A HYBRID HARDWARE/SOFTWARE ARCHITECTURE THAT COMBINES A 4-WIDE VERY-LONG INSTRUCTION WORD SOFTWARE PROCESSOR WITH APPLICATION-SPECIFIC SUPER-COMPLEX INSTRUCTION-SET HARDWARE FUNCTIONS

by

Dara Marie Kusic

BS, University of Pittsburgh, 2003

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

2005

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Dara Marie Kusic

It was defended on

July 6th, 2005

and approved by

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

Steven Levitan, John A. Jurenko Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Raymond R. Hoare, Assistant Professor, Department of Electrical and Computer Engineering

ABSTRACT

A HYBRID HARDWARE/SOFTWARE ARCHITECTURE THAT COMBINES A 4-WIDE VERY-LONG INSTRUCTION WORD SOFTWARE PROCESSOR WITH APPLICATION-SPECIFIC SUPER-COMPLEX INSTRUCTION-SET HARDWARE FUNCTIONS

Dara Marie Kusic, MS

University of Pittsburgh, 2005

Application-driven processor design is becoming increasingly feasible. Today, advances in field-programmable gate array (FPGA) technology are opening the doors to fast and highly-feasible hardware/software co-designed architectures. Over 100,000 FPGA logic array blocks and nearly 100 ASIC multiply-accumulate cores combine with extensible CPU cores to foster the design of configurable, application-driven hybrid processors.

This thesis proposes a hardware/software co-designed architecture targeted to an FPGA. The architecture is a very-long instruction-word (VLIW) processor coupled with super-complex instruction set (SuperCISC) hardware accelerants. Results of the VLIW/SuperCISC show performance speedups over a single-issue processor of 9x to 332x, and entire application speedups from 4x to 127x. Contributions of this research include a 4-way VLIW designed from the ground up, a SystemC VLIW simulator, a zero-overhead implementation of a hardware/software interface, evaluation of the scalability of a shared register file, examples of application-specific hardware accelerants, and an evaluation of shared memory configurations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

1.	ALU	Arithmetic Logic Unit
2.	CISC	Complex Instruction Set Computing
3.	CPU	Central Processing Unit
4.	DFG	Data Flow Graph
5.	DSP	Digital Signal Processing
6.	FPGA	Field Programmable Gate Array
7.	LAB	Logic Array Block
8.	LUT	Look-up Table
9.	PE	Processing Element
10.	RAM	Random Access Memory
11.	RISC	Reduced Instruction Set Computing
12.	VHDL	VHSIC Hardware Description Language
13.	VLIW	Very Long Instruction Word

ACKNOWLEDGEMENT

I would like to express my appreciation to my research and faculty advisor Dr. Raymond R. Hoare for his guidance and direction on this project. I would like to thank Dr. Hoare for the encouragement he continually offers to all members of the VLIW research team and for the positive work environment he fosters. It is truly a privilege to be a member of his research team. I would also like to thank Dr. Steven P. Levitan and Dr. Alex K. Jones for serving on my master's thesis committee. Drs. Hoare, Levitan and Jones have been an invaluable part of my master's studies, encouraging me to continue my education and take an active role in the academic community.

I would also like to thank Sandy Weisberg and Karen Dicks of the Electrical and Computer Engineering staff for their help navigating the university system. Last but certainly not least, I would like to thank my family for their love and support.

1.0 INTRODUCTION

Application-driven processor design is becoming increasingly feasible. In the past, application-specific processing architectures required two sacrifices: a costly development and fabrication process and a performance loss to interface via bus or to off-chip hardware. Today, advances in field-programmable gate array (FPGA) technology are opening the doors to fast and highly-feasible hardware/software co-designed architectures on a single chip with bus-less communication. Current FPGAs contain over 100,000 logic array blocks and nearly 100 application-specific integrated circuit (ASIC) multiply-accumulate cores. When efficiently combined, their processing capabilities exceed high clock-rate desktop processors.

This thesis proposes a hardware/software co-designed architecture targeted to an FPGA. The architecture is a very-long instruction-word (VLIW) processor coupled with super-complex instruction set (SuperCISC) hardware accelerants (or hardware functions). SuperCISC hardware is densely-packed, asynchronous combinational logic circuit that that executes tens to hundreds of equivalent software instructions representing a kernel of application source code within a single, long-latency datapath. A high level diagram of the VLIW/SuperCISC architecture is shown in Figure 1.

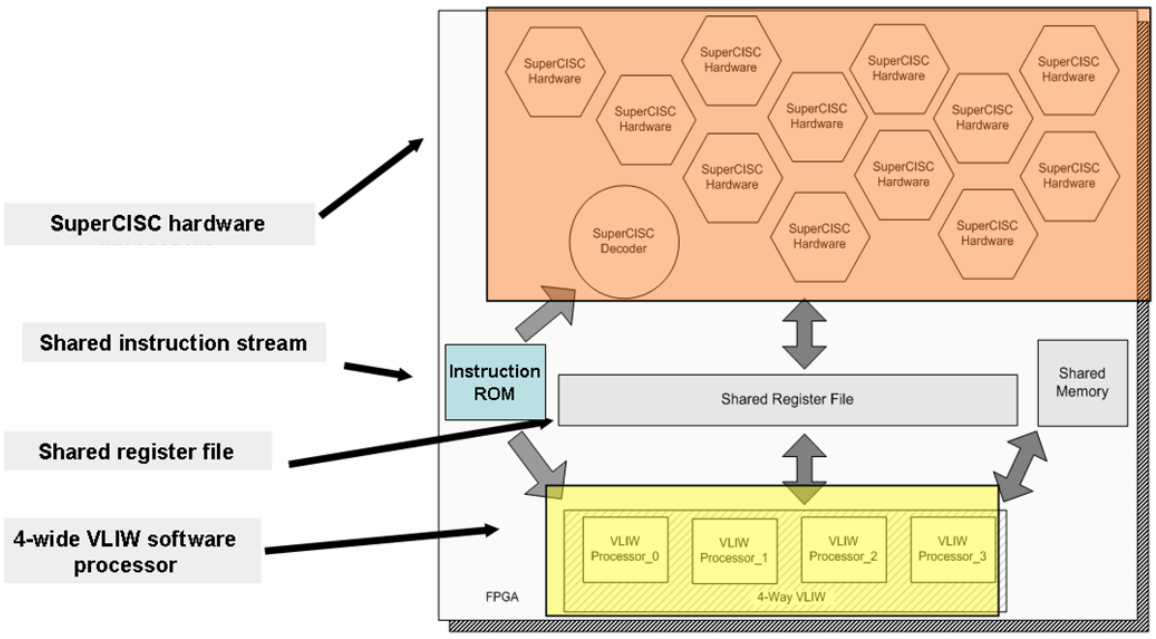


Figure 1. Block diagram of VLIW/SuperCISC architecture.

Results (Section 6.0) of the VLIW/SuperCISC architecture show performance speedups over a single processor of 9x to 332x, and entire application speedups from 4x to 127x. Contributions of this research include a 4-way VLIW designed from the ground up, the design of SystemC-based simulator to mimic the VLIW processor, a zero-overhead implementation of a hardware/software interface, evaluation of the scalability of a shared register file, examples of application-specific hardware accelerants, and an evaluation of shared memory configurations.

Contribution 1: A 4-way VLIW was designed from the ground up in VHDL hardware description language. The VLIW processor combines 4 parallel RISC processors to shared a register file, address space and instruction stream to execute up to 4 different instructions concurrently (scheduled at compile time).

Contribution 2: Designing SystemC-based simulator to mimic the VLIW processor. The SystemC VLIW processor provides run-time instruction counts by type and cycle-accurate snapshots of functional behavior of the system.

