to the lack of ability to leverage the hardware resources and the increased cost of the processor. The compiler used to support the Itanium processor was not ready for release when the processor debuted and offered a poor performance increase over standard execution.

In addition, software developers were required to release separate binaries to utilize the additional processing hardware. In order to circumvent this problem, another company, Transmeta, created a hardware wrapper that translates standard executable binaries into VLIW instructions on the fly. The concept relies greatly on the ability for the hardware to essentially recompile the code at runtime. The cost of the hardware translation increases the overall system cost, but Transmeta feels it is required to support legacy code.

Both of these projects, and many more attempt to utilize available transistors to increase performance. Another use for these available transistor that is being developed is an attempt to create custom hardware based on specific applications. This is not a new concept, some of the earliest main stream processors like the Intel 8086, for example, contained the ability to send data off the chip to a co-processor. The Intel 8087 was later developed to provide additional processing power. The 8087 added the capability to perform complex math operations operations, and floating-point operations up to fifty times greater than execution on the processor.

The concept of creating architectures with the ability to contain reconfigurable co-processing units is being heavily researched. In the embedded market, processors are commonly developed for one or two applications. In order to meet timing constraints, some algorithms must be implemented entirely in hardware. The cost of custom hardware is much greater than the cost of executing an application on a processor. To counter this cost, hybrid systems that allow embedded processors to execute application specific hardware are being developed.

The subject of this thesis is a compilation and design automation flow for algorithms written in 'C'. Presented within are an architecture and a design flow that allow for the creation of application-specific hardware accelerated processors. Initially, investigations into the utilization of field programmable gate array, FGPA, resources led to the creation of large very long instruction word, VLIW, processors. After a design flow was created for the VLIW architecture it was found that many of the available resources were still not utilized. In order to increase the resource utilization, application-specific instructions were added to the processors. These custom instructions are mapped into available resources on the target FPGA. A compilation flow was created to automatically generate the hardware needed for these custom instructions. The architecture was modified to contain a 4-wide VLIW reduced instruction set processor with the ability to execute super complex instructions. The encouraging performance results led to researching the use of standard cell application-specific integrated circuits (ASICs) as a target technology.

1.1 MOTIVATION

Computer architects are always looking at different ways to improve overall processor speed. The current trend is to increase cache size, and add multiple processor cores to a single chip. One of the main reasons for this trend is that the current VLSI fabrication process has made it possible to place billions of transistor on a chip, however, current design tools limit the architects ability to use all of the available gates. In addition, as transistors scale to the tens of nanometers, effects such as clock skew and leakage power are becoming significant hurdles in processor design. Multi-core and large cache architectures allow designers the ability to create a single small design, or reuse an existing design, that contains only a small percentage of the total available gates on the chip. These designs can then be replicated many times on the chip. By treating each of the cores as a separate entities, with separate clock domains, the problem of clock skew becomes much more manageable.

There is no single correct solution to how the additional available resources should be used. Many proposals fail to make the mainstream due to lack to popular support. However, with the increasing size of the embedded computing market and the increasing use of custom processors, the need for high-level synthesis tools is increasing. The approach presented leverages the popular use of FPGA and ASIC targets for speeding up the critical areas of execution. It offers the designer the ability to specify the algorithm on the application level in C/C++ code with the ability to produce a system shown to exceed the performance of a standard VLIW processor.

1.2 KEY CONTRIBUTIONS

To accelerate the sections of code executed in software a VLIW processor was created. The processor was designed to behave like the reduced instruction set Altera NIOS II processor. A single processor system was also developed along side this VLIW processor to benchmark the performance increase. In order to exercise the architectures, a compilation flow for the VLIW processor was developed. The system architect was consulted to extend the NIOS II instruction set for VLIW execution. When tested the VLIW architecture provided less than a 2X increase in performance on average.

In order to create automatically generated hardware functions, an intermediate representation, IR, friendly to the needed compiler transformations was required. A control and data flow graph (CDFG) framework provides both the combinational segments of execution, or data flow, and the control boundaries. The CDFG IR was coded within the SUIF2 compiler. After the front-end of the SUIF2 compiler completes, a built in control-flow pass is executed. From the sequential statements provided by the control flow, data flow graph are created. The IR is currently being used for two other academic projects, in addition to the compiler transformations for hardware generation.

Data flow can be implemented in hardware very efficiently. On the other hand, control flow is suited very well for processor execution. To obtain a highly efficient architecture, the data flow of computational kernels is accelerated in application specific hardware. The looping and control flow structures are executed in the processor. To increase the amount of kernel code executing in hardware, a technique called predication was used to convert control flow to data flow. Hardware predication takes *if-then* and *if-then-else* control flow constructs and transforms them into data flow through the use of multiplexors. Other techniques, such as function inlining, and loop unrolling were also explored to eliminate control flow.

After a large section of data flow has been created for the computational kernel the last step is to translate the IR to a hardware description language. This translation is built into the compiler as a pass. This VHDL generation pass translates the compiler data structures into synthesizable VHDL. The VLIW architecture was modified to execute the generated hardware functions as a custom instruction in the processor. Data between the processor and hardware was shared through the processor's register file. The resulting architecture contains the VLIW processor and tightly coupled application specific hardware. The entire architecture was synthesized and tested for performance. The computationally intensive software kernels, which are typically called by the application many times, show an average increase in performance of 63X. Some benchmark kernels see improvements of over 300X. The entire application execution was also measured. The hardware accelerated VLIW processor showed performance speedups of 12X on average over a single processor implementation. The highest application speedup reached nearly 30 times the execution of the single processor setup.

The power required for processor execution is generally much higher than custom hardware for the same technology. This is contributed to the processor overhead of fetching, decoding, the ability to execute many operations, and write back. The processor was synthesized in a 160nm ASIC process and the benchmark kernels were analyzed for power. The result was compared to the same kernel's power for the generated hardware synthesized in the same 160nm ASIC process. The power requirements were compared, and the required energy was calculated based on the timing information from the ASIC synthesis. For the computational kernels the power improvement of the hardware execution is between just under 50X and over 400X with an average of just over 130X improvement. The energy improvement seen by the hardware execution combines the power efficiency and the decrease latency and therefore is several orders of magnitude lower for the hardware functions. The energy savings range from over 1,000X in the worst case to approximately 60,000X.

Chapter 2 describes the work related to the architecture and compilation design flow. The architecture is described in detail in Chapter 3. The intermediate representation created by the front-end of the compiler is detailed in Chapter 4. Chapter 5 discusses the transformations to the intermediate representation used by the compiler to produce a combinational data flow. Chapter 6 illustrates the conversion of the transformed intermediate code representation into a hardware description. The performance results of the architecture are presented in Chapter 7. Energy and power saving results are presented in Chapter 8. Finally, Chapter 9 describes the conclusion of the work and proposes areas suitable for future work.

2.0 RELATED WORK

Manual hardware acceleration has been applied to countless algorithms and is beyond enumeration here. These systems generally achieve significant speedups over their software counterparts. Behavioral and high-level synthesis techniques attempt to leverage hardware performance from different levels of behavioral algorithmic descriptions. These different representations can be from hardware description languages (HDLs) or software languages such as C, C++, Java, and Matlab.

The HardwareC language is a C-like HDL used by the Olympus synthesis system at Stanford [1]. This system uses high-level synthesis to translate algorithms written in HardwareC into standard cell ASIC netlists. Esterel-C is a system-level synthesis language that combines C with the Esterel language for specifying concurrency, waiting, and pre-emption developed at Cadence Berkeley Laboratories [2]. The SPARK synthesis engine from the UC Irvine translates algorithms written in C into hardware descriptions emphasizing extraction of parallelism in the synthesis flow [3] [4]. The PACT behavioral synthesis tool from Northwestern University translates algorithms written in C into synthesizable hardware descriptions that are optimized for low-power as well as performance [5] [6].

In industry, several tools exist which are based on behavioral synthesis. The Behavioral Compiler from Synopsys translates applications written in SystemC into netlists targeting standard cell ASIC implementations [7] [8]. SystemC is a set of libraries designed to provide HDL-like functionality within the C++ language for system level synthesis [9]. Synopsys cancelled its Behavioral Compiler because customers were unwilling to accept reduced quality of results compared to traditional RTL synthesis [10]. Forte Design Systems has developed the Cynthesizer behavioral synthesis tool that translates hardware independent algorithm descriptions in C and C++ into synthesizable hardware descriptions [11]. Handel-

C is a C-like design language from Celoxica for system level synthesis and hardware software co-design [12]. Accelchip provides the AccelFPGA product, which translates Matlab programs into synthesizable VHDL for synthesis on FPGAs [13]. This technology is based on the MATCH project at Northwestern [14]. Catapult C from Mentor Graphics Corporation translates a subset of untimed C++ directly into hardware [15].

The difference between these projects and the technique presented is that they try to solve the entire behavioral synthesis problem. The presented approach utilizes a 4-wide VLIW processor to execute nonkernel portions of the code (10% of the execution time) and utilizes tightly coupled hardware acceleration using behavioral synthesis of kernel portions of the code (90% of the execution time). The available hardware resources were matched to the impact on the application performance so that the processor core utilizes 10% or less of the hardware resources leaving 90% or more to improve the performance of the kernels.

The synthesis flow presented utilizes a DFG representation that includes *hardware predication*: a technique to convert control flow based on conditionals into multiplexer units that select from two inputs fromthis conditional. This technique is similar to assignment decision diagram (ADD) representation [16] [17], a technique to represent functional register transfer level (RTL) circuits as an alternative to control and data flow graphs (CDFGs). ADDs read from a set of primary inputs (generally registers) and compute a set of logic functions. A conditional called an *assignment decision* then selects an appropriate output for storage into internal storage elements. ADDs are most commonly used for automated generation of test patterns for circuit verification [18] [19]. The technique presented is not limited to decisions saved to internal storage, which imply sequential circuits. Rather, the new technique applies *hardware predication* at several levels within a *combinational* (i.e., DFG) representation.

The support of custominstructions for interface with coprocessor arrays and CPU peripherals has developed into a standard feature of soft-core processors and those which are designed for DSP and multimedia applications. Coprocessor arrays have been studied for their impact on speech coders [20] [21], video encoders [22] [23], and general vector-based signal processing [24] [25] [26].

These coprocessor systems often assume the presence and interface to a general-purpose processor such as a bus. Additionally, processors that support custom instructions for interface to coprocessor arrays are often soft-core and run a significantly slower clock rates than hard-core processors. The VLIW-SuperCISC processor is fully deployed on an FPGA system with detailed post place-and-route performance characterization. Our processor does not have the performance bottleneck associated with a bus interconnect but directly connects the hardware unit to the register file. There is no additional overhead associated with calling a hardware function.

Several projects have experimented with reconfigurable functional units for hardware acceleration. PipeRench [27] [28] [29] [30] [31] and more recently HASTE [32] have explored implementing computational kernels on coarse-grained reconfigurable fabrics for hardware acceleration. PipeRench utilizes a pipeline of subword ALUs that are combined to form 32-bit operations. The limitation of this approach is the requirement of pipelining as more complex operations require multiple stages and, thus, incur latency. In contrast, by using non-clocked hardware functions that represent numerous 32-bit operations. RaPid [33] [34] [35] [36] [37] is a coarse-grain reconfigurable datapath for hardware acceleration. RaPid is a datapath-based approach and also requires pipelining. Matrix [38] is a coarse-grained architecture with an FPGA like interconnect. Most FPGAs offer this coarse-grain support with embedded multipliers/adders. The presented approach, in contrast, reduces the execution latency and, thus, increases the throughput of computational kernels.

Several projects have attempted to combine a reconfigurable functional unit with a processor. The Imagine processor [39] [40] [41] combines a very wide SIMD/VLIWprocessor engine with a host processor. Unfortunately, it is difficult to achieve efficient parallelism through high ILP due to many types of dependencies. The presented processor architecture differs as it uses a flexible combinational hardware flow for kernel acceleration.

The Garp processor [42] [43] [44] combines a custom reconfigurable hardware block with a MIPS processor. In Garp, the hardware unit has a special purpose connection to the processor and direct access to the memory. The Chimaera processor [45] [46] combines a reconfigurable functional unit with a register file with a limited number of read and write ports. The new system differs as it uses a VLIW processor instead of a single processor and the hardware unit connects directly to all registers in the register file for both reading and writing allowing hardware execution with no overhead. These projects also assume that the hardware resource must be reconfigured to execute a hardware-accelerated kernel, which may require significant overhead. In contrast, the new system configures the hardware blocks prior to runtime and uses multiplexers to select between them at runtime. Additionally, the new system is physically implemented in a single FPGA device, while it appears that Garp and Chimaera were studied in simulation only.

In previous work, a 64-way and an 88-way SIMD architecture was created and interconnected the processing elements (i.e., the ALUs) using a hypercube network [47]. This architecture was shown to have a modest degradation in performance as the number of processors scaled from 2 to 88. The instruction broadcasting and the communication routing delay were the only components that degraded the scalability of the architecture. The ALUs were built using embedded ASIC multiply-add circuits and were extended to include user-definable instructions that were implemented in FPGA gates. However, one limitation of a SIMD architecture is the requirement for regular instructions that can be executed in parallel, which is not the case for many signal processing applications. Additionally, explicit communications operations are necessary.

Work by industry researchers [48] shows that coupling a VLIW with a reconfigurable resource offers the robustness of a parallel, general-purpose processor with the accelerating power and flexibility of a reprogrammable systolic grid. For purposes of extrapolation, the cited research assumes the reconfiguration penalty of the grid to be zero and that design automation tools tackle the problem of reconfiguration. The presented system differs because the FPGA resource can be programmed prior to execution, giving us a more realistic reconfiguration penalty of zero. A compiler and automation flow to map kernels onto the reconfigurable device is also provided [49].

3.0 VERY LONG INSTRUCTION WORD PROCESSORS

VLIW (Very Long Instruction Word) processors gained their name from the fact that each instruction word is used to program multiple execution units and therefore requires many bits. By definition, a VLIW processor is able to independently execute on two or more instructions simultaneously. The execution units in VLIW processors have historically been based on the design of a simple processor. These processors are then replicated, typically four to eight times, and placed onto a single chip, essentially creating a multiprocessor system on a chip. In contrast to a traditional superscalar processor in which logic is added to the processor to execute more complex instructions, a VLIW processor uses its multiple execution units to decompose complex instructions and execute the simple instructions in parallel.

Section 3.1 details the design of a VLIW processor's architecture developed by another student, and a mapping on a VLIW processor onto a FPGA (Field Programmable Gate Array). Section 3.2 describes the compiler flow for the VLIW architecture. The results of the VLIW processor can be found in section 3.3.

3.1 A VLIW/SIMD FPGA PROCESSING ARCHITECTURE

In order to examine the scalability of a VLIW architecture for FPGAs, a VLIW architecture was developed and synthesized. The architecture created is shown in Figure 1, the wider instruction stream and the shared register file scale with the size of the VLIW processor. The ALUs (also called processing elements PEs) are identical to that of a single processor VLIW or a stand-alone processor. Rather than having a single instruction executed each

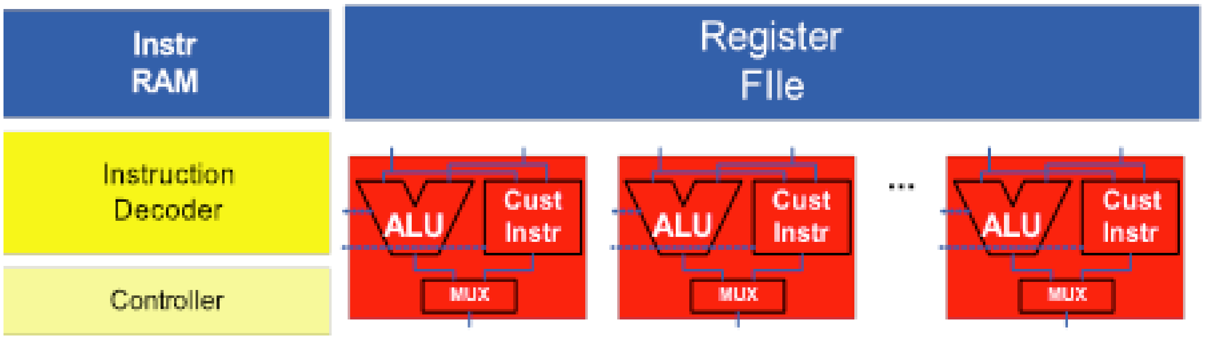


Figure 1: Very Long Instruction Word Architecture.

clock cycle, a VLIW can execute P operations for a P processor VLIW. In this section, the architecture developed in primary by Dara Kusic is detailed, the scalability of each of its components is described.

The VLIW processor array consists of 32-bit processors with a shared register file. The processors are identical in architecture and fully supportive of the RISC instruction set of the NIOS II soft-core processor. Exploitation of the available width of the VLIW architecture relies on the VLIW compiler to concurrently schedule data-independent sequential operations. The system implements a five-stage instruction cycle consisting of instruction fetch, operand fetch, two execute cycles and a writeback stage.

The processing element supports four types of 2-operand arithmetic operations: add/subtract, multiply, logic operations, and shift. The latency of each operation is listed in Table 1. ALU functionality can be augmented for digital signal processing and multi-media applications by implementing custom instructions through utilization of an expanded opcode and digital signal processing blocks for multiply-accumulate operations.

Processor operations are grouped into L operations (multiply, add/subtract, logical, shift, etc). Within each ALU, each of the L operations executes concurrently using separate function units. A multiplexer then selects between the L 32-bit function unit outputs and feeds the result back into the register file. It was noted that there is a wide range of performance for the individual function units and that the size of L has an impact on the

Table 1: Performance of Instructions (Altera Stratix II FPGA EP2S180F1508C4)

ALU Module	Clock	Latency
Adder/Subtractor/Comparator	156.62 MHz	6.38 ns
2-Operand Mulitplier	108.52 MHz	9.21 ns
Logical Unit (AND/OR/XOR)	108.52 MHz	2.37 ns
Variable Left/Right Shifter	216.68 MHz	4.61 ns
Top ALU (4 Modules Above)	102.07 MHz	9.80 ns

overall ALU performance. By grouping the function units together, the size of the multiplexer can be reduced down to a 4-to-1 at a loss of only 6 MHz.

The gains offered by a VLIW supporting a large instruction set come at a price to the performance and area of the design. The number of ports to the shared register file, Figure 2 and instruction decode logic have shown to be the greatest limitations to VLIW scalability.

Multiplexing breadth and width pose the greatest hindrances to clock speed in a VLIW architecture. The effect of multiplexers was studied by charting the performance impact of increasing the number of ports on a shared register file, an expression of increasing VLIW width.

For the P-processor VLIW with an N-element register file, the multiplexers function as routers, as shown in Figure 2. For write operations, each of the N registers requires a 32-bit P-to-1 multiplexer as data can come from any processor. For read operations, each of the P ports requires a 32-bit N-to-1 multiplexer as each processor can read from any of the N registers.

In Figure 3, the number of 32-bit registers is fixed to 32 and the number of processors is scaled. For each processor, two operands need to be read and one written per cycle. Thus, for P processors there are 2 P read ports and P write ports. The register file lost an average of 16 MHz and gained an average of 2x area utilization per processor doubling. At 16 processors, 16% of the FPGA is utilized and the clock frequency is 79 MHz.

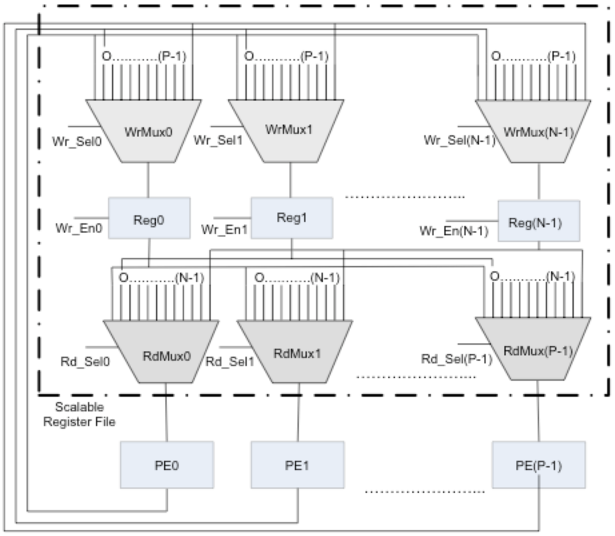


Figure 2: N-element Register File Supporting P-Wide VLIW with P Read Ports and P Write Ports.

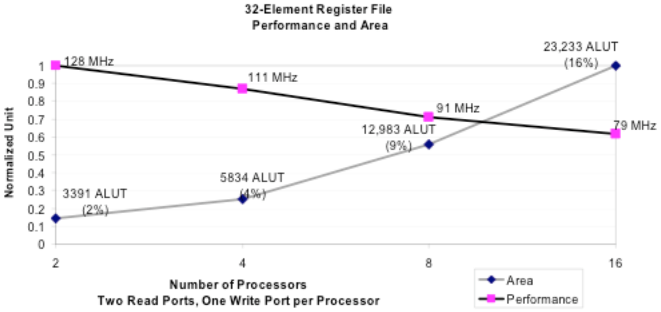


Figure 3: Scalability of a 32-element register file with 2 read and 1 write port per processor for a Stratix II .

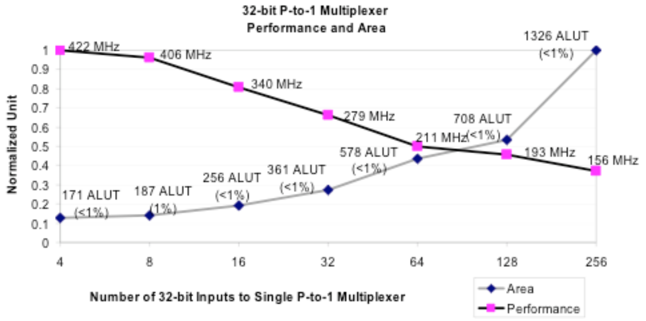


Figure 4: Scalability of an N-element register file having N read and N write ports on an Altera Stratix II.

Figure 4 shows results of scaling both the number of read/write ports and the number of 32-bit registers. In this case, the register file lost an average of 39 MHz and gained an average of 3x area utilization per doubling of the PEs, read/write ports and register file elements. An *ALUT* contains one register and combinational logic; a Stratix II EP2S180 contains 143,520 ALUTs.)

The multiplexer is the most elemental design unit contributing to performance degradation of the register file as the VLIW scales. The next step was to measure the impact of a single 32-bit P-to-1 multiplexer on the Stratix II EP2S180. As the width, P, doubled, the area increased by a factor of 1.4x. The performance took the greatest hit of all the scaling tests, losing an average of 44 MHz per doubling. This can be seen in Figure 5. The performance detriment naturally furthers as the number of P-to-1 multiplexers multiplies to serve as read and write ports to the register file.

The multiplexed data routing system of the register file was replaced by a crossbar architecture resembling a 2-D switched array. The crossbar architecture offered no clear advantage over the multiplexed architecture in terms of either area or performance. The decision was made to retain the multiplexed routing system in favor of its scalability with respect to the VHDL design process.

The second most significant performance bottleneck is the instruction decoder that translates opcodes into control signals. The decoder can be implemented as either combinational

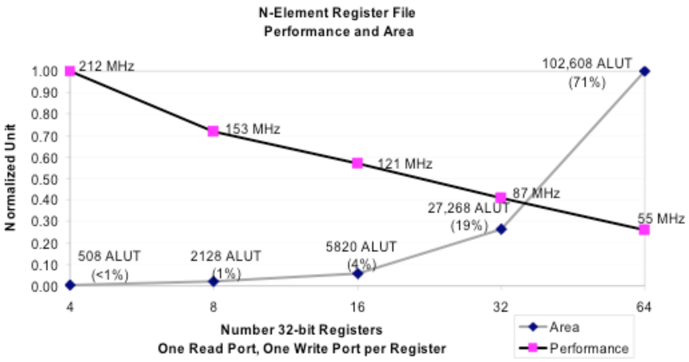


Figure 5: Scalability of a 32-bit P-to-1 multiplexer on an Altera Stratix II (EP2S180F1508C4).

logic, as a ROM, or as a RAM. Table 2 shows the resulting area and performance for an Altera NIOS II decoder that was created. This section was accepted for publication, for further review see [50].

Table 2: Performance data of 3 implementations of VLIW instruction decoder on an Altera Stratix II

Decoder Type	ALUTs	%Area	Clock	Latency
Decoder - ROM	174	< 1	344.23 MHz	2.90 ns
Decoder - RAM (11-bit address)	6630	4	124.16 MHz	8.05 ns
Decoder - Combination Logic	1,192	< 1	108.85 MHz	9.19 ns

3.2 COMPILER FLOW FOR A VLIW/SIMD FPGA PROCESSING ARCHITECTURE

To execute software code on the architecture described in section 3.1, a compilation flow was developed. The input to the flow is C code. The code is read into the compiler and scheduled into instructions that can be performed in parallel. The final result is machine code that can be executed on the VLIW FPGA processor.