Contribution 3: A zero-overhead implementation of a hardware/software interface was designed for the VLIW/SuperCISC architecture. The interface binds the hybrid architecture through a common instruction stream and shared register file. The shared instruction stream enables zero-cycle context switching. The VLIW register file interfaces to the SuperCISC hardware with the addition of one 2:1 multiplexer per register. The extra latency is absorbed by the slack time of the register file respective to the VLIW. Therefore, there is zero noticeable cost to the operating frequency of the VLIW processor.

Contribution 4: This work evaluates the scalability of a shared register file. Measuring performance decline of the register file enables informed design decisions with respect to widening the VLIW processor.

Contribution 5: This work illustrates examples of application-specific hardware accelerants. SuperCISC hardware is an application-driven execution unit that represents a kernel of software code as a densely-packed, asynchronous combinational logic circuit. SuperCISC hardware is currently designed by hand using a 4-step process that starts with application source code profiling, to hardware/software code partitioning, to data-flow graph generation, and finally translation to VHDL hardware modeling language.

Contribution 6: Shared memory configurations are evaluated for performance cost. Two shared memory configurations were presented – an independently-address, interleaved memory architecture, and a vector-ported single-bank memory architecture.

Algorithms for real-time multimedia, signal processing and scientific computing applications continue to push for flexible, scalable, highly-parallel processors. Architectural options for achieving computational acceleration range from homogeneous, parallel processing networks, to high clock rate deep execution pipelines that support single-thread instruction-level parallelism (ILP), to application-specific hardware co-processors that can execute complex sequences of instructions in one time unit. No single option can deliver sought-after performance gains as effectively as drawing upon the advantages of each [1][2].

Homogeneous parallel processors offer some gain in performance, although speedups gained through parallel architecture are often limited by the ILP uncovered in benchmark applications [3][4]. Custom ASIC processors deliver high performance for specified applications, but are accompanied by high fabrication costs [5] and confronted by the challenges of reuse [6]. Reconfigurable devices such as FPGAs support application-specific computing with relatively low costs, but have often been dismissed because of slow performance and limited capacity [7][8]. Recent advances in FPGA technology, however, are beginning to reform that notion [9][10].

By combining a parallel processor with custom hardware paths on newer, faster FPGAs, it is possible to create an architecture in which each processing component compensates for the shortcomings imposed upon the other. The result is a hybrid processor that exceeds ideal performance gains of a homogeneous parallel processor functioning alone, and raises the effective throughput of an embedded processor implemented on an FPGA to compete with high clock rate desktop processors. Advances in design automation tools coupled with ambitious plans for a new era of configurable devices promise to make hybrid architectures a standard of computing. This thesis contributes findings on architectural tradeoffs affecting VLIWs and offer

performance results on hybrid architectures to further the advancement of hybrid processors and configurable devices for accelerated computing.

Four reduced instruction-set computing (RISC) computational units combine in parallel to form the VLIW unit within the architecture. A RISC machine provides low-level language support for a small set of simple instructions in a load/store based architecture [11]. A RISC-based architecture must be flexible enough to support a wide array of applications, but the application often requires large code size to compensate for flexibility.

In the VLIW, four RISC-type execution units share a register file and address space and concurrently execute up to four different instructions from a single application thread. The VLIW compiler ensures there are no data dependencies between the concurrent instructions [12]. The VLIW relies on the instruction-level parallelism discovered by the compiler to achieve parallel speedups.

Hardware accelerants use a complex-instruction set computing (CISC) model as the basis for the design. In the CISC model, one instruction indicates a set of low-level instructions. CISC machines often take tens to hundreds of cycles to execute a single instruction. In the architecture described in this paper, SuperCISC refers to a hardware-based accelerator for which one instruction indicates tens to hundreds of asynchronous operations that operate within a multi-cycle path interfaced to the VLIW. In the CISC model, computational density is provided at the cost of limited flexibility.

A hybrid RISC/CISC processor draws upon the strengths of each type of architecture. The proposed VLIW/SuperCISC architecture couples parallel RISC processors with specialized-execution CISC co-processor hardware. RISC processors are flexible - they make good general purpose processors because a small set of instructions builds a wide array of applications. CISC

processors are efficient – they execute complex operations in a single time instance. A hybrid architecture combines the flexibility of a RISC processor with the burst computational power of a CISC processor.

Section 2.0 introduces related work of similar research projects and presents some information on the Altera Stratix II FPGA technology platform. Section 3.0 describes the architecture of the VLIW/SuperCISC hybrid processor including the shared register file and hardware/software interface. Section 4.0 presents and analysis of the sources of speedup for SuperCISC hardware. Section 5.0 describes the VLIW and hybrid processor modeling methods. Section 6.0 presents performance results of the VLIW and VLIW/SuperCISC architectures as compared to a single-issue processor. Section 7.0 presents performance data on a shared memory architecture which may be implemented in the VLIW/SuperCISC at a future date, and Section 8.0 summarizes the main contributions of this work.

2.0 RELATED WORK

Hardware acceleration is the subject of many academic and industry studies and is supported by the availability of current field-programmable gate arrays (FPGAs). Research detailed in Section 2.1 generally shows significant speedups of hardware-augmented processors over baseline, software-execution processors. Although some researchers call for a new era of configurable devices to support hardware acceleration [7][8][19][20][21][22][23], the VLIW/SuperCISC architecture targets and is performance-profiled to an Altera Stratix II FPGA detailed in Section 2.2.

2.1 RELATED ARCHITECTURES

Related work to the VLIW/SuperCISC architecture includes processor/co-processor designs from industry and from academia. First, work from STMicroelectronics has produced a functional study of coupling reconfigurable hardware processing units with a VLIW. Next, work from University of California Berkeley has studied coupling configurable hardware with a single MIPS processor. Next, work on alternative kinds of configurable technology platforms is presented, followed by a description of parallel processing work from the University of Pittsburgh that precedes the work in this thesis.

Industry research at STMicroelectronics has produced a processor model consisting of a VLIW coupled with a reconfigurable functional unit (RFU) [15], closely aligned with the research presented in this thesis. The VLIW / RFU combines a 4-way ST200 [17] processor core with RFUs that serve as kernel accelerators. The project architecture is studied primarily for the relationship between RFU complexity, Amdahl speedup [12] and latency. Architectural and performance tradeoffs for RFUs of varying complexity are measured through execution times of pixel interpolation and standard average difference (SAD) routines for motion estimation in MPEG4 video encoding. In a worst case technology configuration factor for the RFU, results indicate a 5x speedup of an RFU-enhanced architecture over a baseline 4-way VLIW for benchmark applications, reducing the kernel execution time from 26% to 6% of the entire application. I/O bandwidth proves to be the limiting factor in increasing the width and complexity of the RFUs. While the VLIW/SuperCISC architecture presented in this thesis considers the underlying microarchitecture, the STMicroelectronics VLIW/RFU study performs no analysis of a supporting platform.

The Garp study conducted at University of California, Berkeley [7][8], produced a model for a reconfigurable hardware array coupled to one MIPS processor on a single chip. Garp devotes attention to the memory model supporting dynamic configuration of the coprocessor array, optimizing the model for latency reduction. Dynamic configuration of the SuperCISC co-processors is not considered in this thesis as capacity analysis conducted for this thesis has shown that current higher-end FPGAs can support a large number of co-processors to support application-specific computing without run-time re-configuration [18].

Garp researchers dismiss the aptness of FPGAs for the target of their computational model, instead proposing to target the architecture to currently unfabricated, rapidly configurable

hardware. The Garp project was undertaken in the late 1990's, basing many claims regarding the unsuitability of FPGAs on now-outdated devices. Advances in FPGA technology help to refute many of the claims of unsuitability. For one, FPGAs have greatly significantly increased the number of configurable logic blocks (CLBs), allowing enough area on the chip to contain many computational kernels in support of more than one application. Second, FPGAs now contain embedded high-speed multipliers, allaying Garp concerns about slow execution of complex operations. Third, although the on-chip RAM blocks are still limited, the amount of memory on an FPGA has increased since the first Garp publication.

PipeRench [19][20], as with Garp, argues for a new era of reconfigurable devices in attesting to the performance gains of application-specific computational hardware. PipeRench's main contribution is in proposing a reconfigurable architecture having a coarser logic grain than current FPGAs to support a host of expediters for application-development and run-time execution. PipeRench points out advantages of reconfigurable datapaths not explored in the context of this thesis, such as efficient handling of reduced or variable bit-width computation, and datapath pipelining to support intermediate-stage outputs. The work presented in this thesis focuses on processor architectures that exploit currently-available rather than proposed reconfigurable devices, but concurs that reconfigurable computing will benefit from advances in underlying microarchitecture. PipeRench also differs from the SuperCISC architecture by pipelining stages through the computational hardware, contributing to cycle time waste of simple operations.

HASTE [21] proposes a configurable fabric with interconnected sub-word ALUs. RaPid [22] is a pipelined, coarse-grain configurable architecture, and Matrix [23] similarly proposes a coarse-grain architecture for hardware-based acceleration. HASTE, RaPid and Matrix execute

time-consuming computational kernels on coarse-grained reconfigurable fabrics. Most FPGAs offer comparable coarse-grain support with embedded multipliers/adders. SuperCISC hardware co-processors implemented on an FPGA reduce the execution latency and increase throughput of computational kernels as compared with reconfigurable fabrics.

The Imagine processor architecture [24][25] combines a reconfigurable co-processing functional unit with a processor. The architecture pairs a very wide SIMD/VLIW processor engine with a host processor. The functional unit/host processor relies on ILP for speedup, but it is often difficult to achieve these gains due to low ILP discovered within the application source code. The VLIW/SuperCISC architecture differs from the Imagine processor because it uses a combinational hardware for the main thrust of its application acceleration.

The Chimaera processor [26] combines a reconfigurable functional unit with a register file with a limited number of read and write ports. Our system differs as we use a VLIW processor instead of a single processor and the SuperCISC hardware coprocessors connect directly to all registers in the register file for both reading and writing, allowing hardware execution with no overhead affecting the overall performance of the architecture. Chimaera assumes that the hardware resource must be reconfigured at run-time to execute an application kernel, which may require significant overhead. In contrast, SuperCISC hardware units are configured prior to run-time. Additionally, the VLIW/SuperCISC system is implemented in a single FPGA and thus has a faster design-to-deployment time than Chimaera and any of the reconfigurable fabrics mentioned above.

Recent work in SIMD architecture connects 64 and 88 processing elements using a hypercube network [27]. This architecture was studied for the effects of scaling and shows a modest decline in performance as the number of processors scaled from 2 to 88. Instruction

broadcasting and communication interconnect were the only factors significantly impeding scalability of the architecture. Like the VLIW/SuperCISC, the SIMD ALUs utilize embedded multiply-add cores on the FPGA and make use of a system of custom instructions to extend the ISA of the 'host' processor (VLIW, SIMD, respectively). The SIMD architecture requires a communication strategy not required by the VLIW/SuperCISC. One limitation of a SIMD architecture is the requirement for sameness of the instructions executed in parallel. However, many benchmark applications cannot fully utilize the degree of parallelism in a SIMD architecture. Additionally, the SIMD architecture requires parallel programming while programming for the VLIW/SuperCISC is compiler-driven from serial source code.

Research in industry [28] shows that coupling a VLIW with a reconfigurable resource offers the robustness of a parallel, general-purpose processor (VLIW) with the accelerating power and flexibility of a configurable hardware co-processor. The cited work assumes zero reconfiguration penalty of the co-processing 'grid' and that design automation tools tackle the problem of reconfiguration. The VLIW/SuperCISC differs because the FPGA resources are programmed prior to execution, giving us a more realistic reconfiguration penalty of zero. The VLIW/SuperCISC project is actively developing a compiler and automation flow to map application kernels as SuperCISC hardware to the reconfigurable device.

2.2 TECHNOLOGY PLATFORM

The proposed VLIW/SuperCISC architecture described in Section 3.0 was performance profiled with respect to an Altera Stratix II EP2S180 FPGA. The Stratix II family was selected due to its large number and density of configurable logic-array blocks, embedded DSP multipliers and

improved memory capacity over the Stratix I. Comparable target technology to the Stratix II includes the Xilinx Virtex IV family of devices.

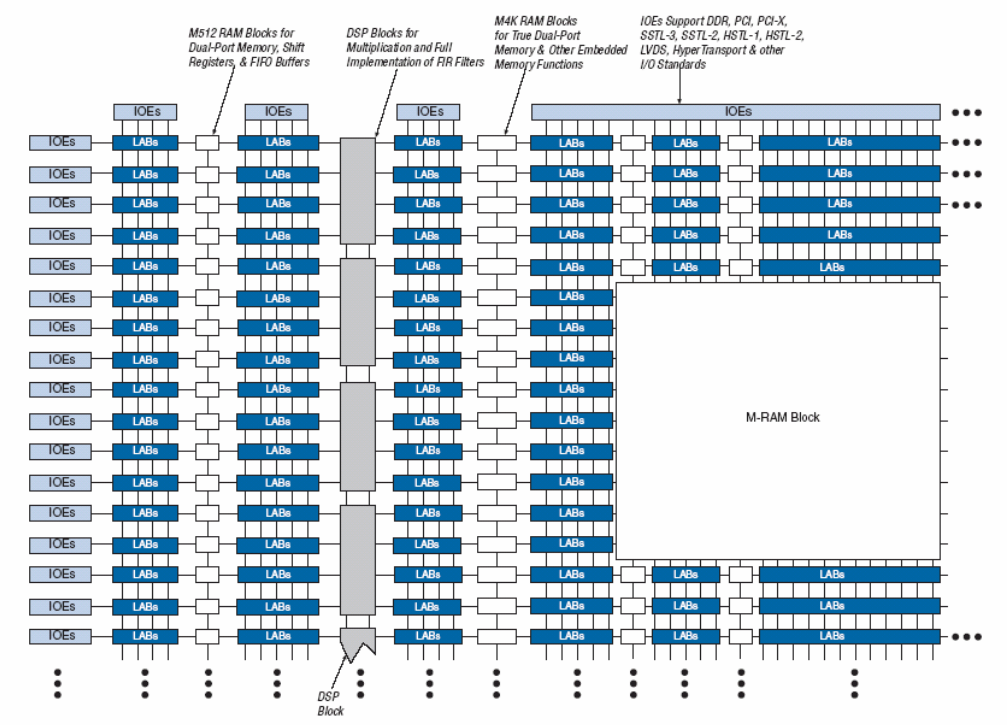


Figure 2. Block-level architecture of Stratix II FPGA [29]

The Stratix II regular row- and column-based architecture shown in Figure 2 distributes configurable resources over the chip, making feasible a massively parallel design in which each component has equal access to resources. Configurable logic cells are called logic array blocks (LABs) or arithmetic lookup tables (ALUTs) each containing eight adaptive logic modules (ALMs), carry chains, arithmetic chains, control signals, and interconnect lines. Figure 3 shows an ALM on a Stratix II with unregistered input, combinational logic, arithmetic units, carry chains and registered output. The arithmetic units support addition, subtraction and logic

operations AND, OR, NOT and XOR. Shift operations are implemented through bit manipulation between the ALMs. Data storage elements are D-type flip-flops.