Trimaran [51] is an open source VLIW compiler based on the IMPACT [52] front-end and the ELCOR [53] back-end. Trimaran uses a machine description language that allows the used to specify changes in the target architecture. The ELCOR back-end was developed to target the Hewlett-Packard HPL-PD family of processors. Third-party back-ends exist to target other architectures. Triceps [54] is a Trimaran back-end to target the StrongARM processor, and Tritanium [55] is available to target the Intel Itanium series of processors.

Trimaran was selected as a framework for the VLIW FPGA compiler. Trimaran was chosen because it is an open source, extensible VLIW compiler. It contains separate front-end and back-end code and code that translates the intermediate representation, IR, between the two. Trimaran was also chosen because the projects mentioned above serve as examples of how to modify the code generation for different processor targets.

The complete compilation flow can be seen in Figure 6 below. Only the Trimaran back-end was modified, the front-end, IMPACT, remained unchanged. The back-end was changed to target the VLIW NIOS II processor from the IMPACT IR. Much of the existing Trimaran code was reused to accelerate development time. An assembler to translate the assembly code into a ROM file was primarily written by Ahmed Muaydh. The assembler generates very long instruction words, and therefore must add idle operations for processors when no instruction is scheduled in the assembly. Figure 7 shows an example of four instructions and their instruction word.

The NIOS II back-end is closely coupled to code from the ELCOR back-end. Only changes that were required for the VLIW NIOS II machine were made. The back-end is responsible for instruction scheduling, register allocation, and code generation. The ELCOR back-end was capable of scheduling several instruction that were not supported by the RISC

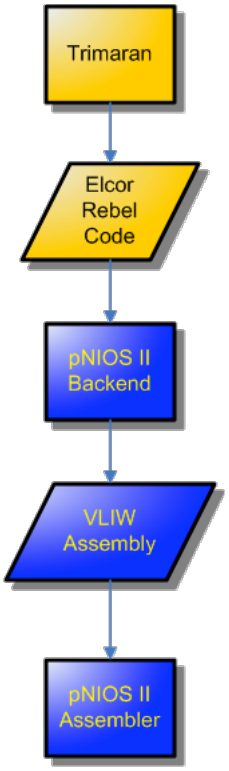


Figure 6: Compilation Flow Blue blocks represent added functionality.

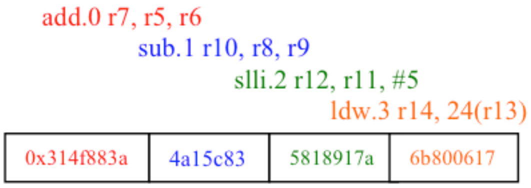


Figure 7: VLIW ROM file format.

NIOS II architecture. In this case, these instructions were decomposed into multiple NIOS II instructions. To adjust the scheduling instructions that were decomposed were given a higher instruction latency in the Trimaran machine description. Another change was the removal of support for predication. The VLIW NIOS II architecture does not support predication thus the ability to use this hardware feature was removed. Another example of the changes to ELCOR was the adaption of a floating-point module from the Triceps project to allow floating-point execution to be modified into a fixed-point implementation.

In Figure 7 there is an example of generated assembly code. Each Nios II instruction has been appended with an ID number corresponding to the processing element it has been scheduled on. In the example, these four instructions have been scheduled for the same cycle because they occur in order, from the lowest ID to the highest ID. The assembler translates each instruction to its binary format from the NIOS II specification. The instructions scheduled for the same cycle are then concatenated and combined with the FPGA ROM format to produce a file that can be read into the FPGA.

3.3 RESULTS

The Trimaran compiler was used to investigate the Instruction Level Parallelism (ILP) of several benchmarks. The compiler generates scheduled VLIW assembly code. From the assembly code, the frequency how many of the processing elements in the VLIW processor are being used every cycle can be inspected. Trimaran is able to output the IPL from the source code.

Figure 8 is an example of a cycle-by-cycle view within Trimaran. A yellow block indicates the start of a basic block; a segment of code that contains only sequential execution from start to finish, or in other words, contains no looping or branching. Within each basic block, a row represents a clock cycle. Each of the blocks to the right side of the clock cycle indicate instructions that are scheduled for execution in that clock cycle. For example, in clock cycle 0 of basic block 4, there are eight instructions scheduled for execution. In the next cycle, there are only three instructions, and in the third cycle zero instructions are scheduled.

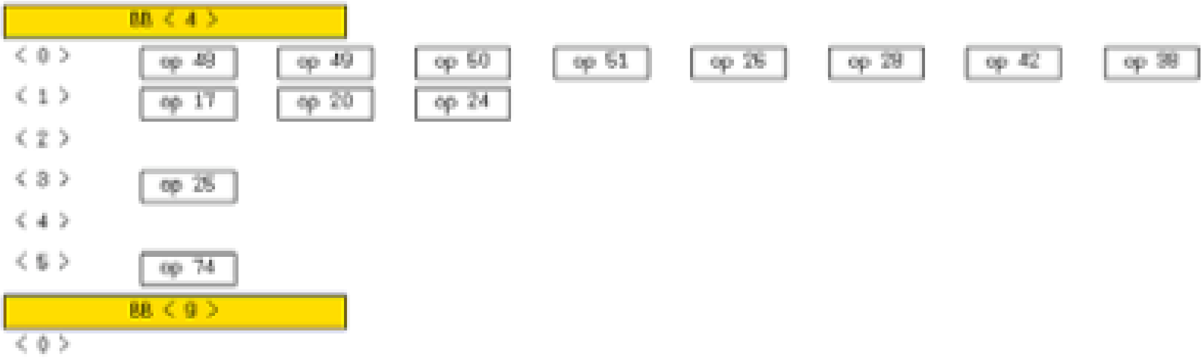


Figure 8: Cycle-by-Cycle visual representation of an FIR filter generated by Trimaran.

Trimaran was designed to target different Hewlett-Packard processors, and thus gives the user the ability to change many of the target processor's features. Trimaran uses a machine description to allow the user to specify register file sizes, number of functional units, individual instruction latency, and more. Changes to these parameters in the machine description allowed the tradeoffs in section 3.1 to be evaluated.

The benchmarks, ADPCM encoder and decoder, MPEG2 encoder and decoder, and an FIR filter were chosen to be examined to detect their ILP by Trimaran. The ADPCM benchmark was choose because of its control oriented execution, the MPEG2 benchmark was chosen because of its data oriented execution, and the FIR filter was chosen as a combination of both control and data flow.

ILP results for two different types of processors were generated. The first architecture was a 4 processing element VLIW with 2 memory ports. The results were compared to an architecture with 100 processing elements and memory ports. The results, seen in Figure 9, show that the average ILP for both architectures is less then 2. Therefore, VLIW architectures with many processing elements will have a low overall utilization, leaving most of the processing elements idle.

Upon examining the cycle-by-cycle output, Figure 8, it can be seen that there are cycles where many processing elements are idle. Both architectures that were studied were given a 2 cycle access time to main memory. In cycle 2 of Figure 8, all of the processors are idle. This

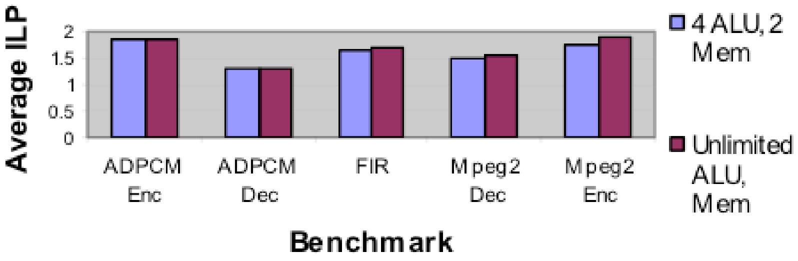


Figure 9: Average IPL of Banchmarks.

is because of a memory overhead. In the following cycle, 3, there is only a single processing element being used. Typically, this means that the instruction that is executing in cycle 3 had a data dependance from one of the instructions in cycle 1. These Data dependancies are one of the chief restrictions of ILP, but there are also several other factors that can limit it.

The scaling tradeoffs presented in Section 3.1 and the ILP limitations presented above indicate that replicating processing elements in a VLIW architecture has very strict limitations. The average ILP of the benchmarks profiled was 2. Therefore, even though the capability of placing up to 64 processing elements in the VLIW architecture within the target Altera Stratix II FPGA is available, many of the resources would never be used. The above results were accepted for publication [50] and provided the initial groundwork for the research described in the following chapters.

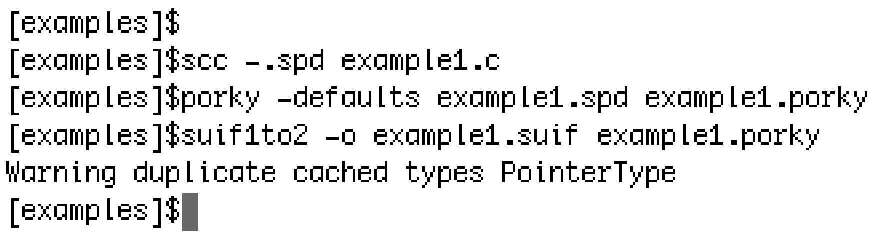
4.0 CONTROL AND DATA FLOW GRAPHS

This chapter provides details of control flow graph, CFG, creation, data flow graph, DFG, creation, and control and data flow graph, CDFG, creation. A control flow package is included as part of the SUIF2 [56] compiler framework's package. Section 4.1 describes the use of the existing control flow package and the extensions added for the purpose of CDFG creation. Section 4.2 describes the SUIF extension that was written to create DFGs. The final goal was to produce a CDFG framework that provides a interface to explore automatic hardware generation. The results of the CDFG generation are found in section 4.3 below. This work is based on the creation of similar graphs used for hardware synthesis; details can be found in [1].

4.1 CONTROL FLOW

The SUIF2 compiler framework was created to support collaborative research in optimizing and parallelizing compilers. Initially, the project included the ability to convert C and C++ files into and out of the SUIF file format, and the ability to do some simple code transformations. Since it was first released many academic and industry projects have created extensions to the framework, which SUIF defines as passes. Since the initial release, the SUIF group has released an add-on package that includes a dead code solver, a data flow solver, and a control flow graph builder.

To build a control flow graph using the extra SUIF package is trivial once the package has been installed. First, the source code is converted into the SUIF2 format. Means for the conversion are provided by the compiler. Figure 10 shows the commands required for



```
[examples]$
[examples]$scc -.spd example1.c
[examples]$porky -defaults example1.spd example1.porky
[examples]$suif1to2 -o example1.suif example1.porky
Warning duplicate cached types PointerType
[examples]$
```

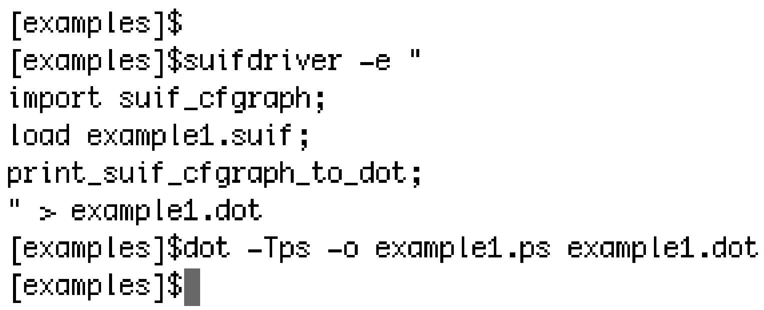
Figure 10: Example showing the commands used to convert from C source to a SUIF2 file format.

converting a C source file into the SUIF2 format. This method uses the SUIF1 front-end and converts the SUIF1 AST into SUIF2 format.

The scc command in Figure 10 calls the SUIF1 front-end and creates a SUIF1 AST from the C source. According to the man page, the porky command is "to be used right after the front end, to turn some non-standard SUIF that the front end produces into standard SUIF. It also does some things, like constant folding and removing empty symbol tables, to make the code as simple as possible without losing information." The final command, suif1to2, converts a SUIF1 AST to a SUIF2 AST.

The SUIF1 front-end is used because the SUIF2 front-end is closed source and is copyright The Portland Group Inc. and Edison Design Group Inc. The PGI front-end does not preserve compiler directives, such as *#pragma*, during the conversion to the SUIF2 AST. Later in section 5.1 these compiler directives will be used.

Figure 11 shows an example of the commands used to create a control flow graph. The first step is to execute the SUIF2 framework binary, suifdriver. Next, the control flow pass is loaded into memory, followed by the SUIF source file. The control flow pass can now be run. The pass will annotate each SUIF source code line with a unique ID number to identify its location in the control flow graph. The pass will also annotate the top level procedure structure with a list of all of the control graph nodes and a list of successors. Each control flow graph node data structure contains a list of its parents, a list of its successors, and a pointer to the SUIF data structure containing the translated source code.



```
[examples]$
[examples]$suifdriver -e "
import suif_cfgraph;
load example1.suif;
print_suif_cfgraph_to_dot;
" > example1.dot
[examples]$dot -Tps -o example1.ps example1.dot
[examples]$
```

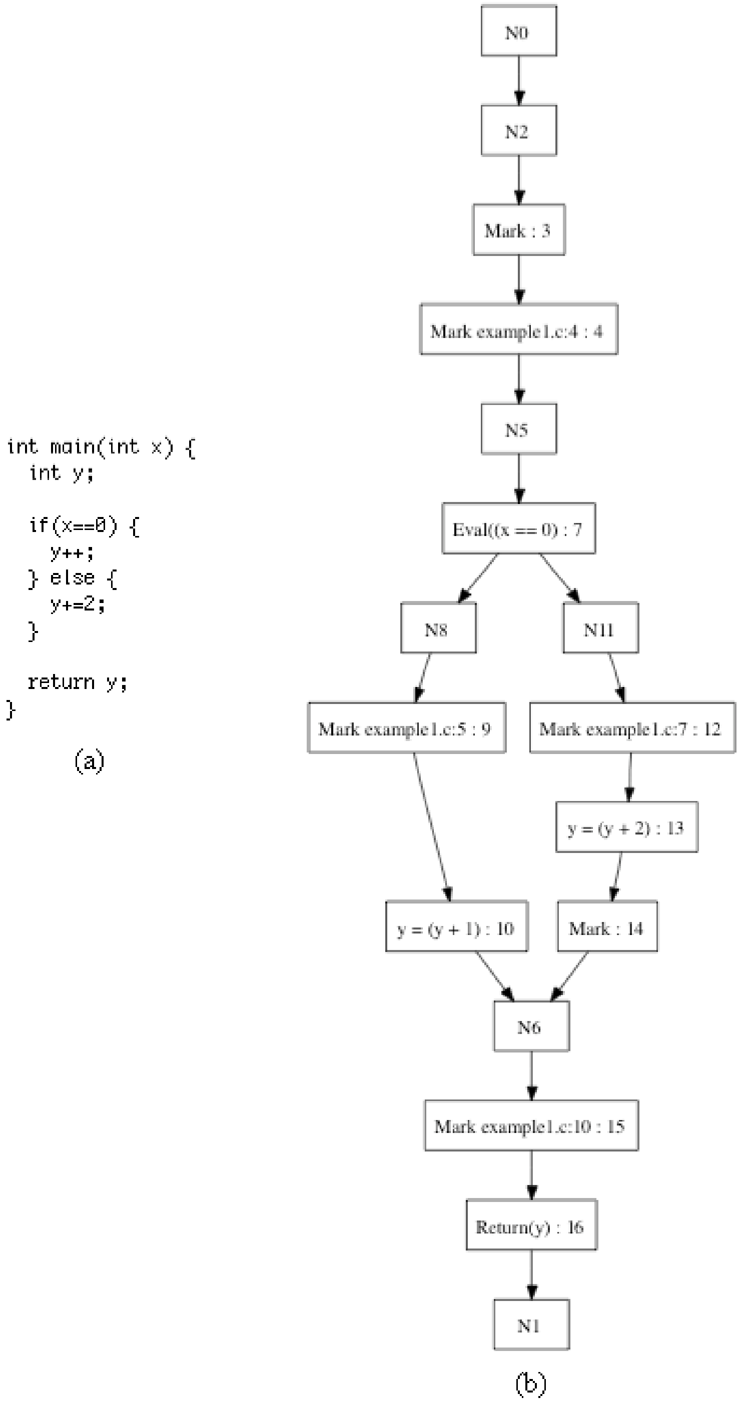
Figure 11: Example showing the commands used to execute the SUIF control flow graph pass.

The control flow graph pass includes two output options, a human readable text format, and a dot format graph which can be parsed with the unix tool dot to create a visual representation of the graph. During the execution of the control flow graph pass in Figure 11, the output was piped to a file. This file contains the dot text representation of the control flow graph. The dot command was used to create a postscript file containing the visual representation of the graph.

Figure 12(a) shows a simple C source code. The C source has been converted into the SUIF2 format and the control flow graph pass was executed on it. Figure 12(b) shows the corresponding control flow graph, that has been generated using the dot tool.

Control flow graphs that are produced from the SUIF pass are modified into a format that allows for easy control and data flow graph creation. Each block in Figure 12(b) represents a control flow graph node. The CDFG representation has to clearly define the differences in control flow and data flow. To do this, basic blocks are created. A basic block is a term used to define a block of code that contains only one control flow input, and only one control flow output. Therefore, the interior of a basic block is strict data flow.

Figure 14 shows the pseudo-code used to convert the CFG blocks into basic blocks. In the function, a list of all nodes that are identified as the beginning of a basic block is kept. The function initializes the list to contain the top node in the graph. A loop completes after there are no more nodes in the list. This loop calls the function, *create_basic_blocks_helper*,



```
int main(int x) {
  int y;

  if(x==0) {
    y++;
  } else {
    y+=2;
  }

  return y;
}
```

(a)

(b)

Figure 12: C source and generated CFG for a simple example.

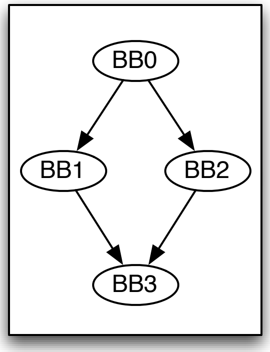


Figure 13: CFG of basic blocks for a simple example.

which checks nodes to see if they should be added to this basic block or the list of starting nodes. The helper function can be seen in Figure 15.

The basic block object is created in *create_basic_blocks_helper*. Figure 15 details the creation of the basic block and how nodes are added to it. After the block is created the algorithm checks to see if it is at the end of the graph. If the node has only one control flow input and only one control flow output it is added to the basic block and the algorithm continues on the successor. If the node has two or more outputs, it belongs to the current basic block, and all of its outputs are added to the starting node list. If the node has two or more predecessors, there is a control flow break before the node. In this case, the node is not added to the basic block. The node with multiple inputs is then added to the starting nodes list if it has not previously been added. This function returns after there is a break in the control flow and nodes are added to the starting nodes list.

Figure 13 depicts the control flow graph of the example code from Figure 12(a). Here, the *if-then-else* statement creates a control flow branch from basic block 0. The path along basic block 1 represents the *if* condition. The path through basic block 2 represents the *then* condition. The two paths converge in basic block 3, after all of the statements in both cases are completed.

```
//list of nodes that are the first node of a basic block
list startNodes

create_basic_blocks(the_graph) {
  //give each basic block a UID
  int uid = 0;
  //get the first node in the graph
  firstNode = the_graph->get_node_iterator()
  //add the first node to the list of start nodes
  startNodes.add(firstNode);

  while(startNodes is not empty) {
    //get the first node in the list
    node = startNodes.first();
    //call the function to create the basic block
    create_basic_blocks_helper(the_graph, node, uid);
    //remove the node from the list
    startNodes.remove(node);
    i++;
  }
}
```

Figure 14: Pseudo-code to call the function to create basic blocks, based on the starting nodes for each basic block.

4.2 DATA FLOW

After the control flow has been completed, the basic blocks that were created contain only data flow. Each basic block contains a list of ordered sequential statements which corresponds to the source code. In order to create a data flow graph for each basic block, the statements are parsed. Each executable SUIF statement is translated into multiple data flow graph nodes. At this point, the CDFG AST is completed and is independent of the SUIF2 AST.

In order to parse the statements an understanding of the SUIF data structures must be attained. In the SUIF AST any object that executes is called an *ExecutionObject*. Each node from the original control flow graph that SUIF creates, as seen in Figure 12(b), contains a pointer to an *ExecutionObject*. The *ExecutionObject* class is an abstract class with two subclasses, *Statement* and *Expression*. Both of these classes are also abstract and contain many subclasses. Each control flow graph node's *ExecutionObject* corresponds to a *Statement* class because each node contains an executable line of source code. Table 3 lists the subclasses of the *ExecutionObject* class.

4.2.1 The *statement* class

The data flow pass parses the statements based on their individual fields. Figure 16 shows a segment of the algorithm used to parse the *statement* class. First the algorithm must identify which type of statement it is parsing. Each statement is parsed in a different manner, based on its data structures. The *StoreVariableStatement* example from the figure contains a pointer to a variable symbol, the symbol is translated into a *cdfg_node_var* object. The statement also contains an expression. The result of the expression is stored into the variable. This is represented in the DFG by a line from the *cdfg_node_var* to the node created from the expression object.

```
create_basic_blocks_helper(the_graph, node, uid) {
  //create the basic block
  new bb(uid);
  while(true) {
    //Get the successors and predecessors
    out = the_graph->get_successors(node);
    in = the_graph->get_predecessors(node);
    if (there are successors)
      //move to the first one
      out.increment();
    } else {
      //if there are no sucessors, at end of the basic block
      break;
    }
    if (there are predecessors)
      //move to the first one
      in.increment();
    }
    if(there are no more successors AND
       there are no more predecessors) {
      //this is a node with only one input and output
      bb->add_node(node);
      //get the predecessor and loop
      out.reset();
      node = out.get();
    } else {
      //check to see if there is more then one successors
      if(there are more successors) {
        //add the node to the basic block. Node has multiple outputs and belongs to this bb
        bb->add_node(node);
        out.reset();
        if(the successors have not been added as startNodes)
          //get the successors and add them
          while(there are more successors) {
            node = out.get();
            startNodes.add(node);
            out.increment();
          }
        }
      }
      if(there are more predecessors) {
        //Node has multiple inputs and doesn’t belong to this bb
        if(the node has not been added as startNodes) {
          startNodes.add(node);
        }
      }
      //all possible paths are covered and
      break;
    }
  }
}
```

Figure 15: Pseudo-code to create the basic blocks, and identify the new starting nodes for basic blocks.

Table 3: The *ExecutionObject* subclasses.

Statement		
	Compound statements	
		StatementList
		ScopeStatement
	High Level Control Flow Statements	
		IfStatement
		WhileStatement
		DoWhileStatement
		ForStatement
	Simple Control Flow Statements	
		BranchStatement
		MultiWayBranchStatement
		JumpStatement
		JumpIndirectStatement
		ReturnStatement
		LabelLocationStatement
	Computational statements	
		StoreStatement
		StoreVariableStatement
		CallStatement
		EvalStatement
	VarArgs	
	Pseudo-Statements	
		MarkStatement
Expression		
	Constant	
	BinaryExpression	
	UnaryExpression	
	SelectExpression	
	Load Expressions	
		SymbolAddressExpression
		LoadExpression
		LoadVariableExpression
		LoadValueBlockExpression
	Data Structure Access Functions	
		ArrayReferenceExpression
		MultiDimArrayExpression
		FieldAccessExpression

```
void parse_statements(execution_object eo){
  //Find the statement type
  if (is_kind_of<BranchStatement>(eo)) {
    ...
  } else if (is_kind_of<CallStatement>(eo)) {
    ...
  }
  ...

  else if (is_kind_of<StoreVariableStatement>(eo)) {
    //Convert to the type
    StoreVariableStatement *stmt = to<StoreVariableStatement>(eo);
    //Get the variable that is stored to
    VariableSymbol *var = stmt->get_destination();
    String varString = var->get_name();
    //check to see if the variable has been used
    int i = _uidmap->get_id(varString.c_str());
    if(i == -1) {
      //the variable has not been used, initialize the version tracker
      i = 0;
      _uidmap->set(varString.c_str(),0);
    } else {
      //increment the version tracker
      _uidmap->set(varString.c_str(),i++);
    }
    //keep the id to later check if the same var is being read and written
    _varid = i;
    //store statements should always create new nodes
    cdfg_node_var *op = new cdfg_node_var(i, varString.c_str());
    //parse the right hand side of the statement
    cdfg_base *cb = parse_expressions(stmt->get_value());
  }

  ...

  else {
    assert(false);
  }
}
```

Figure 16: Pseudo-code for parsing a SUIF statement class.