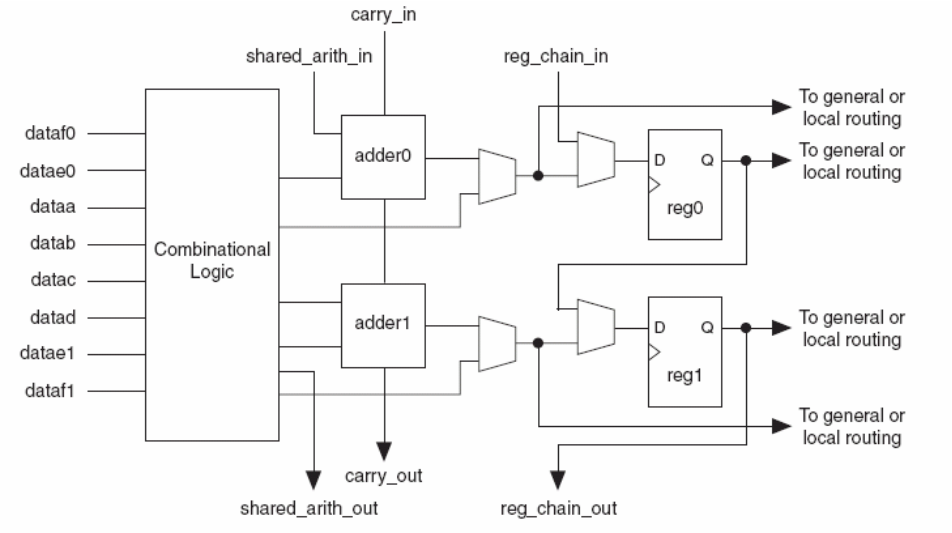


Figure 3. Block-level diagram of an adaptive logic module (ALM) on a Stratix II [29]

There are 768 9x9-bit multipliers on an Altera EP2S180 that can work together to form 96, 36x36-bit multipliers distributed over 4 columns that are available for parallel processing. Data collected for this thesis shown that a 32x32-bit multiplication has a post place-and-route performance of 322 MHz on an EP2S180. Figure 4 shows the DSP block configuration for an 18x18-bit multiply-and-accumulate (MAC). The depth of this diagram would be doubled to support a 36x36-bit MAC. Note the nodes prior to the adder that indicate support of product saturation.

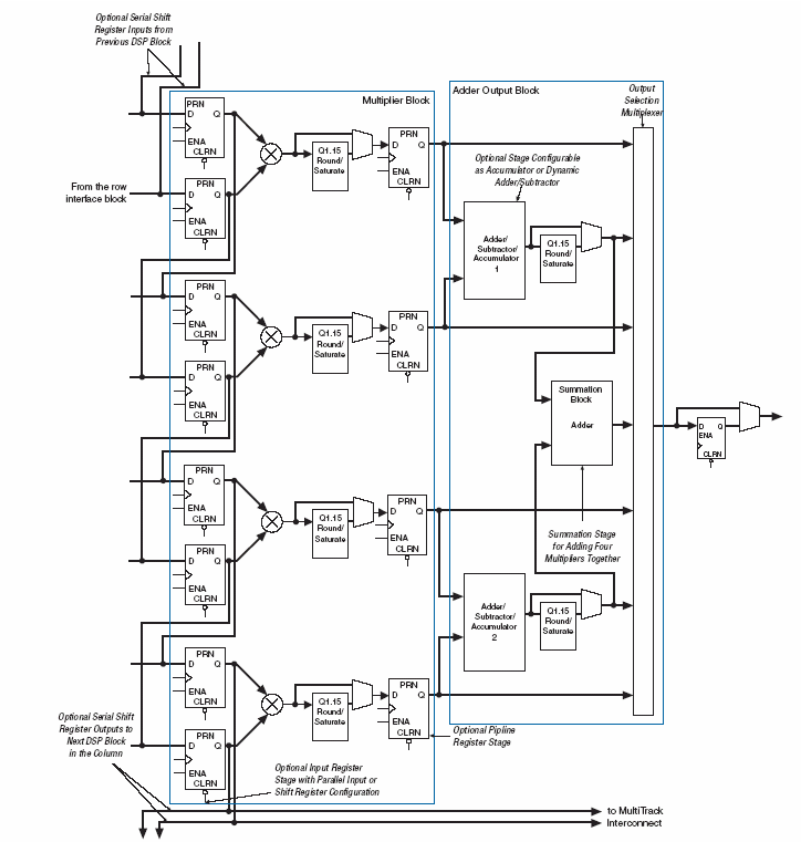


Figure 4. Block-level diagram of a digital signal processor (DSP) showing configuration for 18x18-bit multiply-and-accumulate (MAC) on a Stratix II [29]

Stratix II devices are fabricated in 90nm technology that Altera white papers report can support clock rates up to 500MHz, although Altera Quartus II post-place and route timing results never exceed 422MHz for any design implemented on the device. Parallel processing designs can increase the effective throughput of a single-issue CPU by a factor equal to the width of a multi-issue parallel architecture.

Three types of random-access memory (RAM) blocks are available on a Stratix II: M512 RAM, M4K RAM and M-RAM, performing at ideal maximum clock speeds of 380MHz, 400MHz, and 400MHz, respectively. Each block can support varying storage configurations and port widths. The M512 blocks contain 512 bits plus parity, the M4K contain 4096 bits plus

parity, and the M-RAM blocks have 512K bits plus parity. M512 blocks are small and useful for first-in-first-out (FIFO) modules. Program caches and lookup schemes are suited to the M4K blocks, and large volume data storage are suited to the M-RAM blocks. All but the M512 blocks support true dual-ported memory access. Figure 5 shows the placement of the different types of memory blocks within the row and column architecture of the device, relative positioning to DSP blocks, and orientation with respect to I/O pins.

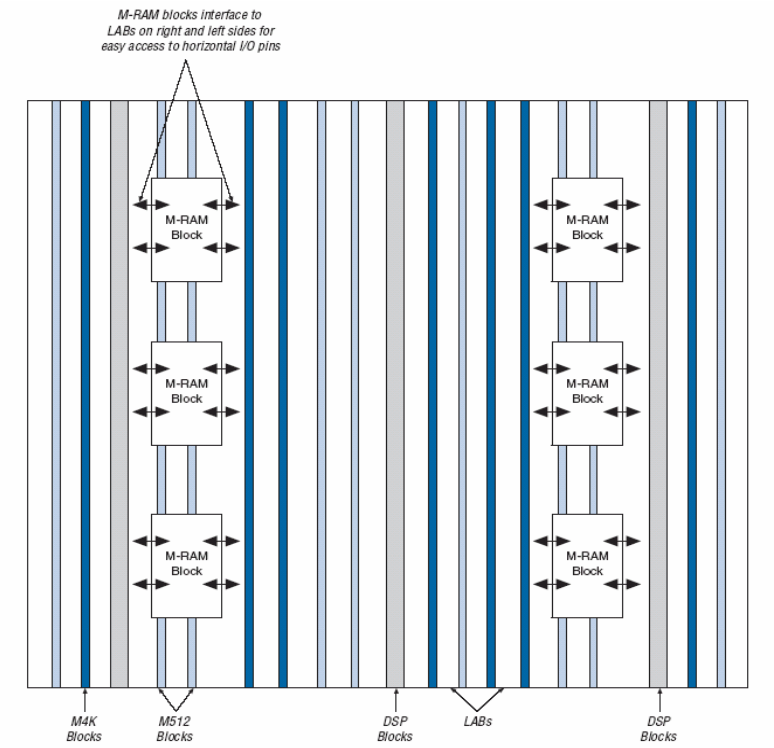


Figure 5. Block-level diagram of the memory distribution on a Stratix II [29]

Mapping an architectural design onto an FPGA involves configuring row and column interconnection signals between LABs, RAM and DSP blocks. Configuration of the logic elements by Quartus II place-and-route software uses the simulated annealing algorithm with an infinite number of solutions. The Quartus II software can be set for varying levels of

optimization for the place-and-route algorithm. The software is mature enough to be reasonably trusted to deliver place-and-route results within an optimal range, but the Quartus II software provides an interface for manual placement modifications if so desired.

The EP2S180 is the largest and most robust of the FPGAs in the Stratix II family. Generally, the abundance of LABs and DSP blocks can support very-wide parallel processor designs. The limiting factor for a processor implementation is often on-chip memory capacity. A design seeking additional memory can use the I/O pins to interface to external memory devices such as DDR, SDRAM and SRAM. Serial interface channels are available for other types of data movement. Table 1 quantizes the available resources on a Stratix II EP2S180.

Table 1. Available resources on an Altera Stratix II EP2S180.

Resource	Count
LUT Columns	100
LUT Rows	96
Total LUTs	144,520
9x9-bit DSP Multipliers / Blocks	768 / 72
M512 RAM Blocks (512 b)	930
M4K RAM Blocks (4 Kb)	768
M-RAM Blocks (512 Kb)	9

3.0 ARCHITECTURAL DESCRIPTION

The RISC/CISC hybrid software/hardware architecture is composed of a 32-bit, 4-way VLIW supporting NiosII RISC instruction-set architecture (ISA) coupled with super-complex instruction set (SuperCISC) co-processing hardware. The hybrid architecture features 4, flexible RISC processors in VLIW configuration for general-purpose processing and SuperCISC hardware accelerants for specific-purpose processing. The VLIW was designed from the ground-up in VHDL to match the NiosII ISA within each processing element. SuperCISC hardware was designed in VHDL to match data-flow-graphs representing application kernels. The performance of single-issue and VLIW CPUs are used as benchmarks against which to compare the results of an architecture supported by SuperCISC hardware.

3.1 HARDWARE/SOFTWARE PARTIONING

A hybrid software/hardware processor requires partitioning the application source code into portions that run on the RISC general-purpose processor and portions that run on the CISC application-specific hardware. Partitioning the source code into software/hardware blocks allows the architecture to pursue two types of application speedup: instruction-level parallelism and hardware acceleration.

The VLIW utilizes instruction-level parallelism (ILP) to achieve application speedup. ILP is the overlap between non-dependent instructions within application code or code block. Average ILP is an indication of the number of instructions that can be executed concurrently in a parallel processor. A VLIW relies on the compiler to uncover ILP and schedule instructions, as opposed to a superscalar processor that schedules concurrent instructions at execution time using special hardware support.

If the VLIW executes 100% of the application, the speedup is limited by the width, or the number of processing units, in the VLIW. A 4-way VLIW executing 100% of the code, with 100% of the processing units active at all times can achieve a maximum of 4x speedup over a single-issue processor operating at the same clock-speed. Amdahl's Law of the limitations imposed on speedup can be expressed as follows [12]:

Equation 1. Amdahl's equation for execution time improvement.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

To illustrate the above equation, suppose an application has a 100 second execution time. If 100% of the 4 VLIW processing units were actively executing 100% of the code, the speedup is 4x. This can be represented by the following example:

Equation 2. Example of Amdahl's execution time improvement of 4-way VLIW with 100% processing elements accelerating 100% of the execution time.

$$\frac{100 \text{ seconds}}{4x} + 0 \text{ seconds} = 25 \text{ seconds, or 4x speedup}$$

Because ILP limits the amount of code affected by the 4x processing power of the VLIW, a second way to achieve gains is through hardware acceleration. A 'rule-of-thumb' that can be called the *90/10 rule* states that 10% of an application's source code is responsible for 90% of the execution time. A CISC-type hardware accelerant targets the 10% code that often dominates execution time and seeks to minimize its execution time through compacted, asynchronous processing elements. Supposing the hardware accelerant effects a 12x speedup on 90% of the execution time, and 10% execution time takes place on the VLIW, the example in Equation 2 would re-calculate as follows:

Equation 3. Example of Amdahl's execution time improvement for 4-way VLIW with 100% processing elements accelerating 10% of single-issue execution time and SuperCISC hardware with 12x speedup accelerating 90% of single-issue execution time.

$$\frac{10 \text{ seconds on VLIW}}{4x} + \frac{90 \text{ seconds on SuperCISC}}{12x} = 10 \text{ seconds, or 10x speedup}$$

The VLIW/SuperCISC pursues two types of performance gains: those achievable through ILP and those achievable through the *90/10 rule*. The 4-way VLIW can execute up to 4 instructions in parallel, indicating an ideal 4x speedup. To achieve a target application speedup of 10x, a

SuperCISC hardware accelerator must have a minimum speedup 12x active upon 90% of the single-issue execution time. This SuperCISC portion of the code is usually a frequently-called function or loop-bound computation. SuperCISC hardware functions execute this fraction of the code within densely-packed, asynchronous computational elements in an effort to minimize execution time. The remaining code is parallelized using its ILP and executed upon the VLIW.

The mass of asynchronous processing elements in a multi-cycle path, or SuperCISC hardware, contributes performance gains through two notable attributes: turning software control-flow into hardware data-flow, and compressing multiple synchronous-ALU operations into a single asynchronous computation. These two performance attributes of SuperCISC accelerators exhibit their gains in varied proportions depending on whether a candidate code block is control-flow intensive, or computation intensive.

Turning software control-flow into hardware data-flow pursues *control flow efficiency*. SuperCISC hardware implements branch control using a multiplexer. The multiplexer inputs articulate all possible branch outcome sequences. The result is a hardware-based control structure that pre-calculates all possible branch outcomes in anticipation of a switch (multiplexer). The multiplexer model adds significant performance gains to portions of code rich in control flow.

Mapping a sequence of synchronous-ALU operations through compact, asynchronous processing elements pursues the gains of *cycle compression*. The asynchronous paths of a SuperCISC function eliminate the cycle-time waste suffered by synchronous pipeline CPU architectures. In a synchronous processor, all operations execute at the global clock rate set by the slowest path of the architecture. But simple operations, such as logical bit-shifts, execute quickly and idle for the remaining clock period. SuperCISC hardware eliminates wasted cycle

time. SuperCISC hardware executes a stream of operations through a series of asynchronous processing elements. Although the latency of the SuperCISC hardware is greater than the VLIW cycle time, SuperCISC hardware densely packs computational nodes on top of one another to execute each node operation at the maximum speed supported by the platform technology. The efficiency of asynchronous processing elements can be referred to as *cycle compression*, and is defined and discussed in Section 4.2. Cycle compression adds significant performance gains to applications rich in arithmetic operations.

SuperCISC hardware contributes significant performance gains at little expense relative to the overall area occupied by the VLIW. The SuperCISC hardware designed in VHDL to execute benchmark kernels profiled for this work each utilize less than 1% of the Altera Stratix II FPGA logic area and adds an average increase of 0.2x to the area of the VLIW for an average 10x boost in performance. SuperCISC hardware shows an average ratio of 50:1 performance to area gain when interfaced with the VLIW.