4.2.2 The *expression* class

Each *statement* class typically has at least one embedded expression class. Expression classes can also have embedded expressions. The *expression* class is parsed in much the same way as the *statement* class. The expression parser is called first from the statement parser, and then it may be called recursively. Figure 17 shows a selection from the expression parsing algorithm. Like the statement parser, once the type of expression is determined CDFG nodes are created and linked to each other based on the expression's data structure.

The *BinaryExpression* class contains an operation and two source operands, which are *expression* classes. A *cdfg_node_binary* class is created and the type is set to the *BinaryExpression* classe's type. Next, the first source operand is sent to the expression parser. The expression parser will return a pointer to a CDFG node. This returned value will be stored as the first operand to the binary expression. The process is repeated with the second source expression.

4.2.3 The CDFG node class

The control and data flow pass creates CDFG node data structures to translate the SUIF objects into a data flow graph. The CDFG node data structures were created to be object oriented. The class structure for each of the nodes can be seen in Table 4. The root abstract class, *cdfg_base*, contains a unique id integer for each node, and a string value. The base node also contains methods that each child class must declare, such as a print method, and a node type identifier.

A *cdfg_node_input* is created when an expression is parsed and the variable has not yet been defined. The *cdfg_node_const* declares a special type of input node, a constant value. A constant input in created when the SUIF expression *Constant* is parsed.

When a variable node is defined, see Figure 16 for the *StoreVariableStatement* example, a *cdfg_node_var* is created. Variable nodes are often removed from the data flow graph because temporary variables between operation nodes are not required for data flow. The option to keep variable nodes in the graph has been added to show a correlation between the source

```
void parse_expressions(expression expr){
  if (is_kind_of<ArrayReferenceExpression>(expr)) {
    ...
  } else if (is_kind_of<BinaryExpression>(expr)) {
    //Convert to the type
    BinaryExpression *ex = to<BinaryExpression>(expr);
    //Find the binary type
    LString lstr = ex->get_opcode();
    String str = String(lstr);
    char *op_char = new char[20];
    type binary_type;

    if (lstr == k_add) {
      op_char = " + ";
      binary_type = add_type;
    } else if (lstr == k_multiply) {
      op_char = " * ";
      binary_type = multiply_type;
    } else if (lstr == k_subtract) {
      op_char = " - ";
      binary_type = subtract_type;
    } else if (lstr == k_divide) {
      op_char = " / ";
      binary_type = divide_type;
    }
    ...
    else {
      assert(false);
    }
    //Find the precision and sign
    DataType *dtype = ex->get_result_type();
    bool sign;
    int precision;
    get_info(dtype, &sign, &precision);
    //Create the node
    cdfg_node_binary *op = new cdfg_node_binary(_idmap->get_new_id(op_char), op_char, sign);

    //Parse the first source
    cdfg_base *cb = create_op_expr(ex->get_source1());
    op->set_first(cb);

    //Parse the second source
    cb = create_op_expr(ex->get_source2(), cnode);
    op->set_second(cb);
    return op;

  }
  ...
  else {
    assert(false);
  }
}
```

Figure 17: Pseudo-code for parsing a SUIF expression class.

Table 4: The CDFG node data structures.

cdfg_base				
	cdfg_node_input			
		cdfg_node_const		
	cdfg_node_var			
		cdfg_node_call		
		cdfg_node_unary		
			cdfg_node_binary	
				cdfg_node_mux

code and the CDFGs. At the time of the parsing, each variable node contains a list of control flow nodes that contains that nodes uses. In a post processing pass, the variable nodes that are defined within the basic block and used outside of the basic block are marked as output nodes. Detecting uses is done by using the SUIF2 reaching definitions package and an upwards use solver that was developed to determine the liveness of any variable. Variable nodes are represented as boxes in the visual representation.

Operations in the SUIF structure are defined as either the *UnaryExpression* class or the *BinaryExpression* class. The type of expression is extracted either a *cdfg_node_unary* or *cdfg_node_binary* node is created. The unary node class is an extension of the variable class that contains only one input node. Similarly, the binary node is an extension of the unary node class that contains exactly two input nodes. These operation nodes are represented in circles in the visual graph.

The *cdfg_node_mux* class is not used during CDFG creation. The mux node is an extension of the binary class and has three inputs, a true value, a false value, and a binary selector. This node is used later, during a graph transformation. The mux node is represented as an inverted trapezoid in the dot representation.

The *cdfg_node_call* class was created to show function calls in the CDFG. This node is the only node that can have any number of inputs. An octagon represents a call node in the visualized graph.

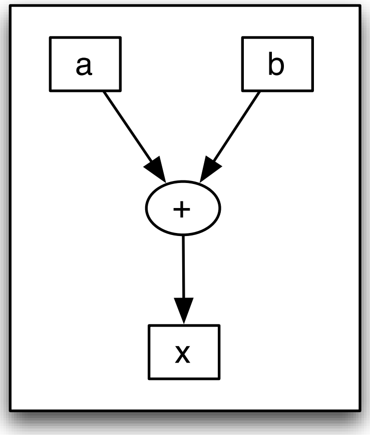


Figure 18: Example data flow graph segment for the statement: $x = a + b$.

4.2.4 A parsing example

The Pseudo-code provided from Figures 16 and Figure 17 can now parse the simple statement: $x = a + b$. The variable x is being used as the results of an addition operation, or a binary operation. This means that the *statement* class that corresponds to the equation would be a *StoreVariableStatement*. The *StoreVariableStatement* is parsed, creates a *cdfg_node_var* for x, and the expression parser is called on the expression.

The expression parser identifies a *BinaryExpression* object and creates a *cdfg_node_binary* class. The expression parser is then called on the first source which is a *SymbolAddressExpression* class. A *cdfg_node_var* for the variable a is created from the *SymbolAddressExpression* class and a pointer to the node is returned. The *cdfg_node_binary* then stores the pointer to the *cdfg_node_var* object as its left operand. The second source is also a *SymbolAddressExpression* class and the same process is repeated as the first source.

The expression parer, having completed, now returns a pointer to the *cdfg_node_binary* object. The statement parser now links the *cdfg_node_var* object for the variable x to the *cdfg_node_binary* object. Figure 18 shows the resulting data flow graph from the parsing of the statement.

```
int main(int x) {
  int y = 0;
  int i;

  for(i=0; i < 100; i++ ) {
    if(x==0) {
      y++;
    } else {
      y+=2;
    }
    x = (x + i) \% y;
  }

  return y;
}
```

Figure 19: Source code of a simple example to show the CDFG pass.

4.3 RESULTS

The control and data flow graph framework was created leveraging the control flow pass within the SUIF2 compiler. The control flow pass was extended to produce basic blocks, code containing only sequential statements. The control flow segment of the CDFG is the connections between these basic blocks. The sequential statements from each basic block are sent to the data flow graph generator. To generate data flow, each of the SUIF statements is parsed and data flow nodes are created. The data flow nodes are stored in the basic blocks.

Like the control flow graph pass, the CDFG pass includes two output options, a human readable text format, and a dot format graph. Dot is a unix tool that can be used to create a visual representation from a textual representation of a graph.

Figure 19 shows the source of a simple example. The example includes both conditional control flow structures and loop control flow structures. After the CDFG pass has completed, and the dot tool has been used to generate a visual representation of the CDFG. The CDFG can be seen in Figure 20.

The CDFG framework provides access to the data structures and allows developers to make changes to the graph. Chapter 5 discusses transformations to the CDFG structure

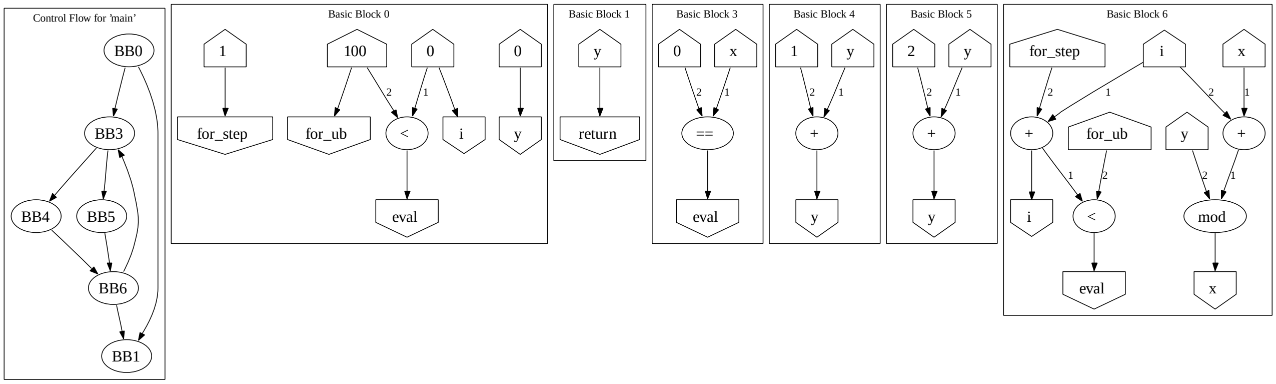


Figure 20: Example control and data flow graph for a simple source code.

to create a combinational data flow. In addition to this work, the framework is currently being used to investigate the mapping of hardware onto a course-grain computational fabric. Another project extends the CDFG framework with a static timing pass to provide users with hardware timing information. A third independent project examines parallel processing benchmark applications in an attempt to resolve communication patterns through inter-procedural variable resolution.

5.0 COMPILATION FOR HARDWARE FUNCTIONS

This chapter illustrates the compilation step required to translate the behavioral description of the application into a strict combinational data structure. Recall, the compiler is designed to create a tractable synthesis tool flow. The flow is outlined in Figure 21. First, the algorithm is profiled to discover the computational kernels. Section 5.1 describes how the source code is profiled to determine the computationally intensive loops.

The computational kernels discovered by profiling are propagated to a synthesis flow that consists of two basic stages. First, a set of well-understood compiler transformations including function inlining, loop unrolling, and code motion are used to attempt to segregate the loop control and memory accesses from the computation portion of the kernel code. The loop control and memory accesses are sent to the software flow while the computational portion is converted into hardware functions using a behavioral synthesis flow.

The behavior synthesis flow converts the computational kernel code into a CDFG representation. A technique called hardware predication is used to merge basic blocks in the CDFG to create a single, larger DFG. This DFG is directly translated into equivalent VHDL code and synthesized. The code transformations are described in detail in Section 5.2.

The remainder of the code, including the loop control and memory access portions of the computational kernels, is sent to a standard software compiler. For the architecture that was developed the code is passed through the Trimaran VLIW Compiler, see 3.2, for execution on the VLIW processor core. Trimaran was extended to generate assembly for a VLIW version of the NIOS II instruction set architecture. This code is assembled into machine code that directly executes on the VLIW processor architecture. The resulting hardware architecture is detected in Figure 22 This sequence was first described in [49].

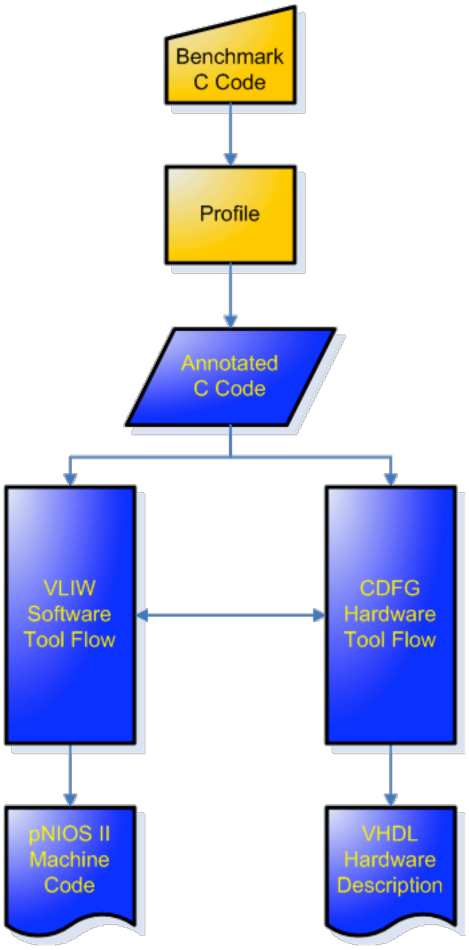


Figure 21: The VLIW - SuperCISC Compilation Flow

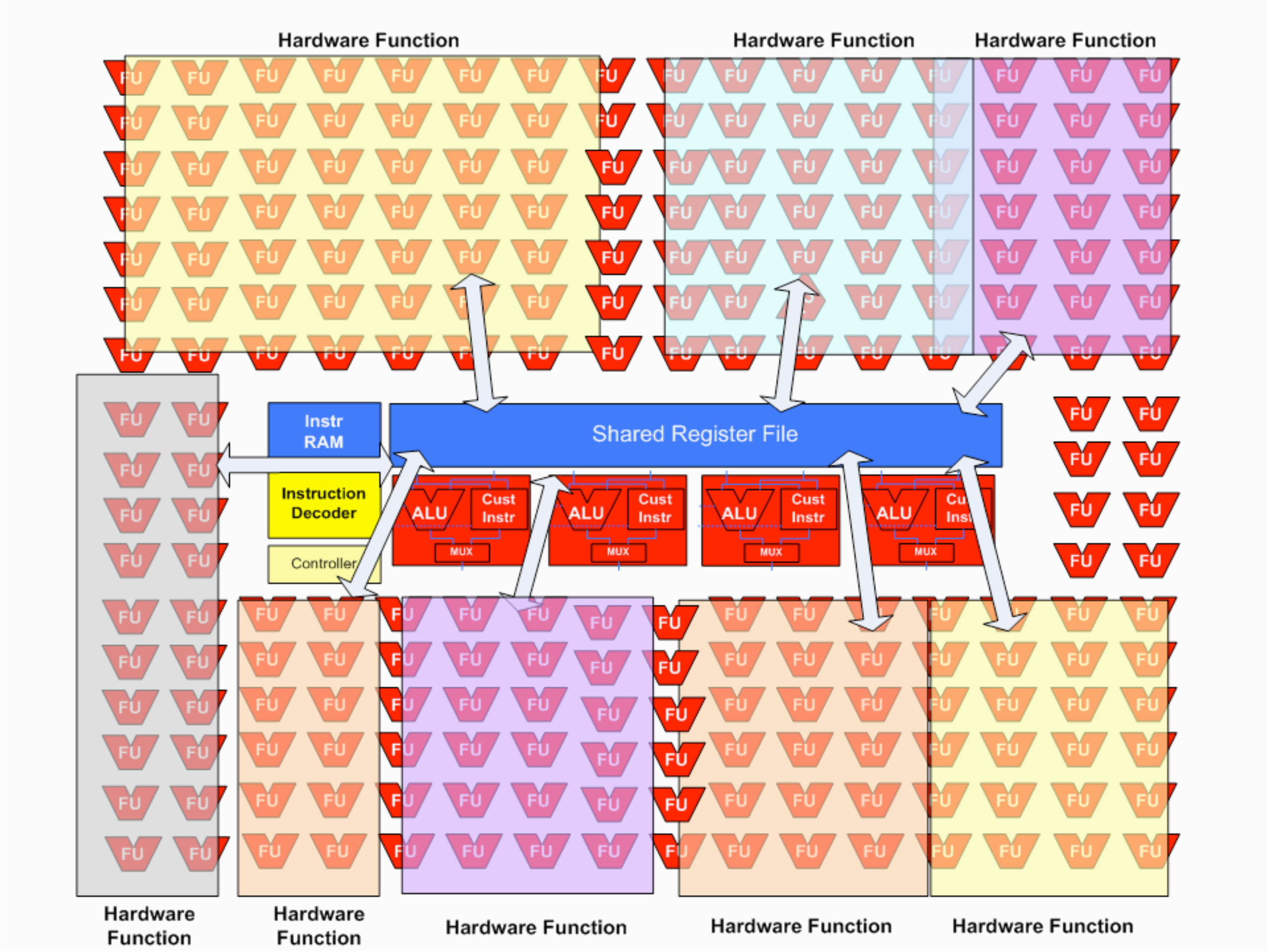


Figure 22: The VLIW - SuperCISC Architecture

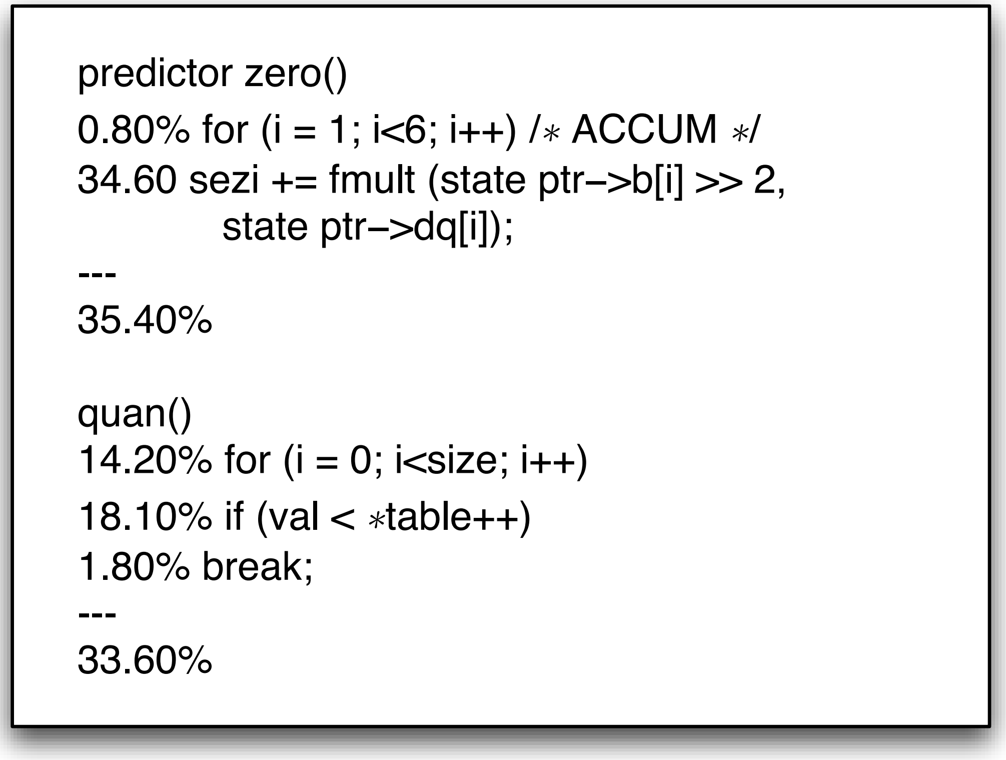
5.1 PROFILING

The profiling of an application's source code is a preprocessing step required for predication. The application is profiled so that the compiler can identify which code segments will eventually be translated to hardware execution. The profile information is passed into the compiler by modifying the source code. The application designer must specify to the compiler which code segments are to be translated into hardware. This step is done by the application designer placing the compiler directive *#pragma HWstart;* at the beginning of a hardware function and the compiler directive *#pragma HWend;* at the end of a hardware function. The results of the profiling preprocessing pass can be seen in Figure 24.

There are several profiling tools available for a variety of platforms. Currently, the Shark profiling tool from Apple Computer [57] can be used to profile programs compiled with the gcc compiler. The Shark profiling tool is designed to discover the code segments that contribute the most to the total program execution time. The tool returns results such as those seen in Figure 23. This Figure shows the two most executed loops from the g721 MediaBench benchmark. Combined, these loops account for a total of nearly 70% of the program execution time. These results are described in [58].

5.2 PREPROCESSING COMPILER TRANSFORMATIONS

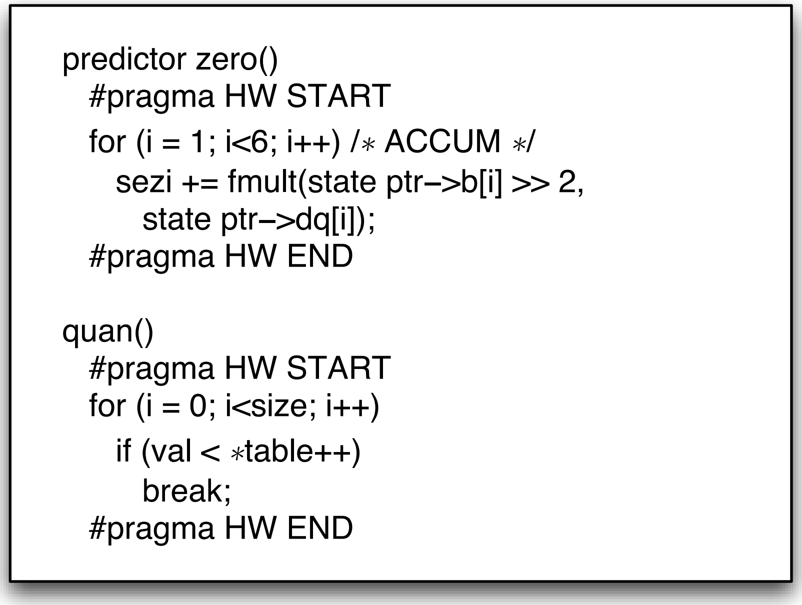
Synthesis from behavioral descriptions is an active area of study with many projects that generate hardware descriptions from a variety of high-level languages and other behavioral descriptions, see Chapter 2. However, synthesis of combinational logic from properly formed behavioral descriptions is significantly more mature than the general case and can produce efficient implementations. Combinational logic, by definition, does not contain any timing or storage constraints but defines the output as purely a function of the inputs. Sequential logic, on the other hand, requires knowledge of timing and prior inputs to determine the output values.



```
predictor zero()
0.80% for (i = 1; i<6; i++) /∗ ACCUM ∗/
34.60 sezi += fmult (state ptr–>b[i] >> 2,
            state ptr–>dq[i]);
---
35.40%

quan()
14.20% for (i = 0; i<size; i++)
18.10% if (val < ∗table++)
1.80% break;
---
33.60%
```

Figure 23: Excerpt of the Shark profiling results for the g721 benchmark.



```
predictor zero()
  #pragma HW START
  for (i = 1; i<6; i++) /∗ ACCUM ∗/
    sezi += fmult(state ptr–>b[i] >> 2,
      state ptr–>dq[i]);
  #pragma HW END

quan()
  #pragma HW START
  for (i = 0; i<size; i++)
    if (val < ∗table++)
      break;
  #pragma HW END
```

Figure 24: Code excerpt of results from Figure 23 after insertion of directives to outline computational kernels that are candidates for custom hardware implementation.

The compilation flow relies only on combinational logic synthesis to create a tractable synthesis flow. The compiler generates data flow graphs (DFGs) that correspond to the computational kernel and, by directly translating these DFGs into a hardware description language like VHDL, these DFGs can be synthesized into entirely combinational logic for custom hardware execution using standard synthesis tools.

Figure 25 expands the behavioral synthesis block from Figure 21 to describe in more detail the compilation and synthesis techniques employed by the compiler to generate the hardware functions. The synthesis flow is comprised of two phases. Phase 1 utilizes standard compiler techniques operating on an abstract syntax tree (AST) to decouple loop control and memory accesses from the computation required by the kernel, which is shown on the left side of Figure 25. Phase 2 generates a CDFG (Control and Data Flow Graph) representation of only the computational code and uses hardware predication to convert this into a single DFG for combinational hardware synthesis. This is an overview the compiler transformations taken from [49].

The kernel portion of the code is first compiled using the SUIF (Stanford University Intermediate Format) Compiler [56]. This infrastructure provides an AST representation of the code and facilities for writing compiler transformations to operate on the AST. The code is then converted to SUIF2, which provides routines for definition-use analysis. The next several transformations are common. The first is function inlining, followed by loop unrolling. The memory accesses are then moved out of the combinational logic with code motion. The resulting AST contains no function, loop, or memory overhead. The final transformation removes all remaining control flow from the AST and only data flow remains.

5.2.1 Definition-Use Analysis

Definition-use (DU) analysis, shown as the first operation in Figure 25, annotates the SUIF2 AST with information about how a symbol (e.g., a variable from the original code) is used. Specifically, a definition refers to when a symbol is assigned a new value (i.e., a variable on the left-hand side of an assignment) and a use refers to an instance in which that symbol is

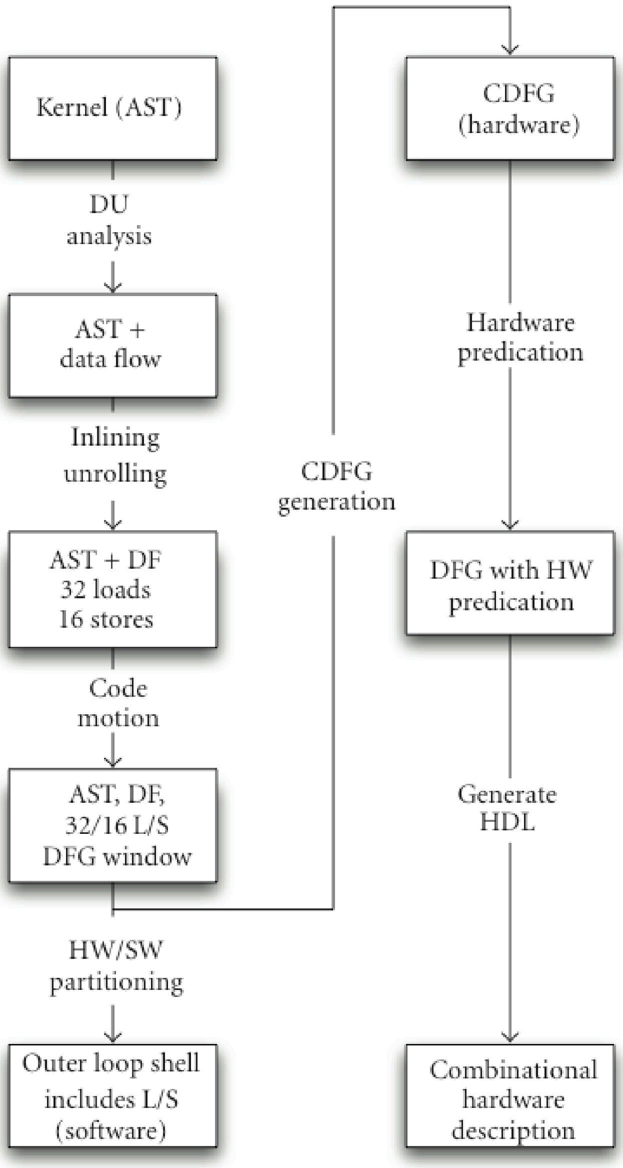


Figure 25: Description of the compilation and synthesis flow for portions of the code selected for custom hardware acceleration. Items on the left side are part of phase 1, which uses standard compiler transformations to prepare the code for synthesis. Items on the right side manipulate the code further using hardware predication to create a DFG for hardware implementation.

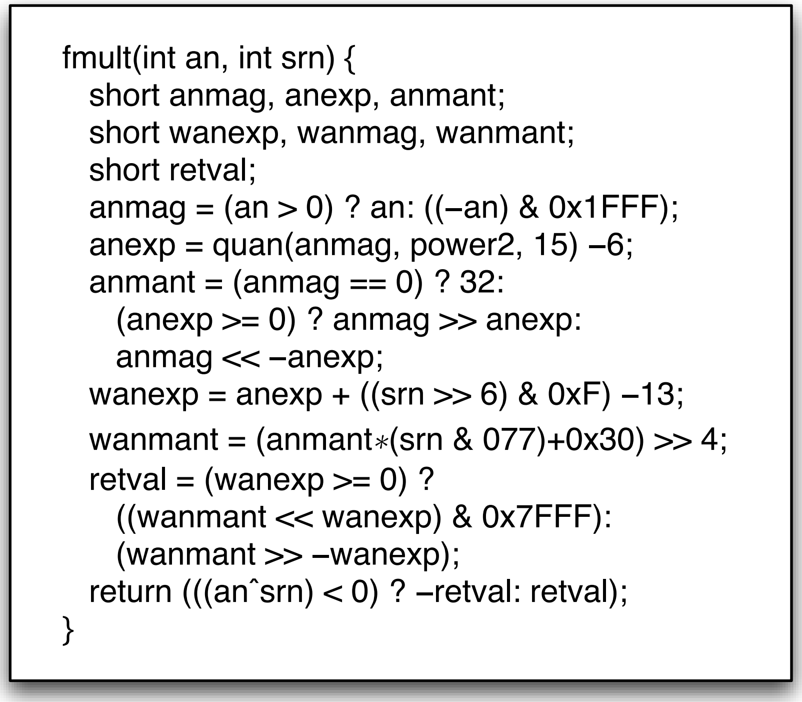
used in an instruction (e.g., in an expression or on the right-hand side of an assignment). The lifetime of a symbol consists the time from the definition until the final use in the code [49].

5.2.2 Inlining and Loop Unrolling

The subsequent compiler pass, as shown in Figure 25, inlines functions within the kernel code segment to eliminate artificial basic block boundaries and unrolls loops to increase the amount of computation for implementation in hardware. The first function from Figure 24, predictor zero(), calls the fmult() function shown in Figure 26. The fmult() function calls the quan() function which was also one of the most executed loops from Shark. Even though quan() is called (indirectly) by predictor zero(), Shark provides execution for each loop independently. Thus, by inlining quan(), the subsequent code segment includes nearly 70% of the program's execution time. The computational kernel after function inlining is shown in Figure 27. Note that the local symbols from the inlined functions have been renamed by prepending the function name to avoid conflicting with local symbols in the caller function.

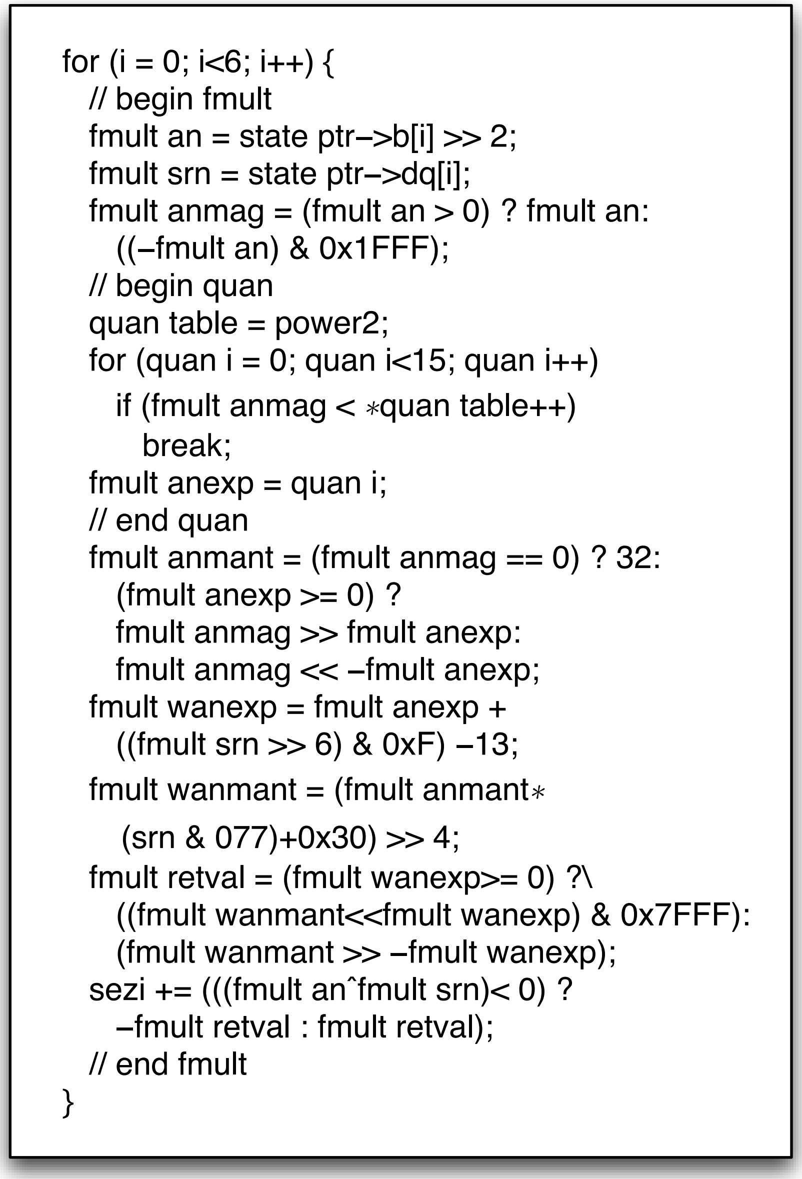
Once function inlining is completed, the inner loop is examined for implementation in hardware. By unrolling this loop, it is possible to increase the amount of code that can be executed in a single iteration of the hardware function. The number of loop iterations that can be unrolled is limited by the number of values that must be passed into the hardware function. In the VLIW architecture this limit is the number of values that can be passed through the register file. In the example from Figure 27, each loop iteration requires a value loaded from memory, the quan_table, and a comparison with the symbol fmult_anmag. Completely unrolling this loop results in a total of 16 reads from the register file. The resulting unrolled loop is shown in Figure 28.

Once the inner loop is completely unrolled, the outer loop may be considered for unrolling. In the example, there are several values in addition to the 16 values from the inner loop that have to be passed between hardware and software execution. In the VLIW architecture, values are passed through the register file. With the current VLIW architecture the limit on the number of values that can be passed between hardware and software execution is exceeded, thus preventing the outer loop from being unrolled. However, by considering a



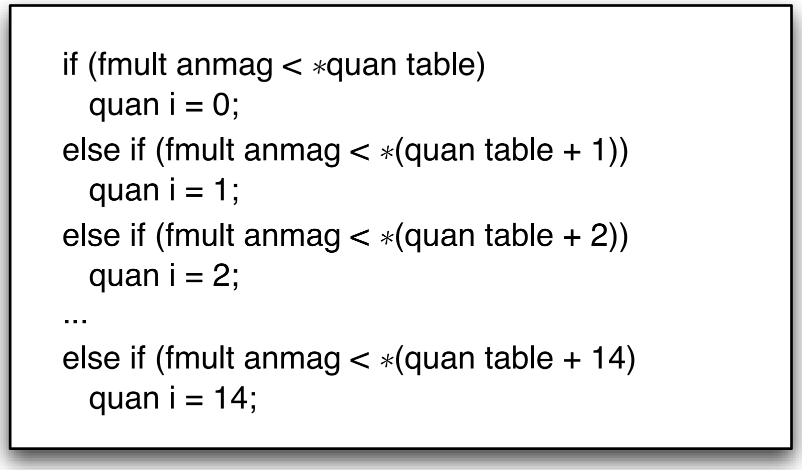
```
fmult(int an, int srn) {
  short anmag, anexp, anmant;
  short wanexp, wanmag, wanmant;
  short retval;
  anmag = (an > 0) ? an: ((–an) & 0x1FFF);
  anexp = quan(anmag, power2, 15) –6;
  anmant = (anmag == 0) ? 32:
    (anexp >= 0) ? anmag >> anexp:
    anmag << –anexp;
  wanexp = anexp + ((srn >> 6) & 0xF) –13;
  wanmant = (anmant∗(srn & 077)+0x30) >> 4;
  retval = (wanexp >= 0) ?
    ((wanmant << wanexp) & 0x7FFF):
    (wanmant >> –wanexp);
  return (((anˆsrn) < 0) ? –retval: retval);
}
```

Figure 26: Fmult function from G.721 benchmark.



```
for (i = 0; i<6; i++) {
  // begin fmult
  fmult an = state ptr–>b[i] >> 2;
  fmult srn = state ptr–>dq[i];
  fmult anmag = (fmult an > 0) ? fmult an:
    ((–fmult an) & 0x1FFF);
  // begin quan
  quan table = power2;
  for (quan i = 0; quan i<15; quan i++)
    if (fmult anmag < ∗quan table++)
      break;
  fmult anexp = quan i;
  // end quan
  fmult anmant = (fmult anmag == 0) ? 32:
    (fmult anexp >= 0) ?
    fmult anmag >> fmult anexp:
    fmult anmag << –fmult anexp;
  fmult wanexp = fmult anexp +
    ((fmult srn >> 6) & 0xF) –13;
  fmult wanmant = (fmult anmant∗
    (srn & 077)+0x30) >> 4;
  fmult retval = (fmult wanexp>= 0) ?\
    ((fmult wanmant<<fmult wanexp) & 0x7FFF):
    (fmult wanmant >> –fmult wanexp);
  sezi += (((fmult anˆfmult srn)< 0) ?
    –fmult retval : fmult retval);
  // end fmult
}
```

Figure 27: G.721 code after function inlining.



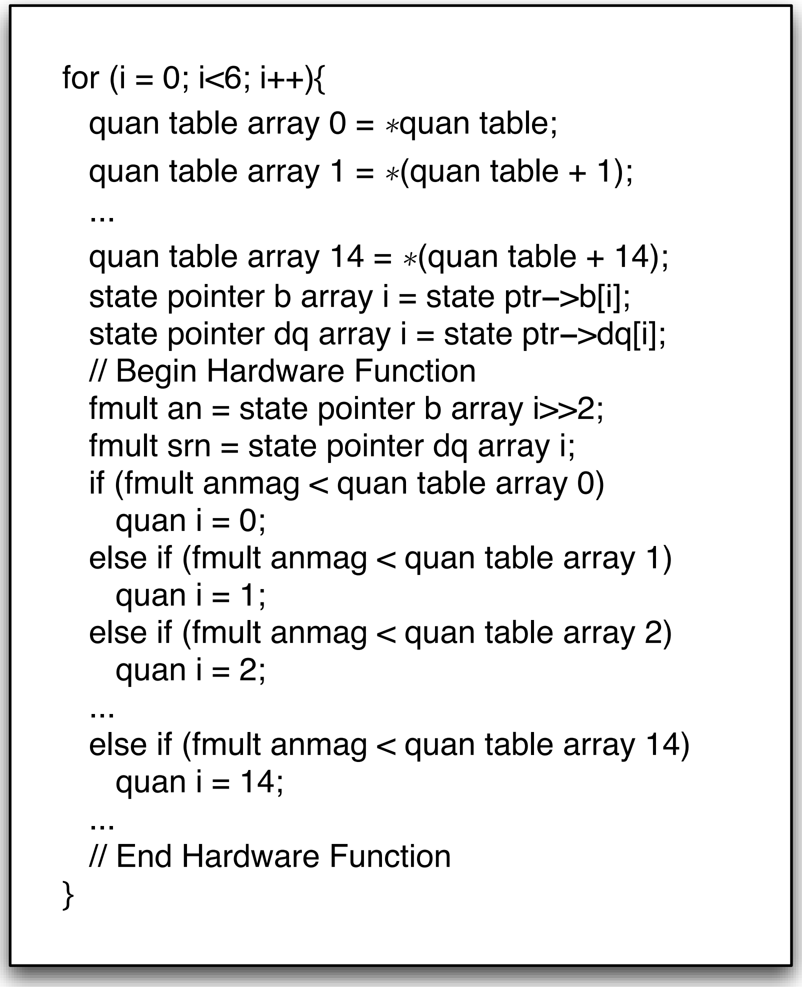
```
if (fmult anmag < ∗quan table)
  quan i = 0;
else if (fmult anmag < ∗(quan table + 1))
  quan i = 1;
else if (fmult anmag < ∗(quan table + 2))
  quan i = 2;
...
else if (fmult anmag < ∗(quan table + 14)
  quan i = 14;
```

Figure 28: Unrolled inner loop of inlined G.721 hardware kernel.

different architecture with a larger register file or special registers dedicated to hardware functions, this loop could be unrolled as well. This is the inlining and unrolling description published in [49].

5.2.3 Code Motion

After unrolling and inlining is completed, the next phase of the compilation flow uses code motion to move all memory loads to the beginning of the hardware function and move all memory stores to the end of the hardware function. This is done so as not to violate any data dependencies discovered during definition-use analysis. The loads from the unrolled code in the G.721 example, as seen in Figure 28 are from the array quan_table that is defined prior to the hardware kernel code. Thus, loading the first 15 elements of the quan_table array can be moved to the beginning of the hardware function code. The loads are now stored in static symbols, which are passed into the hardware function by mapping them to registers. This is possible for all array accesses within the hardware kernel code for G.721. The hardware kernel code after code motion is shown in Figure 29.



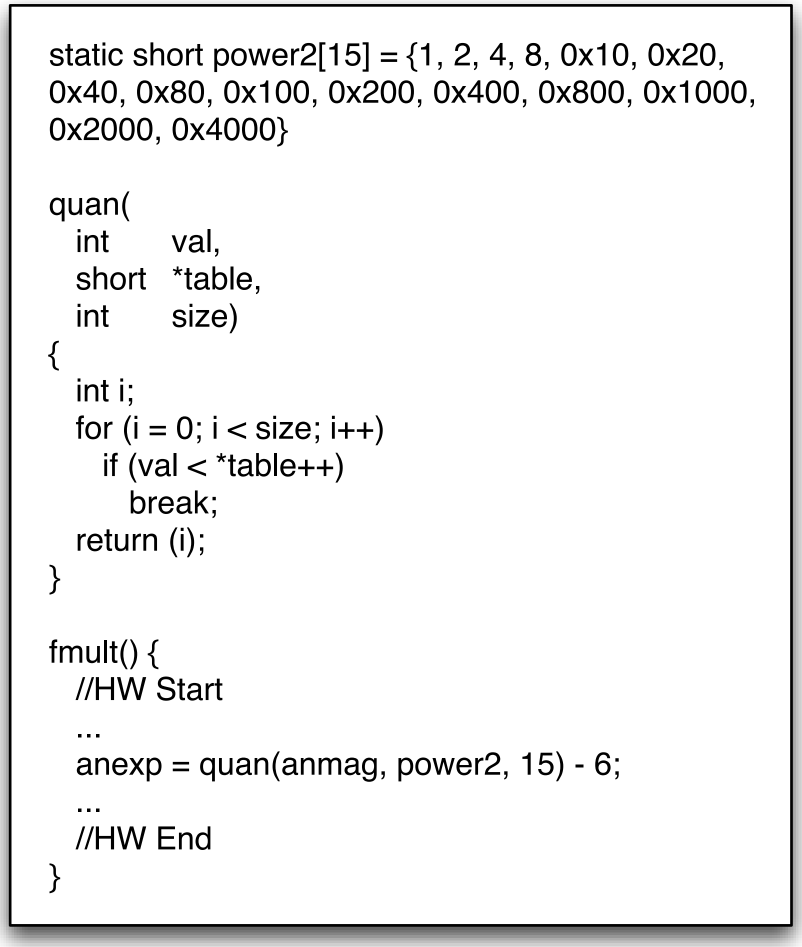
```
for (i = 0; i<6; i++){
  quan table array 0 = *quan table;
  quan table array 1 = *(quan table + 1);
  ...
  quan table array 14 = *(quan table + 14);
  state pointer b array i = state ptr–>b[i];
  state pointer dq array i = state ptr–>dq[i];
  // Begin Hardware Function
  fmult an = state pointer b array i>>2;
  fmult srn = state pointer dq array i;
  if (fmult anmag < quan table array 0)
    quan i = 0;
  else if (fmult anmag < quan table array 1)
    quan i = 1;
  else if (fmult anmag < quan table array 2)
    quan i = 2;
  ...
  else if (fmult anmag < quan table array 14)
    quan i = 14;
  ...
  // End Hardware Function
}
```

Figure 29: G.721 benchmark after inlining, unrolling, and code motion compiler transformations.

The resulting code after DU analysis, function inlining, loop unrolling, and code motion is partitioned between hardware and software implementation. The partitioning decision is made statically such that all code required to maintain the loop (e.g., loop induction variable calculation, bounds checking and branching) and code required to do memory loads and stores is executed in software while the remaining code is implemented in hardware. This distinction is shown in Figure 29, where hardware code is encapsulated by the comments, begin hardware function and end hardware function. Code motion is required for architectures which keep hardware and software memory independent. This code motion description is highlighted from [49].

Another technique that is used by the compiler to remove memory dependencies from the hardware function is the discovery of static loads. It is a common practice for software designers to use an array of predefined values in their code. If the compiler is able to identify a load within the code section that has been marked for hardware execution that is statically defined in the source code, it will translate the values into a ROM. For FPGA synthesis, the values will be coded into a lookup table. For an ASIC design flow, the compiler will generate a black box for the ROM. The goal of treating static memory loads differently from dynamic memory load is to minimize the number of values that have to be passed between software and hardware execution. In addition, by implementing the static array, which is essentially a lookup table, in hardware there should be an overall speed increase. The speed increased is realized by the fact that most modern processors require multiple cycles to access memory.

Upon reexamining the G.721 example, specifically the quantize function, a memory load can be seen. Upon further examination, it can be seen that the reference to the variable *table* is from an argument passed into the quan function. By examining the arguments of the call to the quan function, it can be realized that the variable *table* is a pointer to the static array *power2*. Figure 30 details the call to the quan function, and the memory load from a static array. Once the static load has been identified, the compiler will translate the values to a ROM. In Figure 30 the static array *power2* is loaded from within the hardware function. After detection, the compiler will create a ROM with 15 entries, one for each of the values from the static array.



```
static short power2[15] = {1, 2, 4, 8, 0x10, 0x20,
0x40, 0x80, 0x100, 0x200, 0x400, 0x800, 0x1000,
0x2000, 0x4000}

quan(
  int       val,
  short   *table,
  int       size)
{
  int i;
  for (i = 0; i < size; i++)
    if (val < *table++)
      break;
  return (i);
}

fmult() {
  //HW Start
  ...
  anexp = quan(anmag, power2, 15) - 6;
  ...
  //HW End
}
```

Figure 30: G.721 benchmark example of a static array.

5.3 HARDWARE PREDICATION

Once hardware and software partitioning decisions are made as described in Section 5.2, the portion of the code for implementation in hardware is converted into a control and data flow graph (CDFG) representation. This representation contains a series of basic blocks interconnected by control flow edges. Thus, each basic block boundary represents a conditional branch operation within the original code. Creation of a CDFG representation from a high level language is a well studied technique beyond the scope of this document. However, details on creation of these graphs can be found in 4.

In order to achieve a hardware implementation for the code contained within the computational kernel, the control dependencies of the CDFG must be converted into data flow dependencies. This allows basic blocks, which were previously separated by control flow dependency edges, to be merged into larger basic blocks. If all the control flow dependencies can be successfully converted into data flow dependencies, the entire computational portion of the kernel can be represented as a single DFG. As a result, the DFG can be trivially transformed into a combinational hardware implementation. Chapter 6 below describes a compiler back-end that generates VHDL from this DFG. The VHDL can then be synthesized and mapped efficiently into the target technology using existing synthesis tools.

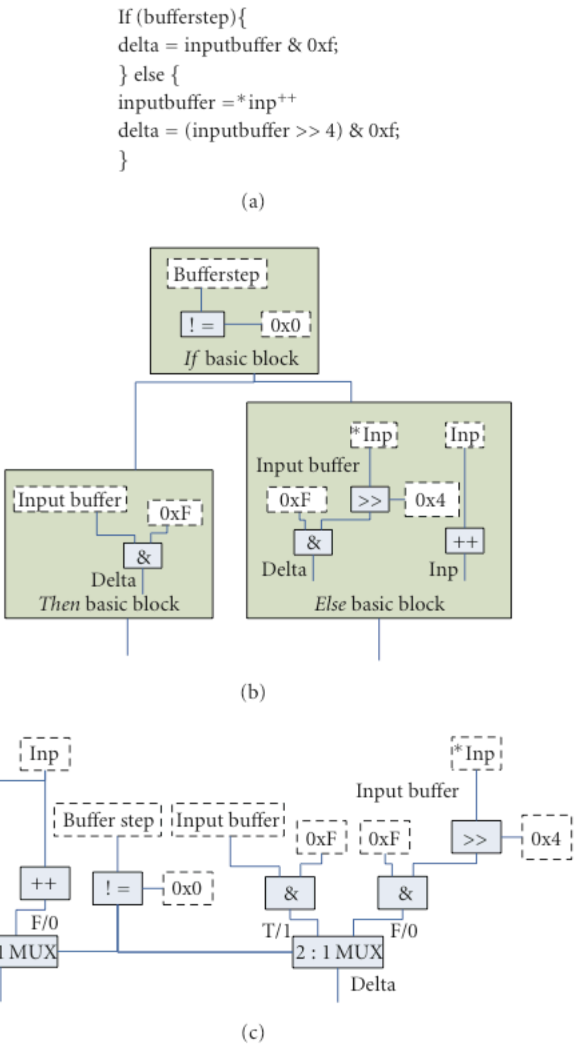
The technique for converting control flow dependencies into data flow dependencies is called hardware predication. This technique is similar to assignment decision diagram (ADD) representation, developed as an alternate behavioral representation for synthesis flows, see Section 2. Predication is also used in traditional VLIW compilers, where both paths of an *if-then-else* statement are often calculated on different processors. A flag is then set during execution based on the conditional, which can be executed on a third processor, which tells the two predicated paths which is the correct path.