The SuperCISC hardware requires no overhead to interface to the VLIW by using a shared instruction stream supporting specialized instructions. The ISA chosen for the VLIW processor designed in VHDL from the ground up supports the NiosII extensible instruction set [13]. The extensible ISA reserves a unique operation code to identify a *custom instruction*. The *custom instruction* calls to the SuperCISC hardware rather than the VLIW. The NiosII processor model requires no modifications to make use of this *custom instruction*, thus there is no overhead for the interface of SuperCISC hardware functions to the VLIW software processor. A high-level block diagram showing the VLIW, SuperCISC hardware and custom instruction calls is shown in Figure 6.

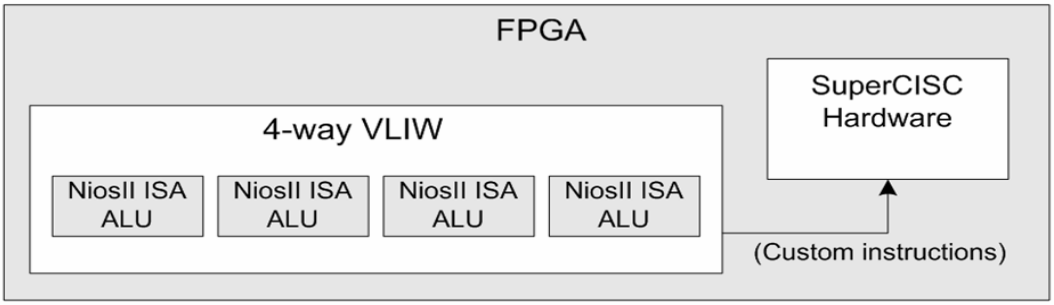


Figure 6. Simplified block-diagram showing custom instruction calls that extended from NiosII ISA 4-way VLIW.

The SuperCISC hardware requires negligible overhead to interface to the VLIW processor through the shared data components. Architectures with hardware accelerants can suffer from latencies because of bus-based communication [14]. The VLIW/SuperCISC architecture has bus-less communications, and instead uses a shared instruction stream and register file to interface the VLIW and SuperCISC. This design feature omits the overhead and complexity of coherence strategies incurred by architectures with localized local data stores and bus-based data sharing.

The VLIW processing units interface with the SuperCISC hardware via a shared 32-element register file and common instruction stream. Access ports to the register file are controlled by data multiplexers and address decoders. The VLIW has access to 8 read and 4 write ports, while the SuperCISC hardware may read from and write to the entire register file through direct access lines (no address decoding).

The VLIW and SuperCISC hardware share a common instruction stream. The instruction vector is 128-bits wide, segmented into 4, 32-bit instructions. Each 32-bit instruction word is dedicated to one of the VLIW processing units. SuperCISC instructions are also issued from the VLIW instruction stream. A controller external to the VLIW intercepts custom instructions and

signals one or more SuperCISC hardware units to commence execution. In the current implementation, VLIW activity is suspended during SuperCISC execution.

The VLIW processing units share an instruction cache, but each VLIW processing unit contains its own local decoder. The presence of local decoders indicates support for non-homogenous instruction execution. Control instructions that alter the program counter are directed to only one processing element in the VLIW. This strategy avoids potential conflicts and limits routing control interconnect overhead to the program counter.

Instructions to the SuperCISC hardware are issued from the VLIW instruction ROM. The SuperCISC hardware processing units receive global control signals (activate, writeback multiplexing) from a global SuperCISC controller. The SuperCISC controller is a ROM that contains the control signals for each SuperCISC hardware unit signified by an address within the ROM.

The result of combining the above elements is a heterogeneous, hybrid architecture united by a shared instruction cache and global data stores. Program flow switches from the VLIW to the SuperCISC co-processing hardware wherever *pragma* statements denote a hardware function in the source code. The design flow for SuperCISC hardware, including a description of how application kernels are selected and where the hardware/software context switching occurs is described in greater detail in Section 3.5. A block diagram of the proposed processor architecture is shown in Figure 7. Resources shared among processing elements in the current implementation are the instruction ROM and register file.

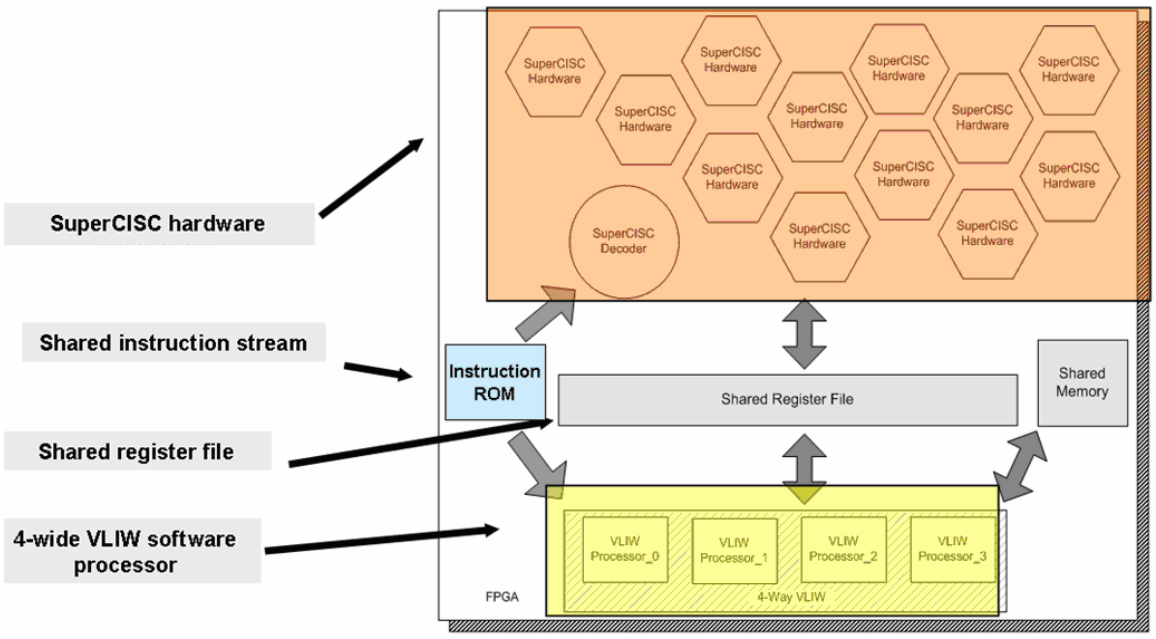


Figure 7. Block diagram of VLIW/SuperCISC architecture.

3.2 VLIW ARCHITECTURE

The VLIW consists of 4, 32-bit integer processors sharing a memory bank, register file and instruction cache. Each VLIW processing unit contains an instruction decoder and NiosII ISA RISC processor. The shared register file has 32 elements with 8 read ports and 4 write ports. The memory is a 12-bit word-addressable true dual-ported memory storing up to 16KB of data. Instructions are 128-bits wide and are stored within a read-only memory (ROM). The instruction 128-bit vector is divided into 4, 32-bit instructions to be directed to the appropriate VLIW processing element (PE), PE0 to PE3.

VLIW processing elements operate on parallelized code of the NiosII instruction developed by the University of Pittsburgh team working on the compiler and design automation for the architecture. The NiosII has an extensible ISA that supports custom instructions

instantiated to specify calls to SuperCISC hardware functions, discussed in Section 3.5 of this thesis. Custom instructions are executed outside of the VLIW core.

Static timing analysis performed by Quartus II synthesis and post-routing processes for the 4-way VLIW on an Altera EP2S180 estimates an operating clock speed of 167MHz. The effective VLIW throughput is 668 millions of operations per second (668 MOPS) for an application with ideal ILP of 4, equal to the number of RISC processors available in the architecture.