Considering a traditional *if-then-else* conditional construct written in C code. In software, an *if-then-else* statement is implemented as a stream of instructions composed of the if comparison, the branch statements, and the code to be executed for both paths. In hardware, an *if-then-else* conditional statement can be implemented using a multiplexer acting as a binary switch to predicated output datapaths. Figure 31 shows several different representations of a segment of the kernel code from the ADPCM encoder benchmark. Figure 31(a) lists the source code, Figure 31(b) shows the corresponding CDFG representation of the code segment, and Figure 31(c) presents a data flow diagram for a 2 : 1 hardware predication (e.g., multiplexer) equivalent of the CDFG from Figure 31(b).

In the example from Figure 31, the *then* part of the code from Figure 31(a) is converted into the *then* basic block Figure 31(b). Likewise the statements from the *else* portion in Figure 31(a) are converted into the *else* basic block in Figure 31(b). The CDFG in Figure 31(b) shows that the control flow from the *if-then-else* construction creates basic block boundaries with control flow edges. The hardware predication technique converts these control flow dependencies into data flow dependencies allowing the CDFG in Figure 31(b) to be transformed into the DFG in Figure 31(c). Each symbol with a definition in either or both of the basic blocks following the conditional statement (i.e., the *then* and *else* blocks from Figure 31(b)) must be predicated by inserting a multiplexer. For example, in Figure 31, the symbol delta is defined in both blocks and these definitions become inputs to a rightmost selection multiplexer in Figure 31(c). The symbol inp is updated in the else basic block only in Figure 31(b). This requires the leftmost multiplexer in Figure 31(c), where the original value from prior to the condition and the updated value from the else block become inputs. All of the multiplexers instantiated due to the conversion of these control flow edges into data flow edges. The combinational path through the data that is chosen is based on the conditional operation from the if basic block in Figure 31(b).

By implementing the logic in this manner, the time required for the conditional and branch statement execution in the processor can be reduced to two levels of combinational logic in hardware. Considering the example of Figure 31, the assembly code requires as many as nine (9) cycles if the *else* path is selected, but the hardware version can be implemented as two levels of combinational logic (constant shifts are implemented as wires).

This type of hardware predication typically works in the general case and creates efficient combinational logic for moderate performance results. However, in some special cases, control flow can be further optimized for combinational hardware implementation. In C, switch statements, sometimes called multiway branches, can be handled specially. While this construct can be interpreted in sequence to execute the C code, directly realizing this



```
If (bufferstep){
delta = inputbuffer & 0xf;
} else {
inputbuffer =*inp++
delta = (inputbuffer >> 4) & 0xf;
}
```

Figure 31: Software code, CDFG, and DFG with predicated hardware example for control flow in ADPCM encoder.

construct with multiplexing hardware containing as many inputs as cases in the original code allows entirely combinational, parallel execution. A second special case exists for the G.721 example described in Section 5.2.3. Consider the unrolled innermost loop shown in Figure 29. This code follows the construction if (cond), else if (cond2), ..., else if (condN). This is similar to the behavior of a priority encoder in combinational hardware where each condition has a priority, such as high bit significance overriding lower bit significance. For example, in a one-hot priority encoder, if the most significant bit (MSB) is 1, then all other bits are ignored and treated as zeros. If the MSB is 0 and the next MSB is 1, then all other bits are assumed 0. This continues down into the least significant bit. When this type of conditional is written in a similar style in synthesizable HDL, synthesis tools will implement a priority encoder, just like a case statement in HDL implements a multiplexer. Thus, for the cases where this type of code is present for either the multiplexer or the priority encoder, this structure is retained. This section is an excerpt from [49].

The algorithm for predication is shown in Figure 32. The algorithm begins by walking the control flow structure of the CDFG framework. The algorithm attempts to find *if-then* and *if-then-else* control flow structures. After these control flow structures have been resolved, the predication pass combines the basic blocks into a single basic block.

Predicating *if-then* and *if-then-else* control flow structures is very similar. The algorithm for predicating *if-then* structures is shown in Figure 33. The algorithm provides two basic functions. The first function is to create a binary switch, and find the correct data path for the switches inputs. The second is to combine the parent and child basic blocks. By combing the basic blocks, the control flow path no longer exists in the CDFG.

The predication of *if-then-else* structures requires the combination of both children into the parent basic block. In addition, both children basic blocks must be taken into consideration. When connecting the source for the false binary path, the algorithm must first look at the opposite child basic block. The algorithm must also look at output nodes from both basic blocks and resolve the control flow for each of them.

```
void predication(cdfg graph){
  for( all basic blocks ) {
    //get the basic blocks children
    if( there are 2 children ) {
      //get the children’s children
      if( the first grandchild has 1 child AND the grandchild is the second child ) {
        //this is an if-then structure
        parse_if_then(parent, child1, grandchild);
      }
      //repeat with other child
      if( the both grandchildren have 1 child AND the grandchildren are the same) {
        //this is an if-then-else structure
        parse_if_then_else(parent, child1, child2, grandchild);
      }
    }
  }
  for( all basic blocks ) {
    if( there is one child AND that child has one parent ) {
      combine(parent, child);
    }
  }
}
```

Figure 32: Pseudo-code for parsing a SUIF expression class.

5.4 RESULTS

All of the compiler transformations described in this chapter are performed for a single purpose; the creation of combinational data flow. Each of the compiler transformations is done to increase the size of the combinational logic. After the application is profiled, the goal is to move as much software execution from the kernel into hardware execution. In the next chapter, chaper 6, the translation of the data structures created by the compiler to synthesizable hardware is described.

The compiler transformations should produce a single basic block CDFG, or essentially a single data flow graph that contains predication. These data flow plus predication graphs have been coined super data flow graphs, or SDFGs. By revisiting the source code from the G.721 benchmark previously described in this chapter, one can examine the resulting SDFG from the various transformed software structures.

```
void parse_if_then(parent, child1, grandchild) {
  //find the evaluation node and delete if
  //create a pointer to the comparison node
  for( output nodes in the child basic block ) {
    //create a mux node
    //set the compare nodes child is set to the mux node's select input
    //set the output node to the true condition of the mux
    //find the false path for the conditional
    for ( output nodes in the parent basic block ) {
      if ( the same variable was output from the parent )
        //set the node to the false condition of the mux
    }
    //if a path was not found try else where
    for ( input nodes in the child basic block ) {
      if ( the same variable was input from the child )
        //set the node to the false condition of the mux
    }
    //if a path was not found try else where
    for ( input nodes in the parent basic block ) {
      if ( the same variable was input from the parent )
        //set the node to the false condition of the mux
    }
    //if the variable has not been found, then it has not yet been defined
    //create an input node
    //set the node to the false condition of the mux
  }

  //if the parent outputs a variable and the child inputs the same variable
  for( output nodes of the parent basic block )
    for( input nodes of the child basic block )
      if( output node is the same as the input node )
        //link the nodes, and remove the input node

  //if both the parent and the child input the same variable and the parent doesn't output
  for( input nodes of the parent basic block )
    for( output nodes of the parent basic block ) {
      if( output node is the same as the input node )
        break;
      for( input nodes of the child basic block )
        if( input node is the same as the input node )
          //remove the child's input node
    }

  //move all nodes from the child to the parent
  //delete the child basic block
}
```

Figure 33: Pseudo-code for the predicating an *if-then* control flow structure.

Figure 34 below contains the G.721 source code for the kernel from the Mediabench benchmark suite. To show the control flow nature of this benchmark, the CDFGs before transformations are added are shown in Figures 35, 36, and 37. Recall the the left-most box contains the control flow information for the basic blocks. To the right of this box each basic block's DFG is presented.

By examining the CDFG for the *predictor_zero* function two function calls to the *fmult* function can be seen. Similarly, the *fmult* function contains a call to the *quan* function. Within the *quan* function there is a *for* loop. This loop must be unrolled to create a single DFG for the entire kernel. Through inspection, it can be determined that the *size* variable of the *quan* function is a constant and therefore the loop can be easily unrolled.

After the *quan* functions *for* loop has been unrolled, the function is inlined to eliminate the function call overhead. Predication removes all control flow from the *if-then* and *if-then-else* structures found in the *fmult* functions. Next, the *fmult* function is inlined and only a single function now exists. The next step is to unroll the loop in the *predictor_zero* function. Once this is complete the loads and stores, to the struct *state_ptr*, are move outside of the hardware function. This enables the creation of a single DFG for the G.721 benchmark. This graph is very large, containing over 1500 nodes, a rough visual representation can be seen in Figure 38.

The contribution of this chapter is the *hardware predication* pass. The other compiler transformations have been done before by various research and industry projects. There are SUIF passes available to do function inline and loop unrolling. For the purposes of research, these transformation were completed by hand. They revised compilation flow can be seen in Figure 39. The yellow blocks represent code from the SUIF2 library, the blue blocks represent coded passes, and the green block represent changes generated by hand.

```
static short power2[15] = {1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80,
0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000, 0x4000};

static int quan( int val, short *table, int size) {
int i;
for (i = 0; i < size; i++)
if (val < *table++)
break;
return (i);
}

static int fmult( int an, int srn) {
short anmag, anexp, anmant;
short wanexp, wanmag, wanmant;
short retval;

anmag = (an > 0) ? an : ((-an) & 0x1FFF);
anexp = quan(anmag, power2, 15) - 6;
anmant = (anmag == 0) ? 32 :
    (anexp >= 0) ? anmag >> anexp : anmag << -anexp;
wanexp = anexp + ((srn >> 6) & 0xF) - 13;

wanmant = (anmant * (srn & 077) + 0x30) >> 4;
retval = (wanexp >= 0) ? ((wanmant << wanexp) & 0x7FFF) :
    (wanmant >> -wanexp);

return (((an ^ srn) < 0) ? -retval : retval);
}

int predictor_zero( struct g72x_state *state_ptr) {
int i;
int sezi;

sezi = fmult(state_ptr->b[0] >> 2, state_ptr->dq[0]);
for (i = 1; i < 6; i++) /* ACCUM */
sezi += fmult(state_ptr->b[i] >> 2, state_ptr->dq[i]);

return (sezi);
}
```

Figure 34: Source Code for the G.721 Benchmark Kernel.

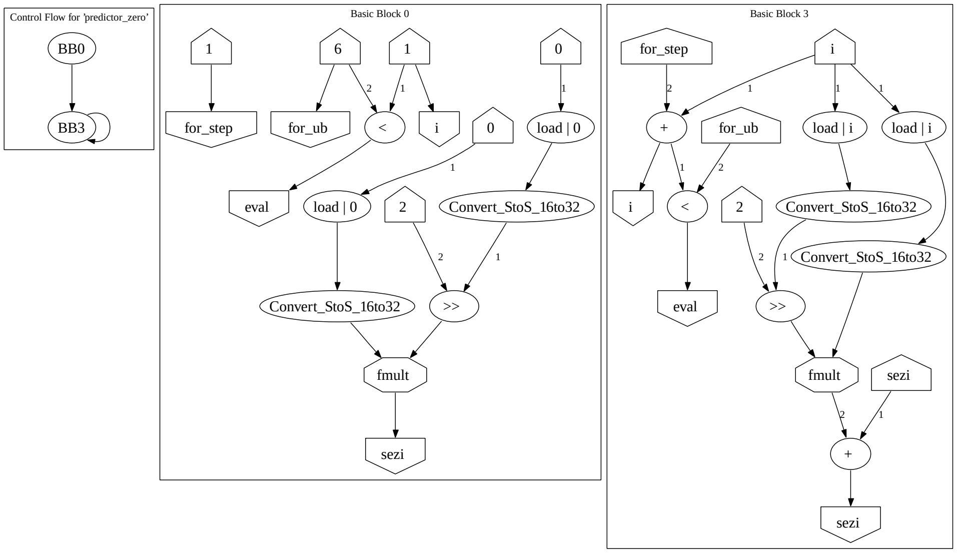


Figure 35: CDFG for the *predictor_zero* Function of the G.721 Benchmark.

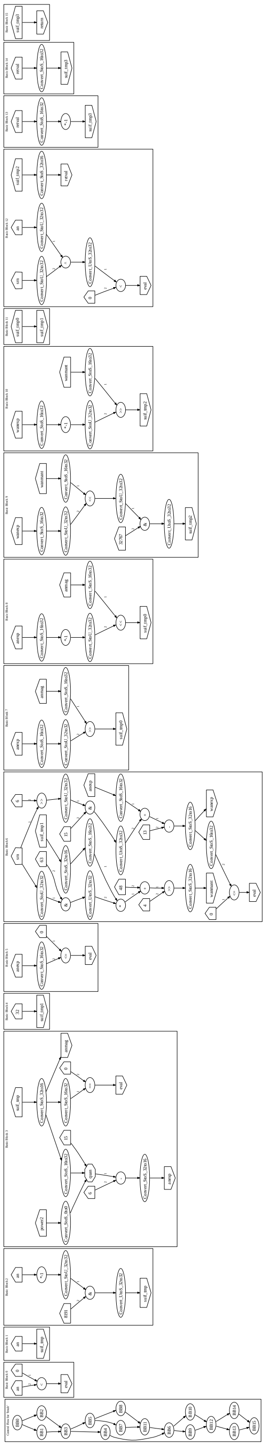


Figure 36: CDFG for the *fmult* Function of the G.721 Benchmark.

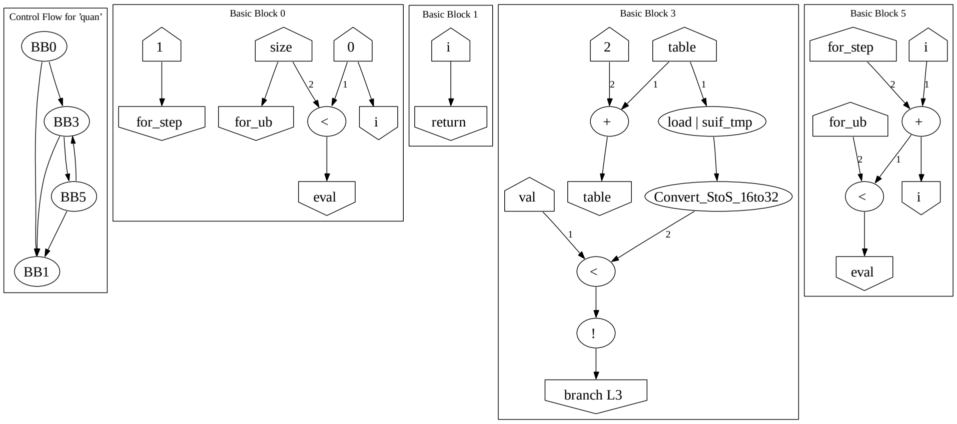


Figure 37: CDFG for the *quan* Function of the G.721 Benchmark.

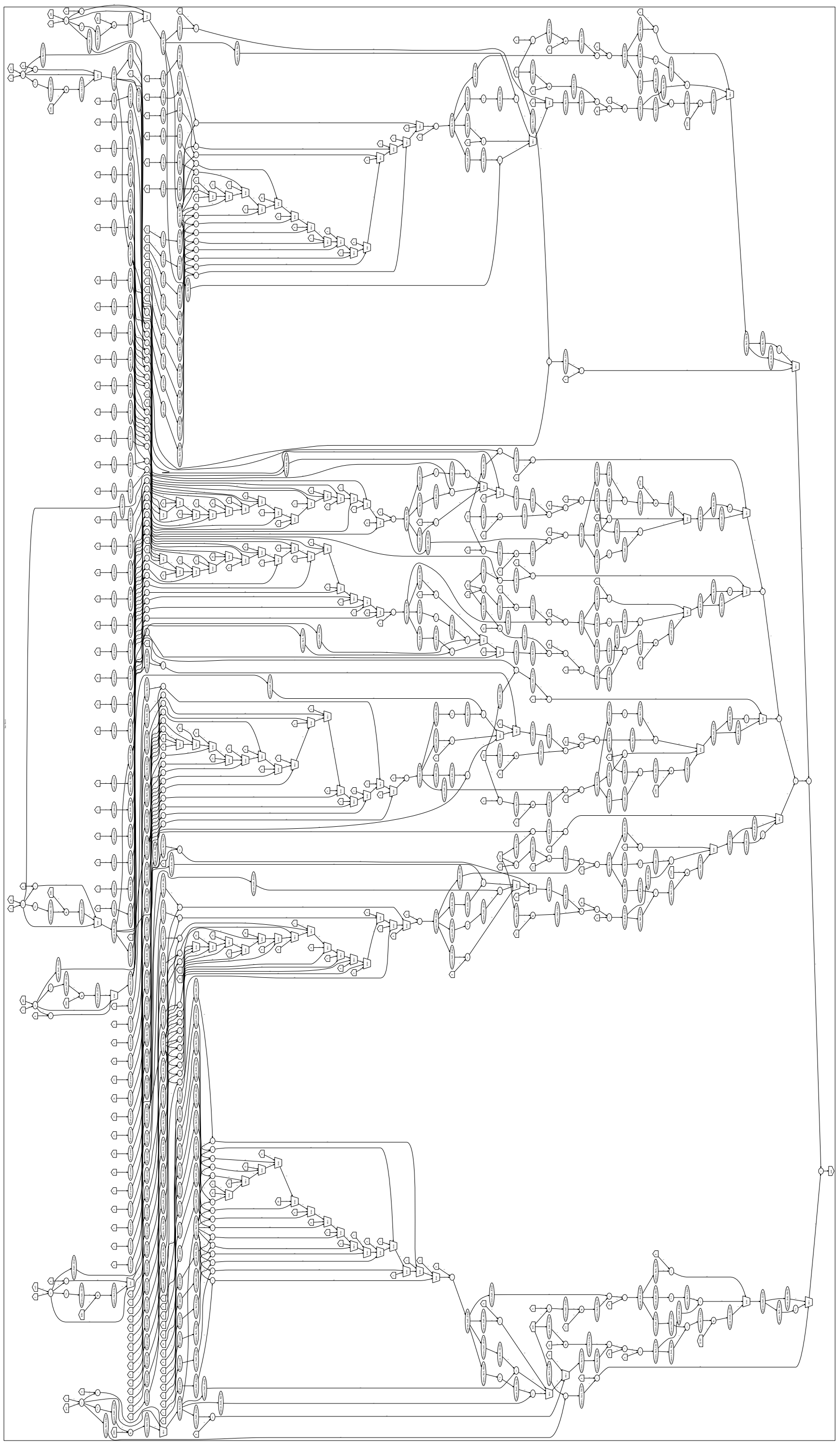


Figure 38: CDFG for the Entire G.721 benchmark after Compiler Transformations.

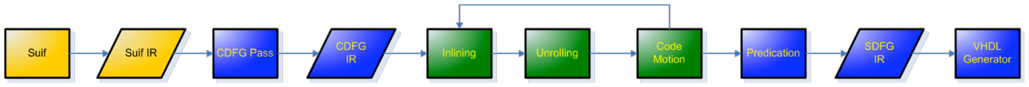


Figure 39: SuperCISC compilation flow - Blue blocks represent added functionality.

6.0 ASPECTS OF HARDWARE GENERATION

Chapters 4 and 5 detail the automatic creation of hardware functions through the compiler flow. This chapter discusses the translation, interfacing, and expected performance from the creation of hardware functions. Section 6.1 describes the details of the SUIF compiler data structure to VHDL translation. After the hardware functions have been generated, they can be synthesized as independent VHDL.

The final goal of the compilation, execution of software code on a processor architecture with hardware functions, can now be attained. Section 6.2 defines a hardware/software interface between the VLIW processor architecture was described in chapter 3 and the hardware function VHDL that is automatically generated.

In Section 6.3, the theory behind the performance gain of the hardware functions is outlined. Because of the poor ILP results of the VLIW architecture, see 3.3, the additional FPGA real-estate is being utilized with application specific hardware to increase the application ILP. This section describes the expected increase in ILP and provides initial benchmark results.

Lastly, this chapter is concluded in section 6.4, where the implementation details and required future work of the hardware generation is addressed.

6.1 VHDL GENERATION

The final result of the compilation flow is synthesizable VHDL. In chapter 4 the compiler created the data structures required for hardware generation. Chapter 5 discussed the transformations to the CDFG data structure required for hardware generation. The input to the

```
void create_hw(cdfg graph){
  //create the design file
  DesignFileClass *df = new DesignFileClass();
  //create the entity declaration
  EntityDeclrClass *ed = new EntityDeclrClass();
  //add the io
  addEntityIO(ed, graph);
  //create the architecture declaration
  ArchitectureDeclrClass *ad = new ArchitectureDeclrClass();
  //walk the graph
  walkCDFG(ad, graph);
  //create the design unit, add the design libraries, and add design unit to the design file
  DesignUnitClass *du = new DesignUnitClass(ed);
  createLibDeclr(du);
  df->addDesignUnit(du);
  //create a second design unit for the architecture class and add the design unit to the design file
  DesignUnitClass *du2 = new DesignUnitClass(a)
  df->addDesignUnit(du2);
}
```

Figure 40: Pseudo-code for top level hardware generation.

hardware generation pass is a CDFG with a single predicated data flow graph. A single VHDL component is created for each type of node. Then, every node in the graph instantiates the corresponding component. This structural VHDL approach was used to allow for the exploration of allowing nodes to be synthesized into different hardware structures.

Figure 40 shows the pseudo-code for the creation of the top level VHDL data structures. A VHDL abstract syntax tree created by Alex Jones for a different project was used for the VHDL IR. This pass translates the CDFG to the VHDL IR. The VHDL AST contains data structures that behave according to the VHDL specification. The highest VHDL data structure is a design file. Therefore, each CDFG creates a *DesignFileClass*. A design file contains an entity declaration and an architecture declaration. Classes are instantiated for each of these data types and added to the design file.

A VHDL entity declaration contains the description of the communication between that entity and the outside. This is done through the use of ports. Each input node to the graph creates into input port to the entity. Similarly, each output node in the graph creates an output port. In Figure 41 this simple algorithm is presented. There is a call of the

```
void addEntityIO(EntityDeclrClass ed, CDFG graph){
  //get the basic block
  bblist the_bblist= cdfg->get_bblist();
  //there should only be a single block
  bb the_bb = the_bblist.first();
  //create ports for each input and output node
  for(each io node in the bb) {
    InterfaceClass *ifc = convertCdfgBase(node);
    ed->addPortInterface(ifc);
  }
}
```

Figure 41: Pseudo-code for Entity Declaration Parsing.

convertCdfgBase function. This function simple looks at the node and determines if it is an input or an output and looks at the node's sign and precision to create the correct signal type.

To complete the translation, the CDFG nodes are translated into the hardware description. The VHDL architecture contains the description describing the function of the input nodes to define the output. The pseudo-code for translating the architecture can be seen in Figure 42. The first stop is to create a signal for every node within the CDFG. The signal is created based on the nodes unique ID. Constant values are handled somewhat differently because a value is initially assigned to them. The next step is to look at each node in the graph and create unique component declarations. The components are declared and added to the architecture declaration. In addition the full VHDL for each component is generated. For example, is an add type is found the full VHDL, entity and architecture for an adder is created and appended to the design file. After all of the component are declared they must be instantiated. This involves looking at each of the operation nodes a second time and linking the signal names which have already been created. The last step is to link the bottom CDFG nodes to the output ports. This is done by finding the output nodes and defining their parents as the signal that drives them.

```
void walkCDFG(ArchitectureDeclrClass ad, CDFG graph){
  //create the list of signals.
  bblist the_bblist= cdfg->get_bblist();
  //there should only be a single block
  bb the_bb = the_bblist.first();
  //create signals for the internal nodes
  for(each node in the bb) {
    if( type is operation or mux) {
      a->addBlock(SignalDeclrClass *sd = convert_signal(cb));
    } else if(constant type) {
      a->addBlock(ConstantDeclrClass *sd = convert_constant(cb));
    } else {assert(false);}
    //Declare the components
    for( each node in bb ) {
      if(have not seen this node type) {
        create component for this type;
      }
    }
    //Instantiate the components
    for( each node in bb ) {
      if( type is operation or mux) {
        instantiate the component;
      }
    }
    connect_ports(ad, graph);
  }
}
```

Figure 42: Pseudo-code for Architecture Declaration Parsing.

The VHDL AST was modified to compile within the SUIF2 compiler. This required changing the STL structures to use the SUIF STL replacement structures. The SUIF STL replacements are made to behave similarly to the STL structures but some of the methods from the STL structures are not defined. The VHDL AST was also modified to add support for VHDL generics. The last change that was made redefined some of the shift operators to use the numeric standard library shift calls correctly.

6.2 INTERFACING SOFTWARE AND HARDWARE

The interface for the hardware functions that is proposed for the architecture is a shared register file. By sharing the register file between the hardware and processor, the hardware function can be called with no additional overhead requirements versus that of executing the code directly in software. This is a significant difference than many other systems that combine processors with coprocessor accelerators.

Consider the execution profile from Figure 43, running an algorithm that contains function calls and loops requires certain types of overhead including pushing/popping off the stack for function arguments, and initializing and maintaining loop counters. For bus based co-processors, there is additional overhead to send data across the bus for execution and receive data back upon completion. However, because the VLIW architecture can access the entire register file for reading and writing there is no additional overhead for executing a function in hardware. While all reads and writes are accomplished by the VLIW processor independently from the hardware unit, this is exactly what the software version requires with no hardware execution. While this software execution time is not explicitly overhead, it can impact the available kernel speedup through hardware acceleration. The complete architecture description can be found in [58].

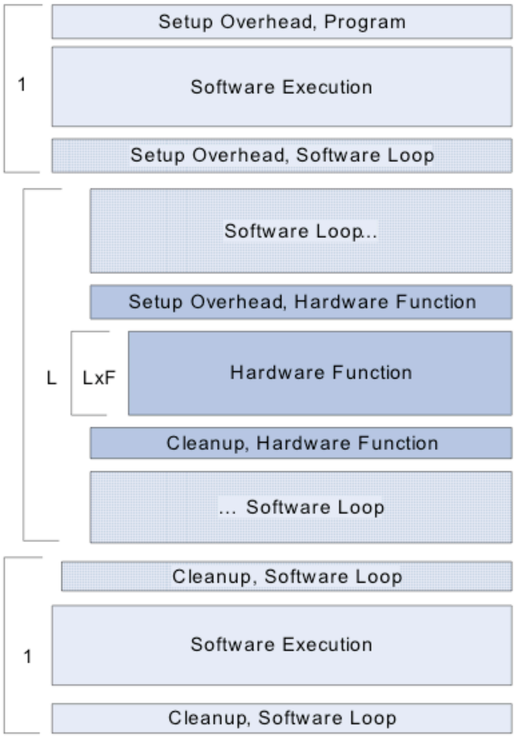
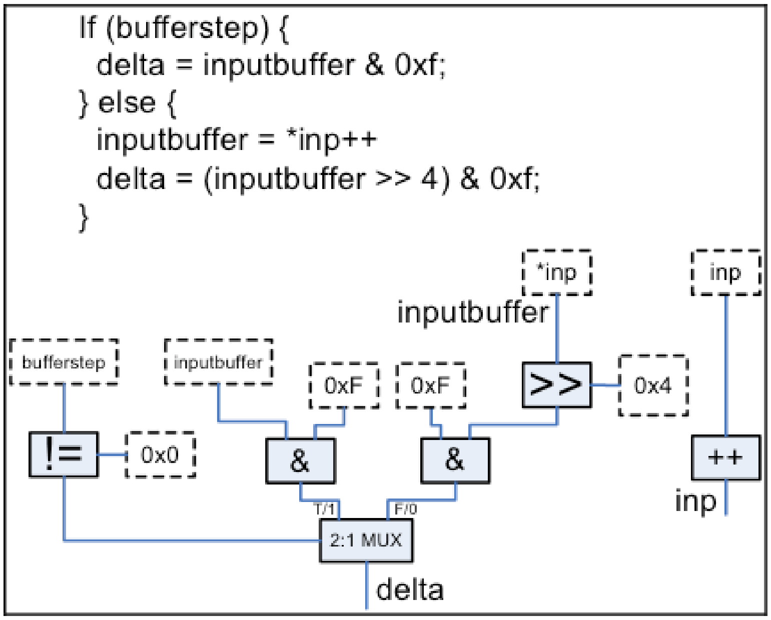


Figure 43: Algorithm execution profile including potential overheads. In the VLIW processor with hardware functions, the hardware functions require no hardware specific setup overhead.



```
If (bufferstep) {
  delta = inputbuffer & 0xf;
} else {
  inputbuffer = *inp++
  delta = (inputbuffer >> 4) & 0xf;
}
```

Figure 44: Software code and data flow graph (DFG) showing control flow in ADPCM encoder.

6.3 ASPECTS OF PERFORMANCE GAIN

Hardware functions can achieve superlinear speedup, a performance gain significantly greater than the an application's ILP. This speedup is due in part to the efficiency of control flow within hardware.

In hardware, an *if-then-else* conditional statement is implemented as a multiplexer acting as a binary switch to predicated output datapaths. In software, an *if-then-else* statement is implemented as a stream of multiple instructions composed of comparisons and branch statements. Assembly code and a data flow diagram for a 2:1 multiplexer equivalent are shown in Figure 44.

To test this concept, several multiplexer were synthesized on an EP2S180 Altera Stratix II FPGA. On the EP2S180, a 32-bit 2:1 multiplexer with 1-way comparison (LT, GT, NE) and default condition runs at 264MHz, and a 2:1 multiplexer with 2-way comparison (LTE, GTE) and default condition operates at 206MHz. Both have latencies that are less than

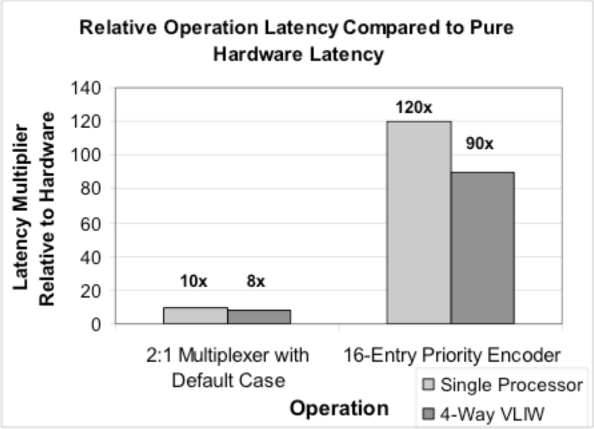


Figure 45: Latency for software equivalent of 2:1 multiplexer and 16-entry priority encoder relative to hardware implementation.

one cycle on a 167MHz processor. An equivalent operation in software requires 6 cycles on a single processor or 5 cycles on a VLIW processor. The latency of software multiplexer implementations relative to a hardware implementation is shown in Figure 45.

The speedup of *if-then-else* statements in hardware is partly due to bypassing of the register file during control flow. The effect of this latency reduction, is called cycle compression, and is discussed below. Speedup can also be explained by the limited control flow possible within the VLIW. Only one slot in the VLIW is permitted to write Reg31, the control register, at a time. Pursuant to branching, only one slot is permitted to perform branch control within a VLIW to avoid possible conflicts. In hardware, these comparisons can be performed simultaneously, and the performance and area cost of the multiplexing is negligible, consisting of a binary switch.

The performance gain of executing control flow in hardware is further supported by the performance of a 16-entry priority encoder that can be used in benchmarks such as the G.721 kernel. An input is compared to a constant set of 16 values, or bins, and the appropriate output value selected. In software, this operation consumes 48 instructions. In hardware, this operation consumes less than one processor cycle. The latency of software relative to a hardware implementation of the encoder is shown in Figure 45.

A 4-wide, 167 MHz VLIW can implement a 16-entry priority encoder in 36 instructions, realizing only a 25% savings in instruction count over a single processor. Figure 45 also implies that control flow constructs are a severe limit to ILP.

Cycle compression allows a hardware function to execute a sequence of arithmetic operations at a fraction of latency for the same sequence performed on a pipelined RISC processor. A standard CPU wastes a portion of the cycle time for simple operations, such as fixed-shifts, to wait for the critical path. Hardware, however, can execute an arithmetic operation according to the node's maximum clock speed within the chip without having to wait before proceeding to the next node.

The benchmark processor for comparison is the 4-way VLIW NIOS processor described in chapter 3. The 4-way VLIW executes all arithmetic operations at a constant fMax of 167MHz. This operating frequency is the same as the industry produced Altera NIOS processor. The critical path of the VLIW processor is not set within ALU, so a proportion of the cycle time for every ALU operation is spent waiting or idle to accommodate the slowest path of the CPU. For example, on an EP2S180 device, a 32-bit addition can execute at 346MHz, more than twice the speed of any operation on the 4-way VLIW. Operations having one fixed operand show better performance than those having two unfixed operands. The latency for a candidate hardware function can be estimated empirically by summing latencies for each node within the DFG and finding the maximum to determine the critical path.

Figure 46 charts the cycle time compression for core ALU functions. Logic operations benefit the most from execution within function and variable shifters the benefit the least. On a Stratix II, multipliers are implemented within digital signal processing (DSP) blocks rather than within lookup tables (LUTs) and therefore make no demand on the configurable logic area. A 32-bit multiplier within an EP2S180 requires 8 DSP blocks, or 1% of the 768 total DSP blocks available.

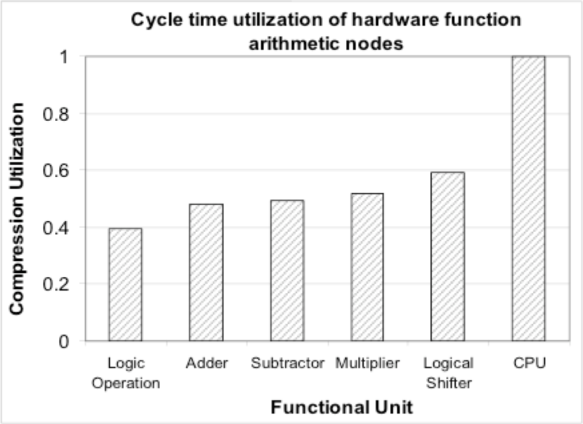


Figure 46: Cycle time utilization of hardware function arithmetic nodes as normalized to the cycle time of a 167 MHz processor on an Altera EP2S180.

6.4 RESULTS

The final output of the entire compilation flow is synthesizable VHDL. In Chapter 7 the performance results of different benchmarks is shown. These results were found by generating the VHDl through the compiler flow and finding the delay required for each hardware function. If we look at a simple segment of code it a easier to understand how the hardware is created. Figure 47 shows an example of a simple segment of code that has been marked for hardware creation. The actual benchmarks contain much larger segments of code. The resulting CDFG can been seen in Figure 48. The generated VHDL can be seen in Figure 49. The VHDL in Figure 49 does not show the generated components.

Each of the generated VHDL entities was synthesized with both Design Compiler and Synplify Pro. The results from the synthesis can be seen in Table 5. The Synplify Pro target was the Altera Stratix EP1S80F1020C Device. The Design Compiler synthesis target was 160nm OKI ASIC standard cells. The RFID primitive was synthesized to the Xilinx Spartan 3 xc3s400-4pq208 device, because the project targets low-power FPGAs.

```
int main(int x) {
  int y = 0;

#pragma HWstart;
    if(x==0) {
      y++;
    } else {
      y+=2;
    }
#pragma HWend;

  return y;
}
```

Figure 47: Simple Source Code Marked for Hardware Creation.

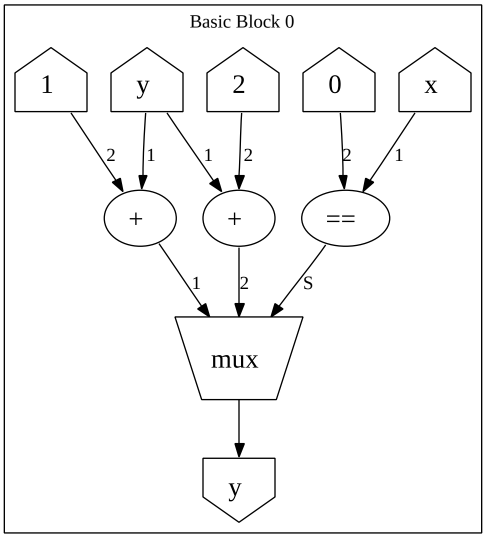


Figure 48: CDFG for the Simple Source Code from Figure 47.

```
entity main is
port (
signal x: IN signed(31 DOWNTO 0);
signal y_out: OUT signed(31 DOWNTO 0);
signal y: IN signed(31 DOWNTO 0));
end main;
architecture behavior of main is
signal sig2: signed(31 DOWNTO 0);
constant con4: signed(31 DOWNTO 0):= "00000000000000000000000000000000";
signal sig14: signed(31 DOWNTO 0);
signal sig6: signed(31 DOWNTO 0);
constant con8: signed(31 DOWNTO 0):= "00000000000000000000000000000001";
signal sig10: signed(31 DOWNTO 0);
constant con12: signed(31 DOWNTO 0):= "00000000000000000000000000000010";

component MUX_S_32
port (
signal A: IN signed(31 DOWNTO 0);
signal B: IN signed(31 DOWNTO 0);
signal S: IN signed(31 DOWNTO 0);
signal C: OUT signed(31 DOWNTO 0));
end component;
component add
port (
signal A: IN signed(31 DOWNTO 0);
signal B: IN signed(31 DOWNTO 0);
signal C: OUT signed(31 DOWNTO 0));
end component;
component is_equal_to
port (
signal A: IN signed(31 DOWNTO 0);
signal B: IN signed(31 DOWNTO 0);
signal C: OUT signed(31 DOWNTO 0));
end component;
begin
I0:is_equal_to
port map (A => x, B => con4, C => sig2);
I1:MUX_S_32
port map (A => sig6, B => sig10, S => sig2, C => sig14);
I2:add
port map (A => y, B => con8, C => sig6);
I3:add
port map (A => y, B => con12, C => sig10);
process (sig14) begin
y_out <= sig14;
end process;
end behavior;
```

Figure 49: Generated VHDL for the Simple Source Code from Figure 47.

Table 5: Hardware function synthesis results.

Hardware Unit	Synplify Pro Fmax (MHz)	 LC Utilization	Design Compiler Delay (ns)
ADPCM Encoder	51.7	0.87%	16
ADPCM Decoder	72.6	0.58%	9
G.721	39.4	5.50%	–
GSM	46.9	0.29%	20
GSM Unrolled	18.9	3.07%	–
IDCT Column	45.1	1.37%	22
IDCT Row	47.7	1.26%	20
RFID Primitive*	81.0	0.002%	–

7.0 PERFORMANCE RESULTS

To implement the architecture including the hardware functions, several industry computer-aided design tools were used to accomplish the functional testing and gate-level synthesis tasks of the design flow. Mentor Graphics' FPGA Advantage 6.1 was used to assist in the generation of VHDL code used to describe the core processor architecture. The processor was built up to support all of the operations described in the NIOS II instruction set.

Synplicity's Synplify Pro 7.6.1 was used as the RTL synthesis tool to generate the gate-level netlist targeted to the Altera Stratix II ES2S180F1508C4 from the VHDL description. The netlist was then passed to Altera's Quartus II 4.1 for device specific, placement and routing, and bitstream generation. At this level, post placement and routing results were extracted for additional manual optimization, timing accurate simulation, and verification. It is at this point that the timing information about the hardware functions can be inserted into the software. Both Altera's Quartus and Synplicity's Amplify for FPGA allow manual routing modifications for optimization of design density and critical paths.

For functional simulation and testing of the design, the machine code output was passed from the compiler design flow into the instruction ROM used in modeling the design. ModelSimSE 5.7 was used to generate the waveforms to confirm the functional correctness of the VLIW processor with hardware function acceleration.

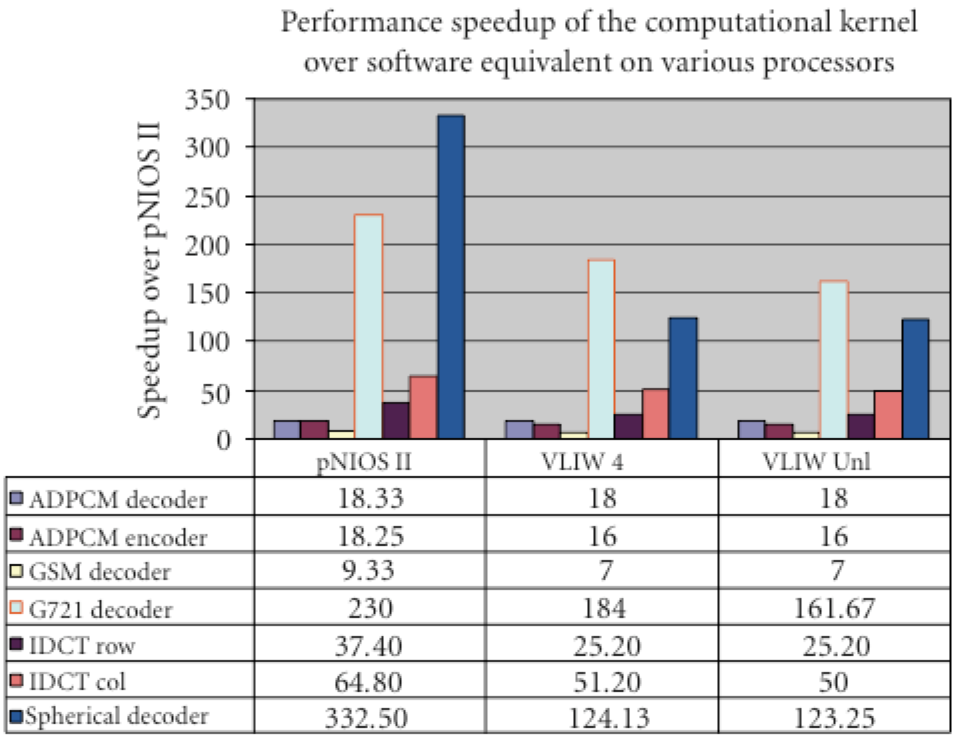
Through a series of optimizations to the critical path, a maximum clock speed of 166 MHz for the VLIW and clock frequencies ranging from 22 to 71 MHz for the hardware functions equating to combinational delays from 14 to 45 ns was achieved. Then, the benchmark execution times of the VLIW both with and without hardware acceleration against the pNIOS II embedded soft-core processor were compared.

To exercise the processor, core signal processing benchmarks from the MediaBench suite were selected. These include ADPCM encode and decode, GSM decode, G.721 decode, and MPEG 2 decode. As described in Chapter 6, from each of the benchmarks a single hardware kernel was extracted with the exception of MPEG 2 decode for which two kernels were extracted. In the case of GSM and G.721, the hardware kernel was shared by the encoder and decoder portions of the algorithm.

The performance improvement of implementing the computational portions of the benchmark on a 4-way VLIW, an unlimited-way VLIW, and directly in the hardware compared to a software implementation running on pNIOS II is displayed in Figure 50. The VLIW performance improvements were fairly nominal ranging from 2% to 48% improvement over pNIOS II, a single ALU RISC processor. The performance improvement of the entire kernel execution is compared for a variety of different architectures in Figure 51. The difference between Figures 50 and 51 is that the loads and stores required to maintain the data in the register file are not considered in the former and run in software in the latter. When the software-based loads and stores are considered, the VLIW capability of the processor has a more significant impact. Overall kernel speedups range from about 5X to over 40X.

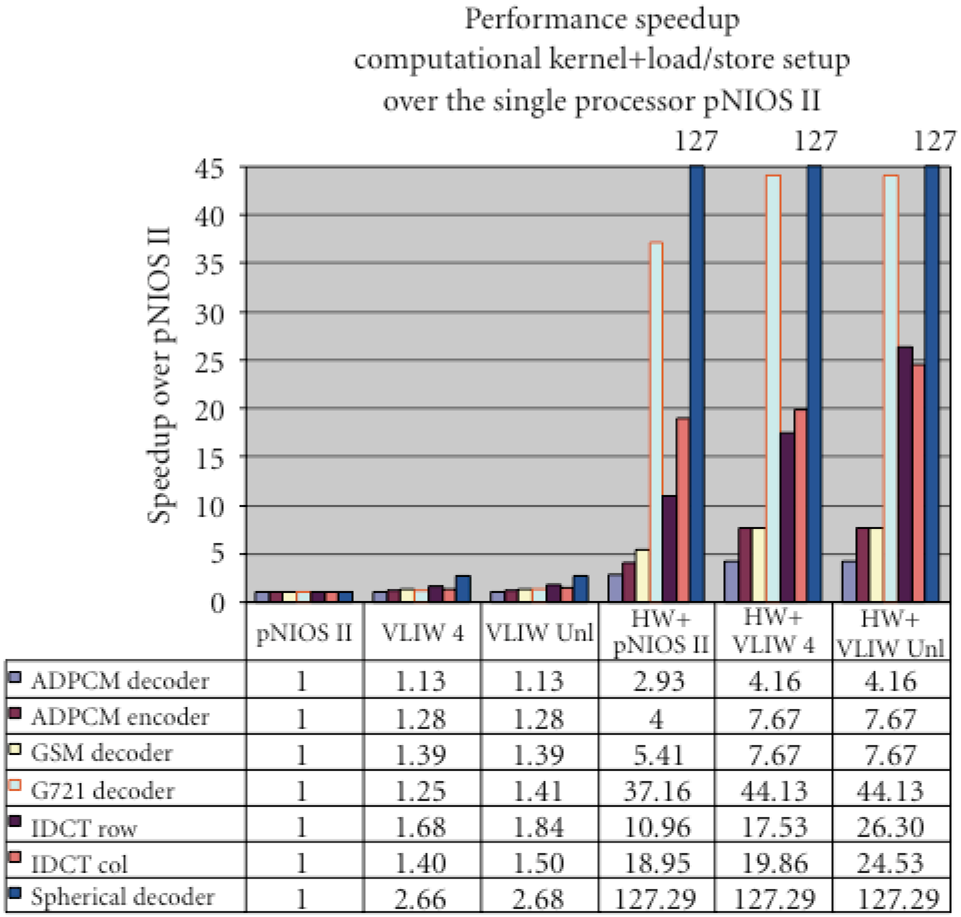
The width of the available VLIW has a significant impact on the overall performance. In general, the 4-way VLIW is adequate, although particularly when considering the IDCT-based benchmark, the unlimited VLIW shows that not all of the ILP available is being exploited by the 4-way VLIW.

The performance of the entire benchmark is considered in Figure 52. The execution times for both hardware and VLIW acceleration was considered and compared to the pNIOS II processor execution. The overheads associated with hardware function calls are included. While VLIW processing alone again provides nominal speedup (less than 2X), by including hardware acceleration, these speedups range from about 3X to over 12X and can reach nearly 14X when combined with a 4-way VLIW processor. These results have been published in [49].



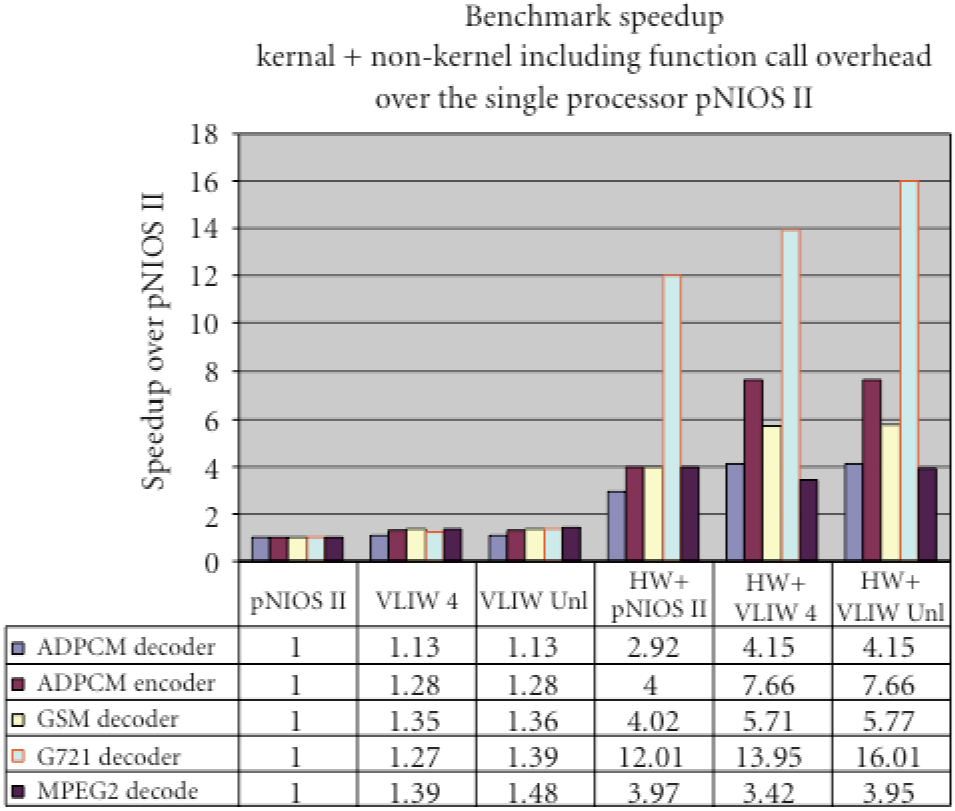
	pNIOS II	VLIW 4	VLIW Unl
■ ADPCM decoder	18.33	18	18
■ ADPCM encoder	18.25	16	16
□ GSM decoder	9.33	7	7
□ G721 decoder	230	184	161.67
■ IDCT row	37.40	25.20	25.20
■ IDCT col	64.80	51.20	50
■ Spherical decoder	332.50	124.13	123.25

Figure 50: Performance improvement from hardware acceleration of computational portion of the hardware kernel.



	pNIOS II	VLIW 4	VLIW Unl	HW+ pNIOS II	HW+ VLIW 4	HW+ VLIW Unl
■ ADPCM decoder	1	1.13	1.13	2.93	4.16	4.16
■ ADPCM encoder	1	1.28	1.28	4	7.67	7.67
□ GSM decoder	1	1.39	1.39	5.41	7.67	7.67
□ G721 decoder	1	1.25	1.41	37.16	44.13	44.13
■ IDCT row	1	1.68	1.84	10.96	17.53	26.30
■ IDCT col	1	1.40	1.50	18.95	19.86	24.53
■ Spherical decoder	1	2.66	2.68	127.29	127.29	127.29

Figure 51: Kernel speedups several architectures when considering required loads and stores to maintain the register file.



	pNIOS II	VLIW 4	VLIW Unl	HW+ pNIOS II	HW+ VLIW 4	HW+ VLIW Unl
■ ADPCM decoder	1	1.13	1.13	2.92	4.15	4.15
■ ADPCM encoder	1	1.28	1.28	4	7.66	7.66
□ GSM decoder	1	1.35	1.36	4.02	5.71	5.77
□ G721 decoder	1	1.27	1.39	12.01	13.95	16.01
■ MPEG2 decode	1	1.39	1.48	3.97	3.42	3.95

Figure 52: Overall performance speedup of the entire application for several architectures including overheads associated with function calls.