Processing of instructions occurs within a 6-stage pipeline consisting of instruction FETCH0, DECODE, operand FETCH1, EXE0, EXE1 and WB. Multiplication completes in two EXE cycles and all other arithmetic operations execute in one cycle. Branch instructions execute only on PE0 to eliminate potential conflicts in the control flow of the program. LOAD and STOR operations are limited to PE0 and PE1 due to dual ports on the memory, therefore limiting ILP of memory operations to 2.

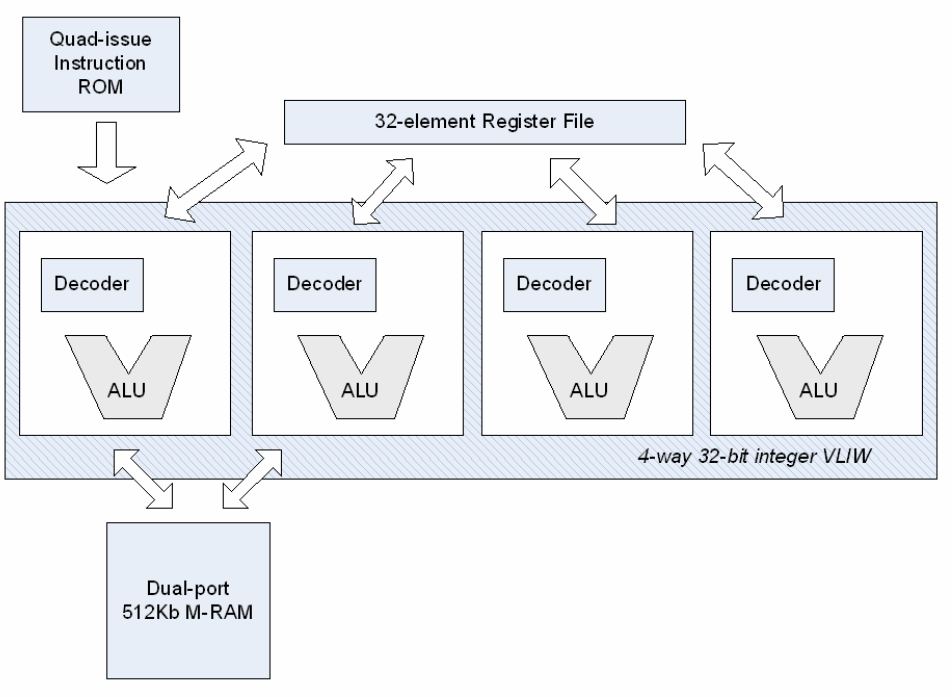


Figure 8. Block diagram of 4-way VLIW.

An adder external to the core arithmetic logic unit (ALU) of the processing elements calculates memory addresses because its addition has shown to improve overall performance of the VLIW. Reducing the size of the memory and invoking multiple shifters in the ALU dedicated to different types of shift operations further optimize the performance of the VLIW. A block diagram showing the ALU, data store and instruction-issue elements of the VLIW is shown in Figure 8.

3.3 SHARED REGISTER FILE

A shared 32-element register file having 8 multiplexed read ports and 4 multiplexed write ports unifies the hybrid architecture. An arbitrarily-sized register file can be represented in Figure 9,

where *P* processing elements interface to *N* registers. The gains offered by a wide VLIW are offset by the performance degradation of scaling access ports to the register file. The number of multiplexed data ports has shown to be the major impedance to scaling a register file in a multiprocessor architecture. However, SuperCISC hardware access to the register file requires a slightly simpler configuration than the VLIW. This is due to the SuperCISC's direct access lines that do not require an address decoder. An illustration of the augmented register file is shown in Figure 12.

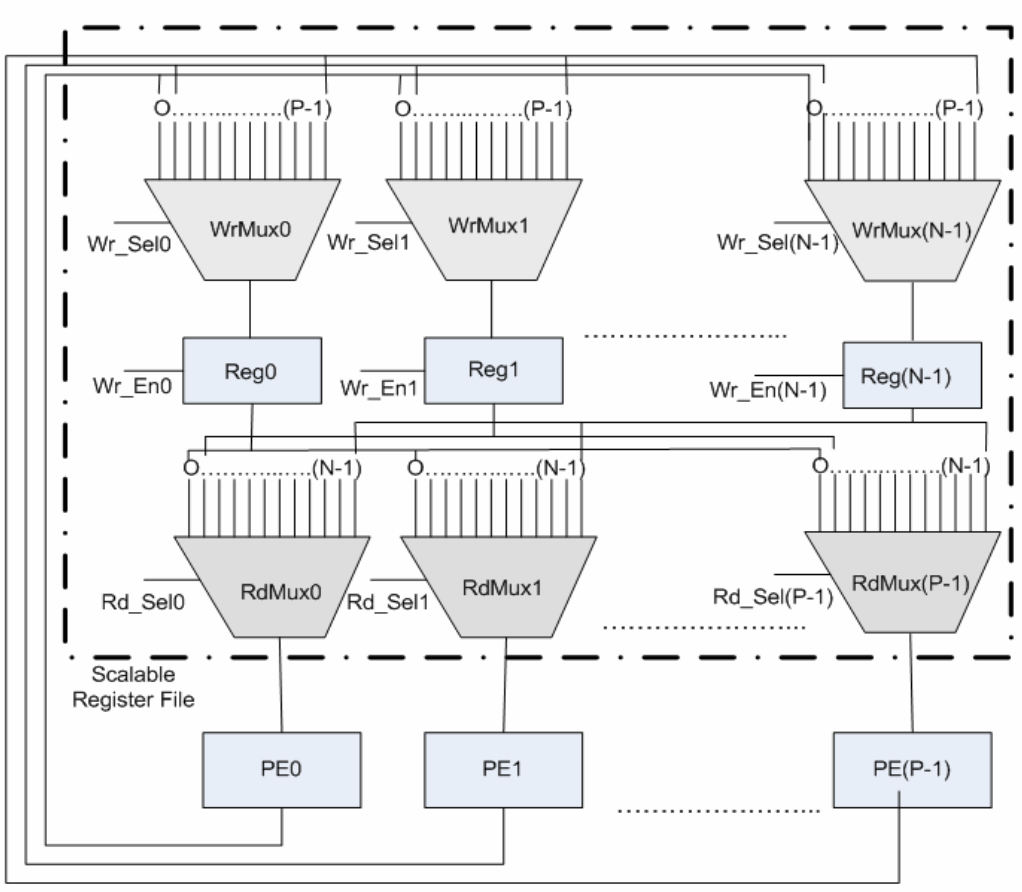


Figure 9. *N*-element register file supporting *P*-wide VLIW with *P* read ports and *P* write ports.

Figure 9 shows a *P*-processor VLIW with an *N*-element register file. In front of each port sits a multiplexer that routes data to and from the register file. Multiplexers on write ports have a width, *P*, equal to the number of processors and exist in quantities equal to the number of registers, *N*. Each of the *N* registers requires a 32-bit *P*-to-1 multiplexer. Multiplexers on read ports have a width equal to the number of registers, *N*, and exist in quantities equal to 2x the number of processors, *P* (to support 2 operand fetches). Each of the 2*P* read ports requires a 32-bit *N*-to-1 multiplexer.

Multiplexers are fundamental to scaling a register file. The effect of multiplexers is charted in Figure 10 by plotting the degrading performance of a shared register file against an increasing number of ports. When the number of register is fixed, the ports become an expression of the width of the VLIW. The additional routing and wider multiplexers to accommodate a wider VLIW architecture adds complexity to the register file and imposes bottlenecks upon the overall performance.