8.0 REDUCING POWER AND DELAY ELEMENTS

Power consumption in integrated microsystems is increasing at an alarming rate and has led to significant study of low power hardware design techniques [59] [60] [61] [62] [63] [64] [65] [66] [67]. As the market needs for smaller yet capable portable devices grow, integrated designers are struggling with two controversial goals to reduce power while increasing the amount of on-chip logic.

Low power embedded microprocessors such as ARM have long been the core of many portable devices such as portable digital assistants (PDAs) and cellular phones. Recently, several new low-power design techniques for application-specific instruction set architecture processors (ASIPs) have been proposed [68] [69] [70] [71] [72]. Companies like CoWare [73] and Tensillica [74] [75] also claim to have power-optimized ASIPs.

However, as embedded processing moves toward multiple processing cores on the chip, this raises serious questions about power management and consumption in the device. Traditional processor cores contain significant amounts of overhead to retain general purpose execution capability such as highly general ALUs, instruction decoders, register files, and data paths. Until recently, these components have only been optimized to increase performance of the processor. It is also likely that by having multiple processor cores on the same device, there will be additional overheads involved with communication between these devices either through interconnect or shared memory resources. All of these overheads compound the power impact rather than improving this issue.

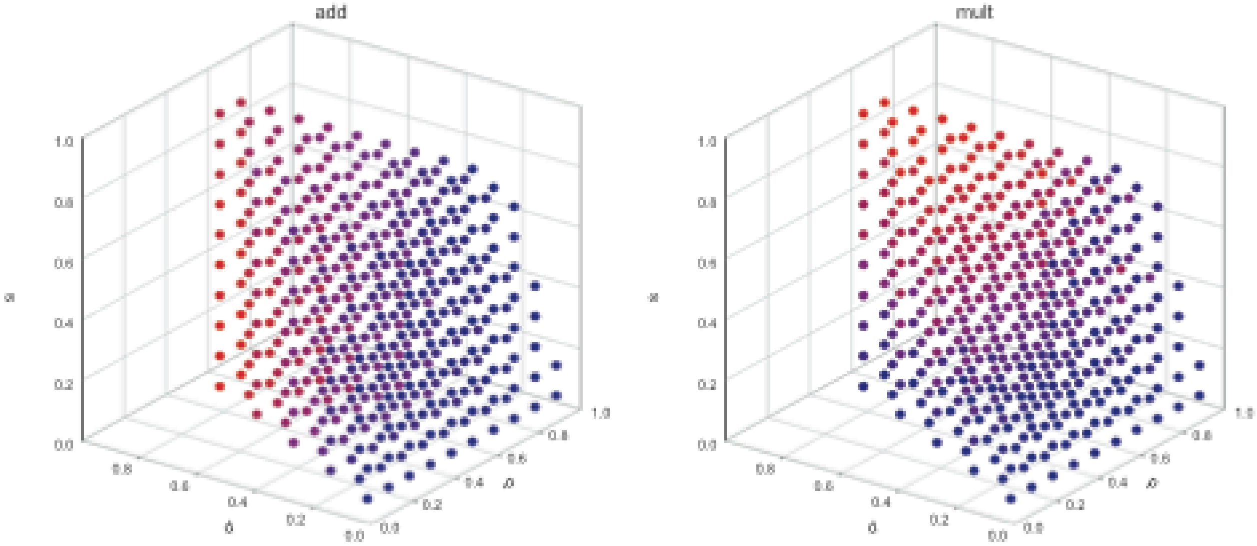
Described in Chapter 7, the SuperCISC hardware functions have a significant benefit for the performance. In this section, the advantage of SuperCISC hardware function execution for power and energy reduction in the processor. Section 8.1 describes the strategy used for analyzing different components of the processor core and in particular the SuperCISC hardware functions. In Section 8.2 the impact of SuperCISC hardware functions on the power and energy consumed by the computational kernels from the benchmarks is detailed. Both of these section can are from previously published work and can be seen in [76]. Lastly, in Section 8.3 a technique to prevent early switch of signals and the work done to support the changes to the compilation flow is discussed.

8.1 POWER MODELING AND ANALYSIS

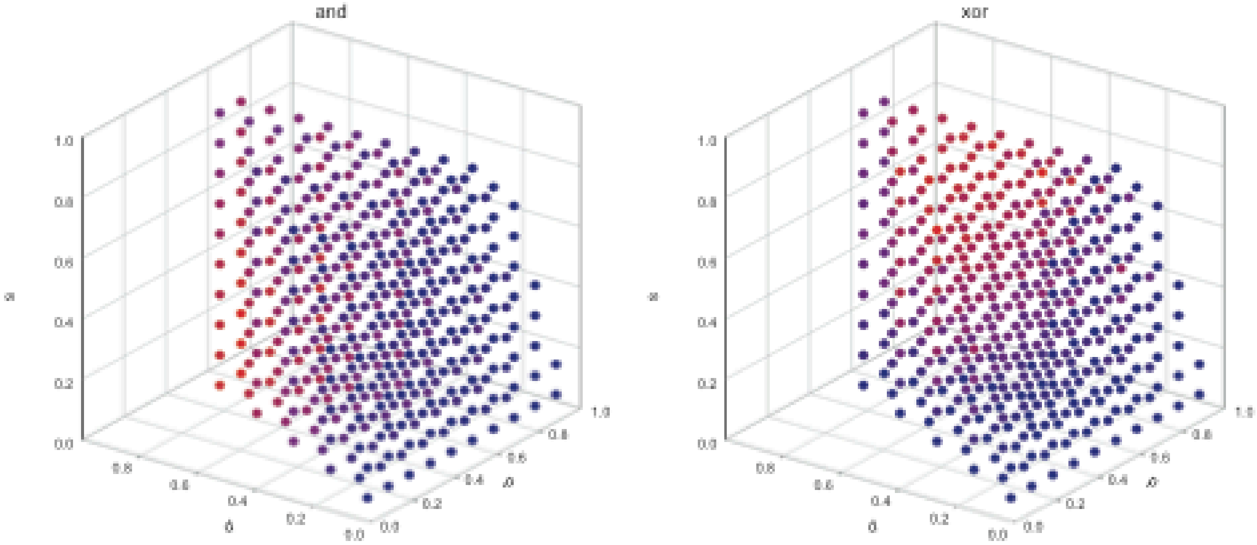
Several techniques have been proposed for power macromodeling to support high level and static power analysis of digital circuits [77]. In particular, the work found in [78] [79] [80] suggest three parameters for accurate estimation of power within digital circuits: average input signal probability p, average input transition density d, and input spatial correlation s. In a technique similar to [81] several types of functional units and other types of devices found in our system have been profiled, such as decoders and multiplexers, with different values of p, d, and s.

The power consumption for several ALU operations synthesized with standard cells for a 160nm OKI ASIC process is displayed graphically in Figure 53. These charts plot p, d, and s versus power. Power is indicated as a color between red and blue where solid blue represents the least power consumed by the device and red indicates the most power consumed by the device. Measurements were taken at 0.1 intervals in each dimension p, d, and s. The actual input vectors for power simulation were generated using the technique described in [80]. The synthesis was executed with Design Compiler and the power was estimated using PrimePower; both tools are from Synopsys [82]. It should be noted that several combinations for p, d, and s are not possible which is the reason for the wedge-like shape.

While transition density (d) is often considered the only metric of interest, Figure 53 presents an indication of how this can be insufficient. The adder in Figure 53(a) shows that the most power is consumed when spatial correlation is low and transition density is high. The AND function in Figure 53(c) is even more sensitive to spatial correlation than the adder.



(a) Adder (b) Multiplier



(c) AND Operation (d) XOR Operation

Figure 53: 4-D plots of p, d, and s versus power for several ALU operations synthesized with standard cells for a 160nm OKI ASIC process. Power is indicated as a color between red and blue where solid blue represents the least power consumed by the device and red indicates the most power consumed by the device. Measurements taken at 0.1 intervals in each dimension p, d, and s.

The multiplier in Figure 53(b) is exactly the opposite, showing higher power consumption with high spatial correlation. Finally, the XOR operation shown in Figure 53(d) requires the most power with high spatial correlation with intermediate switching density.

To better understand the impact of power in the SuperCISC hardware functions, several architectural techniques for implementing the computation were profiled. The results of this profile are displayed in Figure 54. These results are for hardware synthesized using standard cells for a 160nm OKI ASIC process. The *Simple* bar corresponds to a completely independent block written directly in VHDL and synthesized for each operation, ALU is a synthesizable ALU built structurally from components using the Mentor Moduleware Component Library without considering power consumption as a design metric. The *Component* bar synthesizes individually each operation from the larger ALU. Finally, the *Opt.* ALU is a low power ALU design where components of the design were synthesized, profiled individually and the results were combined together. Each of *Simple*, *Component*, *Opt.* ALU, and ALU were profiled for power running each operation (e.g. addition, xor, etc) independently. The resulting power number is taken by computing the average of multiple power simulations that cover the entire space of p, d, and s with discrete 0.1 intervals.

From Figure 54 it appears that the non-power optimized ALU is switching all of the different hardware blocks irrespective of which operation is being executed. For example, during a XOR operation, it appears that the multiplier, adder, etc all appear to be doing work even though the value is not utilized. While the Moduleware blocks consume more power than the equivalent *simple* directly synthesizable VHDL blocks, this overhead is in the 10-20% range. The optimized ALU does seem to only switch the operation being executed (e.g. such as the multiply case), however, there appears to be an approximately 50μW overhead associated with additional latches, decoding logic, and multiplexing over the individual Moduleware blocks. This information is useful to providing insight into the power impact of SuperCISC functionality. The generation of the power results are contributed to this thesis through the work of Gayatri Mehta. These results were previously published in [76].

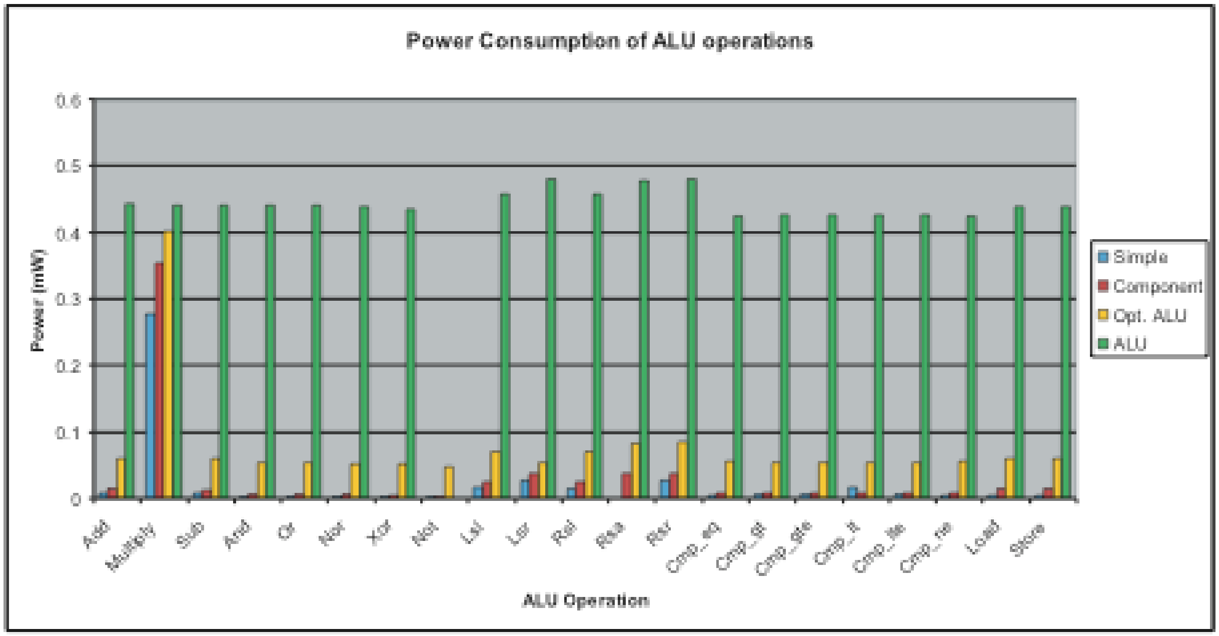


Figure 54: Power profile of several different functional unit implementation techniques. Results for a standard cell 160nm OKI ASIC process.

8.2 POWER IMPACT OF SUPERCISC FUNCTIONS

In Sections 5.3 and 6.3, the impact of converting control-flow into dataflow for combinational implementation and the effect of cycle-compression, respectively for performance improvement using SuperCISC hardware functions is described. In this section, the impact of the SuperCISC strategy on power consumption is discussed.

In Section 6.3, it was shown that an *if-then-else* construct requires multiple assembly instructions in the processor but requires only a comparator and multiplexer in hardware. This type of operation reduction significantly reduces the power consumed by hardware. One tradeoff in this respect is the need to potentially perform *additional* work due to parallel execution not required by a sequential processor. However, the benchmarks have shown that conditionals impacted by this sort of predication tend to have very short *then* and *else* clauses making this overhead small compared to the savings of reducing the number of operations.

There appears to be a power analog to cycle compression that was shown in Section 6.3 to help performance. Figure 54 shows that there is a significant power impact of executing ALU operations on different types of structures. The ALU optimized for performance requires approximately an order of magnitude more power than the individual ALU components synthesized separately. In some cases, even the overhead from the power-friendly ALU is an order of magnitude more expensive than the individual operations. Thus, when a hardware function is implemented directly in hardware the *power compression* of these ALU operations can become significant. *Power compression* is defined as the reduction in power due to the extraction of the logic or arithmetic operation of interest from an entire ALU.

Figure 55 shows the impact of this power compression. The power required by all operations other than multiplication is less than 5% of the total power required by the ALU and the multiplier is only slightly over 60% of the total ALU power requirement.

Finally, several other factors contribute to the SuperCISC hardware function's potential for power savings. First, the elimination of a clock tree and clocked registers from the hardware execution can provide significant power savings over execution in the processor which accesses three registers each cycle. Also, the removal of other control stages that are

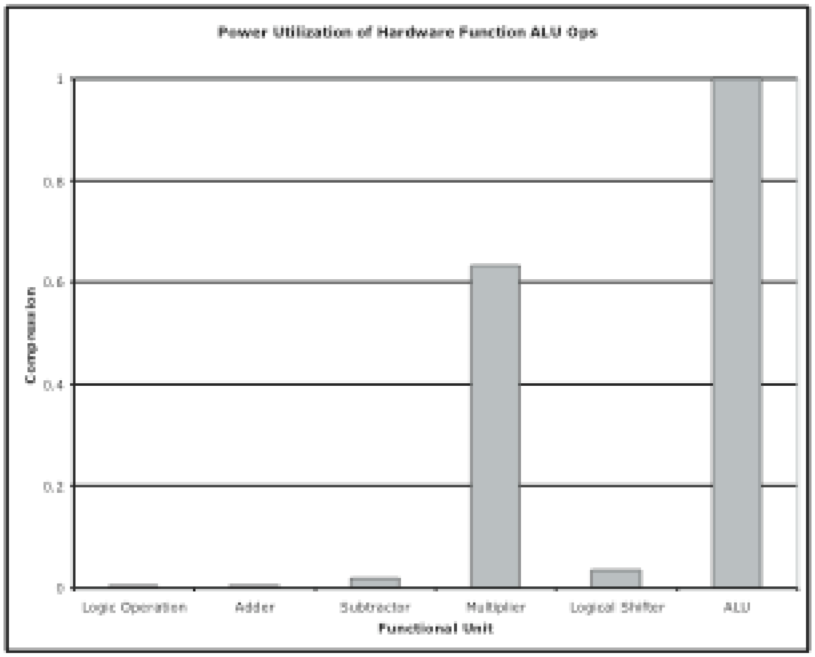


Figure 55: Power compression of various classes of ALU operations.

required by a processor such as instruction fetch and decode provides additional opportunity for savings. While, the SuperCISC may be executing many operations in parallel, it can eliminate many of these overheads required by the processor.

The power consumption of the computationally complex kernels implemented in a SuperCISC hardware function and executed on the VLIW processor is displayed in Table 6. This is graphically displayed as times improvement in Figure 56. For these computational kernels we are seeing a power improvement of between just under 50X and over 400X for the SuperCISC hardware function with an average of just over 130X improvement.

Figure 56 shows the power savings along with the performance improvements for the benchmark suite. As both power and performance are improving simultaneously, the overall energy improvement should be significant. The energy required by the SuperCISC hardware functions compared to the VLIW processor is shown in Table 6 and the improvement is shown graphically in Figure 57. The energy improvement is several orders of magnitude for the SuperCISC hardware functions ranging from over 1000X to approximately 60000X.

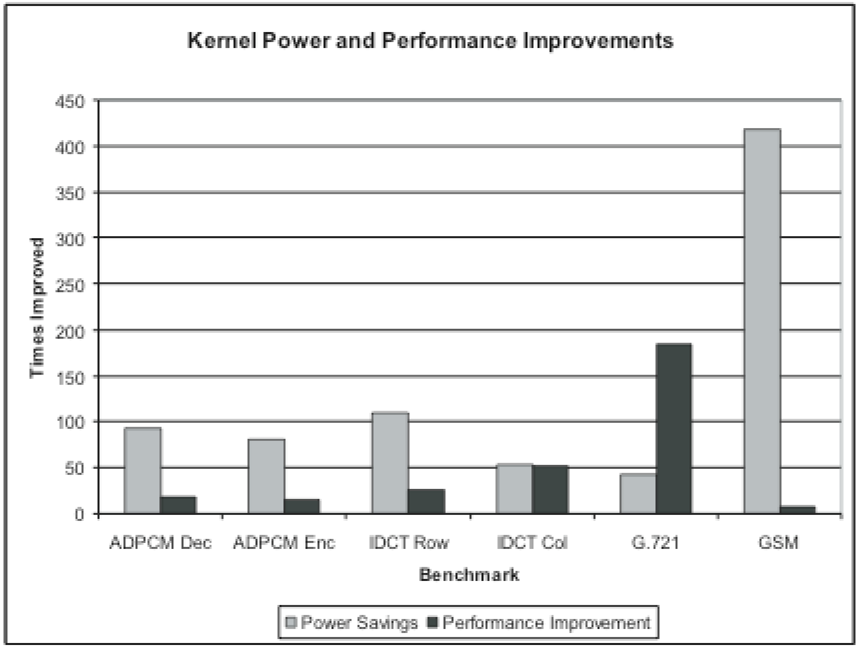


Figure 56: Power reduction shown with performance improvement for benchmark suite.

Table 6: Power and energy required by the VLIW processor core and the SuperCISC hardware functions for computational kernels after 160nm OKI standard cell ASIC synthesis.

	SuperCISC		pNIOS II VLIW	
Benchmark	Power (mW)	Energy (μJ)	Power (mW)	Energy (μJ)
ADPCM Dec	7.53	1.16	701.20	2378.97
ADPCM Enc	8.59	3.69	695.60	6264.6
IDCT Row	6.47	2.96	708.70	43701.68
IDCT Col	13.34	13.07	708.50	43701.68
G.721	16.71	256.37	701.80	3442613.83
GSM	1.69	101.66	705.20	421131.22

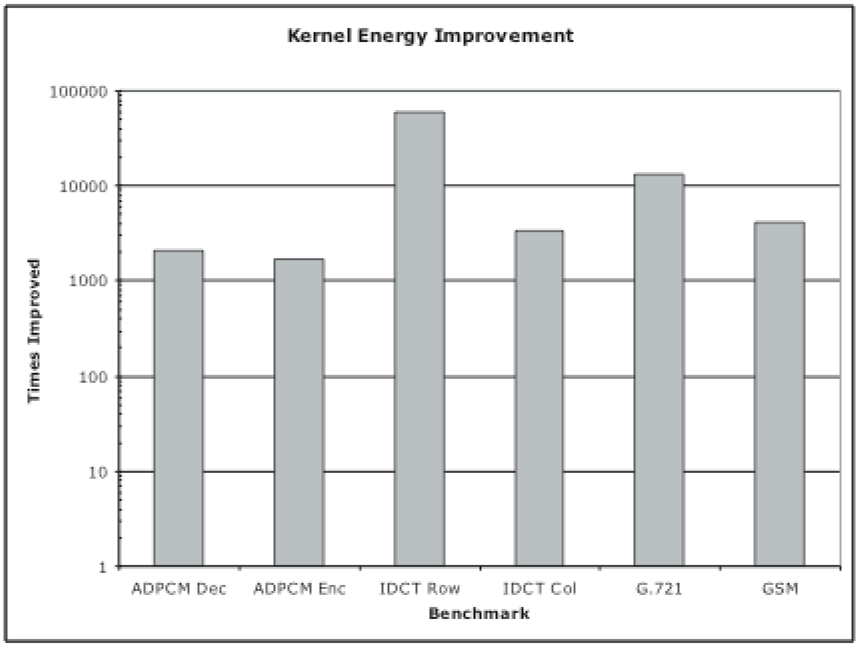


Figure 57: Energy improvement of SuperCISC over VLIW for benchmark suite.

The results presented here are analyses of the computational kernel portion of the benchmark codes. However, while this savings does apply to the bulk of the executing code, it is certainly not applied to the entire application. The savings at theapplication-level are approaching the limit of allowed savings by Ahmdal's law. The generation of the power compression results are contributed to this thesis through the work of Gayatri Mehta. These results were previously published in [76].

8.3 DELAY ELEMENT AND LATCH GENERATION

Combination hardware functions have the potential to consume much power due to the glitching of signals in the logic path. Glitching refers to the unwanted change of a function's output because of differing arrival times in the circuit. For example, if an adder and a multiplier feed into a subtracter, the adder's output, taking less time to complete, will

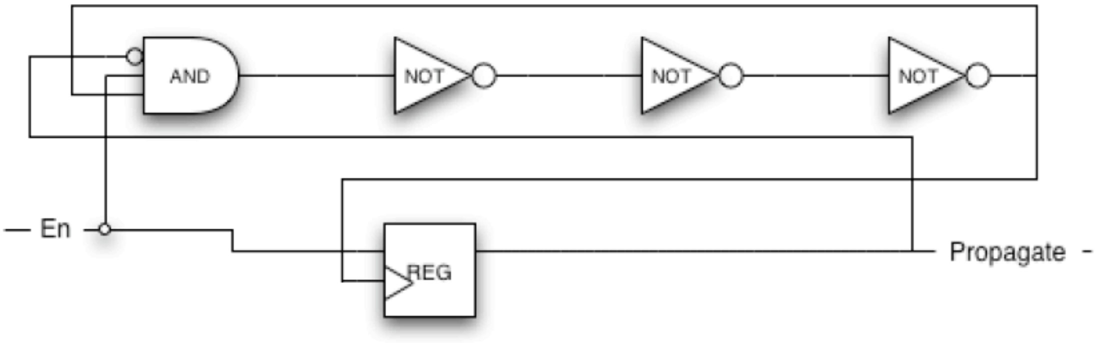


Figure 58: Proposed Delay Element.

arrive before the multiplier is finished. In this case the subtracter will begin to subtract the stabilized result from the added with the unstable results of the multiplier. This is the definition of glitching.

A solution to this problem has been presented. The concept is to place delay elements in the combinational path that prevent the inputs to some combination logic from glitching. The input to a delay element is an enable signal that is a single bit and arrives with the data. This bit drives an inverter chain and produces a propagate signal after the bit has toggled through the chain. Once the propagate signal becomes high the inverter chain stops switching. The proposed logic for the delay element can be seen in Figure 58. The propagate signal drive a level sensitive latch which allow switch to occur after the rising edge of its clock.

The delay element has been synthesized in 160nm ASIC technology. Figure 59 shows the linear function derived from the results of different sized elements. Figure 60 shows the associated power and energy cost for the delay element. After these calculations were completed the cost of adding a delay element to the combinational logic was modeled. A static timing was was created by another student to determine the best possible location for delay elements based on the 160nm technology process. This work is still in the early stages.

The static timing pass creates annotations to the CDFG marking where delay nodes are to be inserted. A separate pass was created to insert the delay nodes and latches into the

$$delay = 0.262 * n_{inv} + 0.506ns$$

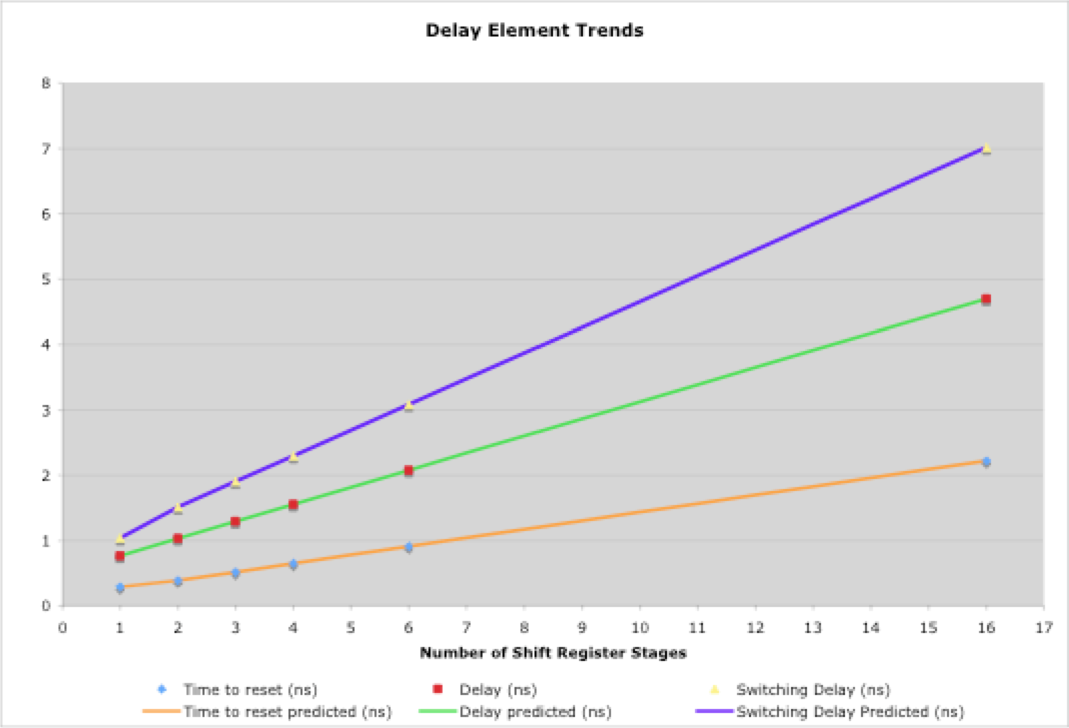


Figure 59: Proposed Delay Element's Delay Function.

$$Energy = 0.289 * n_{inv} + 0.561 \mu J$$

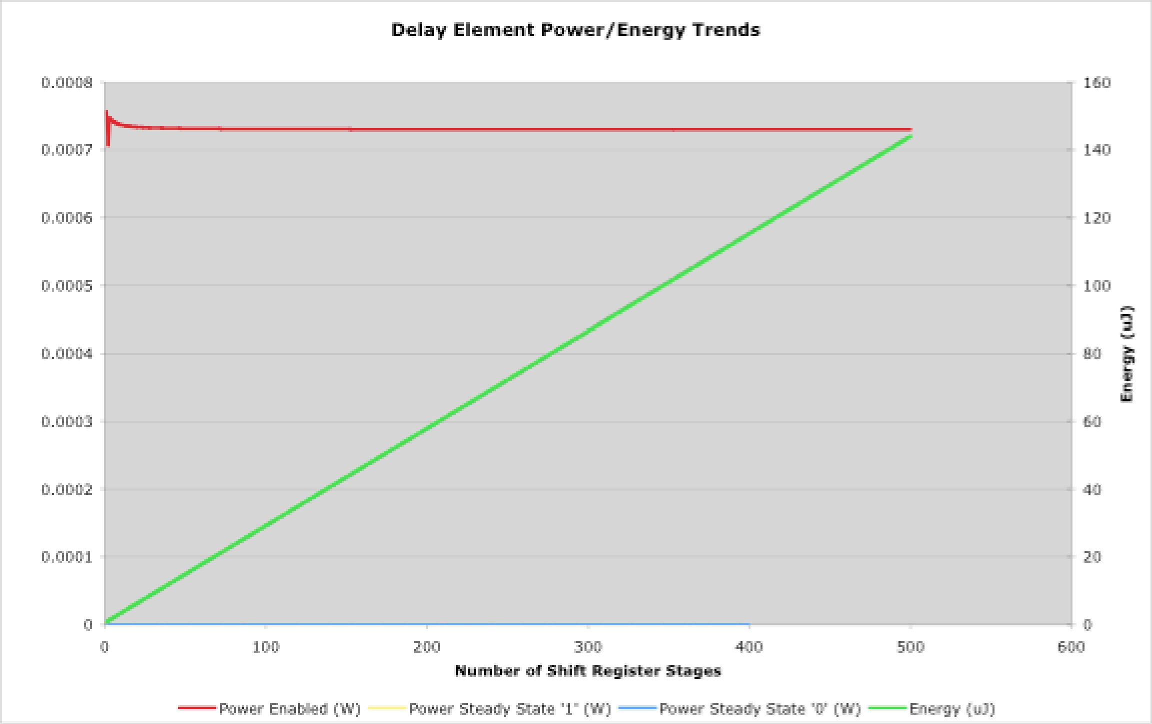


Figure 60: Proposed Delay Element's Power and Energy Cost.

CDFG. This pass provides users with a simple examples of how to interface with the CDFG IR. The pass read an annotation to the CDFG, creates a new node, and inserts the node into the middle of the graph.

The VHDL generation pass was modified to account for the new node types. VHDL has been generated for a delayed graph and synthesized. The resulting power saving are not yet available because the calculation to determine delay node placement is still being tested.

9.0 CONCLUSIONS

9.1 CONTRIBUTIONS

This thesis describes the VLIW / SuperCISC processor architecture developed for comparison to multi-core embedded processor architectures. The proposed processor architecture offers a two-fold attack to solving the problem of maximizing Instruction-Level Parallelism, ILP. The architecture was excised using a compilation flow that was extended from the VLIW compiler project, Trimaran. The VLIW processing is shown to achieve a maximum 2X application speedup when given one hundred processors to schedule instructions on. However, when superCISC hardware functions are used to accelerate the computational kernel's of the application, the VLIW has a significant impact providing, in some cases, up to an additional 3.6X. It provided an average of 2.3X over a single processor and hardware alone. This range falls within the additional predicted 2X to 5X processing capability provided by the 4-way VLIW processor.

To create superCISC hardware functions, a compilation flow from C/C++ to VHDL was created. Only segments are code are marked for translation into VHDL. The compiler first creates a control and data flow graph (CDFG) intermediate representation (IR) of the source code. A CDFG provides the framework needed to explore different compiler transformations. The IR that was created for the compiler transformations for hardware generation, however, it is currently being used to investigate the mapping of hardware onto a course-grain computational fabric, and also to examine parallel processing benchmark applications in an attempt to resolve communication patterns through inter-procedural variable resolution.

In order to use a shared register file to transfer data between the VLI W processor and the superCISC hardware functions the architecture must be limited to the number of hardware functions. If a bus is used to connect the processor and hardware functions, there can be many more hardware functions, however, a bus requires communication overhead. To solve this problem hardware functions of the greatest possible size are created. The hardware functions are entirely combinational logic and therefore control flow boundaries require a break in the function. In addition loading and storing data to the main memory also requires a break. To solve the control flow problem, several standard compiler transformations are proposed. These transformation include function inlining, loop unrolling, and code motion. If addition a technique called hardware predication has been developed. Hardware predication takes *if-then* and *if-then-else* control flow constructs and transforms them into data flow through the use of multiplexors. Through the use of hardware predication the control flow bound kernels are able to create large hardware functions.

After the superCISC hardware function have been transformed by the compiler only a single large data flow graph with predication exists. The last compilation transformation takes the CDFG IR and creates a hardware description language file. The VHDL generation pass translates the compiler data structures into synthesizable VHDL. After the VHDL has been synthesized into an Altera Stratix II ES2S180F1508C4 FPGA and connected to the VLIW processor the system was tested. The computationally intensive software kernels, which are typically called by the application many times, show an average increase in performance of 63X. Some benchmark kernels see improvements of over 300X. The entire application execution was also measured. The hardware accelerated VLIW processor showed performance speedups of 12X on average over a single processor implementation. The highest application speedup reached nearly 30 times the execution of the single processor setup.

The final contribution to this thesis is an investigation into the power requirements of the architecture. An reduction in the power consumption of the architecture is expected because the power required for processor execution is generally much higher than custom hardware for the same technology. The processor was synthesized in a 160nm ASIC process and the benchmark kernels were analyzed for power. The result was compared to the same kernel's power for the generated hardware synthesized in the same 160nm ASIC process. The power requirements were compared, and the required energy was calculated based on the timing information from the ASIC synthesis. For the computational kernels the power improvement of the hardware execution is between just under 50X and over 400X with an average of just over 130X improvement. The energy improvement seen by the hardware execution combines the power efficiency and the decrease latency and therefore is several orders of magnitude lower for the hardware functions. The energy savings range from over 1,000X to approximately 60,000X. The automation of the hardware functions allows for the hardware to include possible power saving elements. The concept of a delay element, which delays the switching of a signal based on static timing of the path was theorized. The ability to add additional hardware for power savings was added to the compilation flow.

9.2 FUTURE RESEARCH

The initial results from the VLIW-SuperCISC architecture are very encouraging. Currently a significant bottleneck in the architecture is memory loads and stores. The number and size of the hardware functions is limited to the values being passed through a shard register file. To be able to read and write from a section of main memory that is used by both the hardware and the processor would increase performance greatly. This, however, would require significant changes to the architecture and the design flow.

There are also several aspects of the compiler that can benefit from continued improvement. Optimizations to the CDFG can be implemented to see an immediate reduction in the size of the hardware generated. Hardware structures such as d encoders can be created from switch statements or for loops. A decoder would allow the hardware to have a much greater ILP.

Lastly, during my work with the project the applications tested were almost exclusively multimedia based benchmarks. Testing other benchmarks for speedups would should that the VLIW-SuperCISC concept can improve on all types of C applications.

BIBLIOGRAPHY

[1] G. D. Micheli, D. Ku, F. Mailhot, and T. Truong, "The olympus synthesis system for digital design," *IEEE Design & Test of Computers*, 1990.

[2] L. Lavagno and E. Sentovich, "Ecl: a specification environment for system-level design," in *Proceedings of the 36th Design Automation Conference (DAC 99)*, 1999, pp. 511–516.

[3] S. Gupta, N. D. Dutt, R.K.Gupta, and A.Nicolau, "Spark : A high-level synthesis framework for applying parallelizing compiler transformations," in *International Conference on VLSI Design*, 2003.

[4] S. Gupta, N. Savoiu, N. D. Dutt, and A. N. R. K. Gupta, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions on Computer Aided Design*, Februrary 2004.

[5] A. K. Jones, D. Bagchi, S. Pal, P. Banerjee, and A. Choudhary, *Pact HDL: Compiler Targeting ASIC's and FPGA's with Power and Performance Optimizations*, R. Graybill and R. Melhem, Eds. Boston, MA: Kluwer Academic Publishers, 2002.

[6] X. Tang, T. Jiang, A. Jones, and P. Banerjee, "Compiler optimizations in the pact hdl behavioral synthesis tool for asics and fpgas," in *IEEE International SoC Conference (IEEE-SOC)*, September 2003.

[7] E. Jung, "Behavioral synthesis using systemc compiler," in *Proceedings of 13th Annual Synopsys Users Group Meeting (SNUG 2̆01903)*, March 2003.

[8] D. Black and S. Smith, "Pushing the limites with behavioral compiler," in *Proceedings of 9th Annual Synopsys Users Group Meeting (SNUG 2̆01999)*, March 1999.

[9] K. Bartleson, "A new standard for system-level design," Synopsys White Paper, 1999.

[10] R. Goering, "Behavioral synthesis crossroads," EE Times Article, 2004.

[11] D. J. Pursley and B. L. Cline, "A practical approach to hardware and software soc tradeoffs using high-level synthesis for architectural exploration," in *Proceedings of the GSPx Conference*, March-April 2003.

[12] S. Chappell and C. Sullivan, "Handel-c for co-processing and co-design of field programmable system on chip," Celoxica White Paper, 2002.

[13] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. Uribe, "Overview of a compiler for synthesizing matlab programs onto fpgas," *IEEE Transactions on Very large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 312–324, 2004.

[14] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A matlab compiler for distributed, heterogeneous, reconfigurable computing systems," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2000, pp. 39–48.

[15] S. McCloud, "Catapult c synthesis-based design flow: Speeding implementation and increasing flexibility," Mentor Graphics, Tech. Rep., 2004.

[16] V. Chaiyakul and D. D. Gajski, "Assignment decision diagram for high-level synthesis," University of California, Irvine, Calif, USA, Tech. Rep. 92–103, December 1992.

[17] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntactic variances," in *Proceedings of 30th Design Automation Conference (DAC '93)*, June 1993, pp. 413–418.

[18] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional rtl circuits using assignment decision diagrams," in *Proceedings of 37th Design Automation Conference (DAC '00)*, June 2000, pp. 43–48.

[19] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential atpg for functional rtl circuits," in *Proceedings of IEEE International Test Conference (ITC '03)*, vol. 1, September-October 2003, pp. 290–298.

[20] V. A. Chouliaras and J. Nunez, "Scalar coprocessors for accelerating the g723.1 and g729a speech coders," *IEEE Transactions on Consumer Electronics*, vol. 69, no. 3, pp. 703–710, August 2003.

[21] E. Atzori, S. M. Carta, and L. Raffo, "44.6% processign cycles reduction in gsm voice by low-power reconfigurable co-processor architecture," *Electronics Letters*, vol. 38, no. 24, pp. 1524–1526, November 2002.

[22] J. Hilgenstock, K. Herrmann, J. Otterstedt, D. Niggemeyer, and P. Pirsch, "A video signal processor for mimd multiprocessing," in *Proceedings of the 1998 Design Automation Conference*, San Francisco, CA, June 1998.

[23] R. Garg, C. Chung, D. Kim, and Y. Kim, "Boundary macroblock padding in mpeg-4 video decoding using a graphics co-processor," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 719–723, August 2002.

[24] C. Hinds, "An enhanced floating point coprocessor for embedded signal processing and graphics applications," in *Conference Record of the 33rd Asilomar COnference on Signals, Systems and Computers*, Pacific Grove, CA, October 1999.

[25] J. C. Alves and J. S. Matos, "Rvc-a reconfigurable coprocessor for vector processing applications," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1998.

[26] T. Bridges, S. W. Kitchel, and R. M. Wehrmeister, "A cpu utilization limit for massively parallel mimd computers," in *Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[27] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "Piperench: A virtualized programmable datapath in 0.18 micron technology," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 2002.

[28] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: A reconfigurable architecture and compiler," *in IEEE Computer*, vol. 33, no. 4, April 2000.

[29] S. C. Goldstein, H. Schmit, M. Moe, and et al, "Piperench: a coprocessor for streaming multimedia acceleration," in *Proceedings of 26th IEEE International Symposium on Computer Architecture (ISCA 99)*, May 1999, pp. 28–39.

[30] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, "Managing pipeline-reconfigurable fpgas," in *Proceedings of 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 98)*, February 1998, pp. 55–64.

[31] H. Schmit, "Incremental reconfiguration for pipelined applications," in *Proceedings of 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM 97)*, April 1997, pp. 47–55.

[32] B. A. Levine and H. Schmit, "Efficient application representation for haste: Hybrid architectures with a single, transformable executable," in *IEEE Symposium on FPGAs for Custom Computing Machines(FCCM)*, 2003.

[33] C. Ebeling, D. C. Cronquist, and P. Franklin, "Rapid - reconfigurable pipelined datapath," in *in the 6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[34] C. Ebeling, D. C. Cronquist, P. Franklin, and C. Fisher, "Rapid - a configurable computing architecture for compute-intensive applications," University of Washington, Department of Computer Science and Engineering, Seattle, Wash, USA, Tech. Rep. TR-96-11-03, 1996.

[35] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg, "Mapping applications to the rapid configurable architecture," in *Proceedings of 5th Annual IEEE*

Symposium on FPGAs for Custom Computing Machines (FCCM 97), April 1997, pp. 106–115.

[36] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, “Specifying and compiling applications for rapid,” in *Proceedings of 6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM98)*, April 1998, pp. 116–125.

[37] D. C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, “Architecture design of reconfigurable pipelined datapaths,” in *Proceedings of 20th Anniversary Conference on Advanced Research in VLSI*, March 1999, pp. 23–40.

[38] E. Mirsky and A. Dehon, “Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources,” in *in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996.

[39] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, “The imagine stream processor,” in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, September 2002, pp. 282–288.

[40] B. Khailany, W. J. Dally, U. J. Kapasi, and et al., “Imagine: media processing with streams,” *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[41] J. D. Owens, S. Rixner, U. J. Kapasi, and et al., “Media processing applications on the imagine stream processor,” in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 295–302.

[42] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS processor with a reconfigurable coprocessor,” in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 12–21. [Online]. Available: citeseer.nj.nec.com/hauser97garp.html

[43] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “The garp architecture and c compiler,” *IEEE Computer*, vol. 33, April 2000.

[44] T. Callahan, “Kernel formation in garpcc,” in *Proceedings of 11th Annual IEEE Symposiumon FPGAs for CustomComputing Machines (FCCM 03)*, pp. 308–309.

[45] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, “The chimaera reconfigurable functional unit,” in *IEEE Symposium on FPGAs for Custom Computing Machines(FCCM)*, 1997, pp. 87–96.

[46] S. Hauck, M. M. Hosler, and T. W. Fry, “High-performance carry chains for fpgas,” in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 98)*, February 1998, pp. 223–233.

[47] R. Hoare, S. Tung, and K. Werger, “A 64-way simd processing architecture on an fpga,” in *IASTED International Conference on Parallel and Distributed Computing and Systems*, 2003.

[48] S. Dutta, A. Wolfe, W. Wolf, and K. O'Connor, "Design issues for very-long-instruction-word vlsi video signal processors," in *IEEE Workshop on VLSI Signal Processing*, 1996.

[49] R. Hoare, A. K. Jones, D. Kusic, J. Fazekas, J. Foster, S. Tung, and M. McCloud, "Rapid vliw processor customization for signal processing applications using combinational hardware functions," *EURASIP Journal on Applied Signal Processing*, 2005, in second review.

[50] A. Jones, R. Hoare, I. Kourtev, J. Fazekas, D. Kusic, J. Foster, S. Boddie, and A. Muaydh, "A 64way vliw/simd fpga processing architecture and design flow," in *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2004.

[51] A. Nene, S. Talla, B. Goldberg, H. Kim, and R. M. Rabbah, "Trimaran: An infrastructure for compiler research in instruction level parallelism," 1998.

[52] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu, "The impact epic 1.0 architecture and instruction set reference manual," IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-98-04, 1998.

[53] S. Aditya, V. Kathail, and B. R. Rau, "Elcor's machine description system: Version 3.0," HP Labs, Tech. Rep. HPL-98-128, 1998.

[54] L. N. Chakrapani, W. F. Wong, and K. V. Palem, "Enhancing the trimaran compiler infrastructure to support strongarm code generation," CREST, Georgia Institute of Technology, Atlanta, GA, Tech. Rep. CREST-TR-01-01, 2001.

[55] Y. Chobe, B. Narahari, R. Simha, and W.Wong, "Tritanium: Augmenting the trimaran compiler infrastructure to support ia64 code generation," The George Washington University, Washington DC.

[56] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[57] Optimizing with shark, big payoff, small effort. Apple Computer, Inc. [Online]. Available: http://developer.apple.com/tools/shark_optimize.html

[58] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, , and J. Foster, "An fpga-based vliw processor with custom hardware execution," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2005.

[59] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee, "Pact hdl: a c compiler targeting asics and fpgas with power and performance optimizations," in *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*. ACM Press, 2002, pp. 188–197.

[60] K. Roy and S. Prasad, *Low-Power CMOS VLSI Design.* John Wiley and Sons Inc., 2000.

[61] J.-M. Chang and M. Pedram, "Module assignment for low power," in *European Design Automation Conference*, 1996. [Online]. Available: citeseer.nj.nec.com/chang96module.html

[62] K. Khouri, G. Lakshminarayana, and N. Jha, "Impact: A highlevel synthesis system for low power control-flow intensive circuits," in *Proc. Design Automation & Test in Europe Conf.*, 1998, pp. 848–854. [Online]. Available: citeseer.nj.nec.com/khouri98impact.html

[63] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proceedings of ICCD*, October 1994, pp. 318–322.

[64] Z. X. Shen and C. C. Jong, "Exploring module selection space for architectural synthesis of low power designs," in *IEEE International Symposium on Circuits and Systems*, 1997.

[65] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. Brodersen, "Optimizing power using transformations," *IEEE Transactions on CAD*, vol. 14, no. 1, pp. 12–31, 1995. [Online]. Available: citeseer.ist.psu.edu/chandrakasan95optimizing.html

[66] E. Musoll and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," in *Proceedings of the International Symposium on Low-Power Design*, 1995, pp. 99–104. [Online]. Available: citeseer.nj.nec.com/musoll95highlevel.html

[67] N. K. Jha, "Low power system scheduling and synthesis," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design.* IEEE Press, 2001, pp. 259–263.

[68] J. eun Lee, K. Choi, and N. D. Dutt, "Energy-efficient instruction set synthesis for application-specific processors," in *Proceedings of ISLPED.* ACM, 2003.

[69] S. Chandar, M. Mehendale, and R. Govindarajan, "Area and power reduction of embedded dsp systems using instruction compression and re-configurable encoding," in *Proeedings of ICCAD*, 2001.

[70] L. Benini, A. Macii, E. Macii, and M. Poncino, "Selective instruction compression for memory energy reduction in embedded systems," in *Proceedings of the 1999 international symposium on Low power electronics and design.* ACM Press, 1999, pp. 206–211.

[71] J.-G. Cousin, O. Sentieys, and D. Chillet, "Multi-algorithm asip synthesis and power estimation for dsp applications," in *Proceedings of ISCAS*, 2000.

[72] C. Glokler and H. Meyr, "Power reduction for asips: A case study," in *Proceedings of the Wkshp. Signal Processing Systems (SIPS)*, 2001. [Online]. Available: citeseer.ist.psu.edu/article/glokler01power.html

[73] CoWare, "The lisatek solution: Automated embedded processor design and software development tool generation," CoWare, Inc.," Datasheet.

[74] R. E. Gonzalez, "Xtensa – a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[75] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems.* ACM Press, 2003, pp. 137–147.

[76] A. K. Jones, R. Hoare, D. Kusic, G. Mehta, J. Fazekas, and J. Foster, "Reducing power while increasing performance with supercisc," *ACM Transactions on Embedded Computing Systems*, p. 125, 2005.

[77] F. N. Najm, "A survey of power estimation techniques in vlsi circuits," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 2, no. 4, pp. 446–455, 1994.

[78] S. Gupta and F. N. Najm, "Power macromodeling for high level power estimation," in *DAC '97: Proceedings of the 34th annual conference on Design automation.* ACM Press, 1997, pp. 365–370.

[79] Z. Chen and K. Roy, "A power macromodeling technique based on power sensitivity," in *DAC '98: Proceedings of the 35th annual conference on Design automation.* ACM Press, 1998, pp. 678–683.

[80] X. Liu and M. C. Papaefthymiou, "A markov chain sequence generator for power macromodeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, July 2004.

[81] ——, "A static power estimation methodology for ip-based design," in *Design, Automation, and Test in Europe*, March 2001, pp. 280–287.

[82] Synopsys Inc., "Design compiler and primepower manual," `www.synopsys.com`.