In Figure 10, the number of 32-bit registers is held constant and the number of processors is scaled. There are 3*P* ports on the register file, that is, for *P* processors there are 2*P* read ports and *P* write ports. Performance of the register file can be represented by the equation (273 MHz – (16**P*/2)), where *P* is in the set of values represented by integer powers of 2. The register gains an average of 2x area utilization per doubling of *P*.

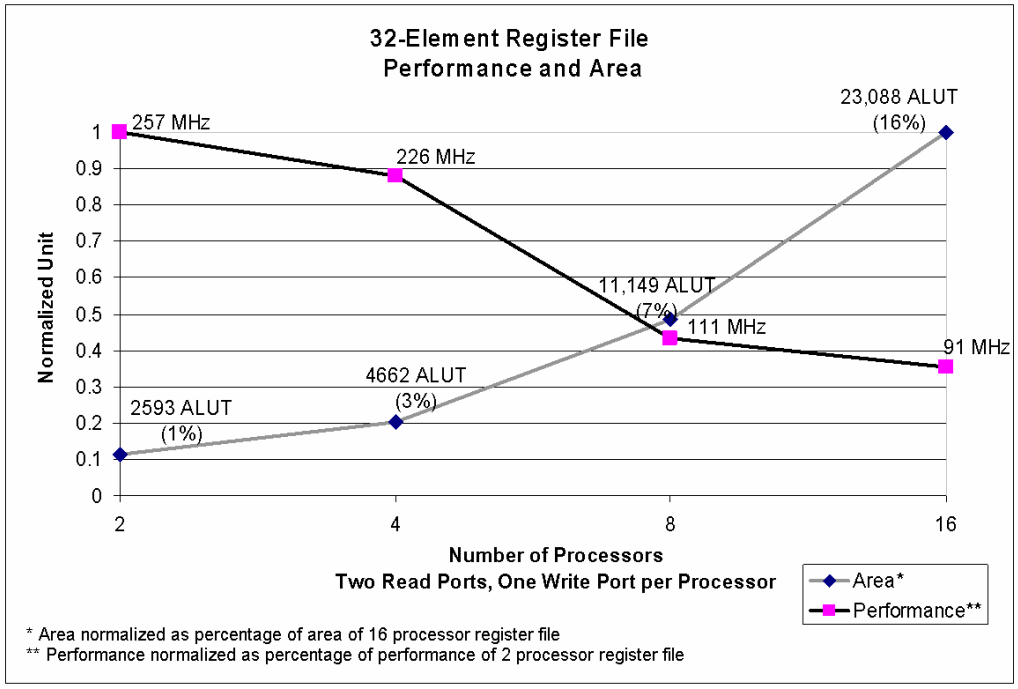


Figure 10. Scalability of a 32-element register file for *P* processors having 2*P* read and *P* write ports on an Altera Stratix II EP2S180.

As the most elemental design unit contributing to performance of the register file, a multiplexer can be analyzed in isolation. Figure 11 shows the impact of increasing width to a 32-bit *P*-to-1 multiplexer on the Stratix II EP2S180. As the width of the multiplexer doubles, the area consumed on an EP2S180 for the multiplexer increases by a factor of 1.4x. The performance loses an average of 44 MHz with each doubling of the width of the multiplexer. To expand the width of the VLIW, both the width and quantity of multiplexers must increase. Therefore, the register file incurs two strikes upon its performance to increase the width of a VLIW.

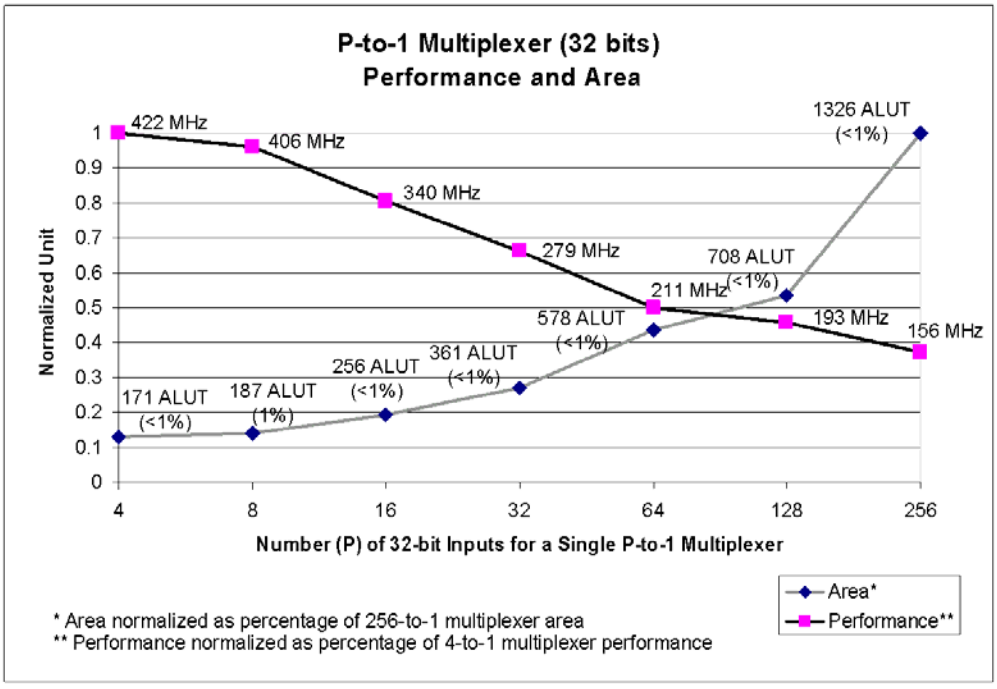


Figure 11. Scalability of a 32-bit *P*-to-1 multiplexer on an Altera Stratix II EP2S180.

In this context, the parallel processing power of a wider VLIW is offset by the overhead costs of adding more complex access ports to the register file. This effect applies somewhat differently to VLIW than to the SuperCISC hardware functions, which interface to the register file in a more simplified manner than the VLIW, shown in Section 3.4. The cost of interfacing the SuperCISC to the register file is consumed by the slack time of the resister file within a 167MHz VLIW architecture on the Stratix II FPGA. In further support of adding processing power by interfacing SuperCISC hardware rather than more VLIW processing elements, the speedups achieved by adding SuperCISC hardware functions to the VLIW far exceed the speedups of a wider VLIW, as illustrated in Section 6.2.1.

Considering the costs of scaling the register file for the VLIW, it is important to know the levels of ILP that can be discovered by the compiler within a set of target applications before choosing a parallel architecture. For example, choosing an 8-way VLIW when the average ILP of target applications is 1.6x would lead to underutilized resources and a complexity that adds more cost to the performance than realized benefit. The width of the VLIW then becomes an impedance rather than a facilitator to fast execution of low-ILP applications.

3.4 ZERO-OVERHEAD HARDWARE/SOFTWARE INTERFACE

The VLIW interfaces to the SuperCISC hardware in two ways: sharing a common instruction stream and sharing a common register file. The two forms of hardware/software interface contribute no noticeable cost in performance to the VLIW – the instruction stream has a zero-cycle context switch between VLIW and SuperCISC, and the added latency of interfacing SuperCISC hardware to the register file is absorbed by the slack time of the 226MHz register file with respect to the 167MHz VLIW.

Adding direct access of the SuperCISC hardware detracts little from the performance of the register file. SuperCISC hardware has a different access strategy to the register file than the VLIW processing units. A SuperCISC hardware function reads from the register file in direct lines linking each register to all SuperCISC hardware functions (no address decoding). The SuperCISC hardware function may write to all 32 registers using a direct line access. The register file needs only a 2:1 writeback multiplexer in front of each register to context switch from VLIW access to SuperCISC access. A diagram of the register file showing SuperCISC hardware access is shown in Figure 12.

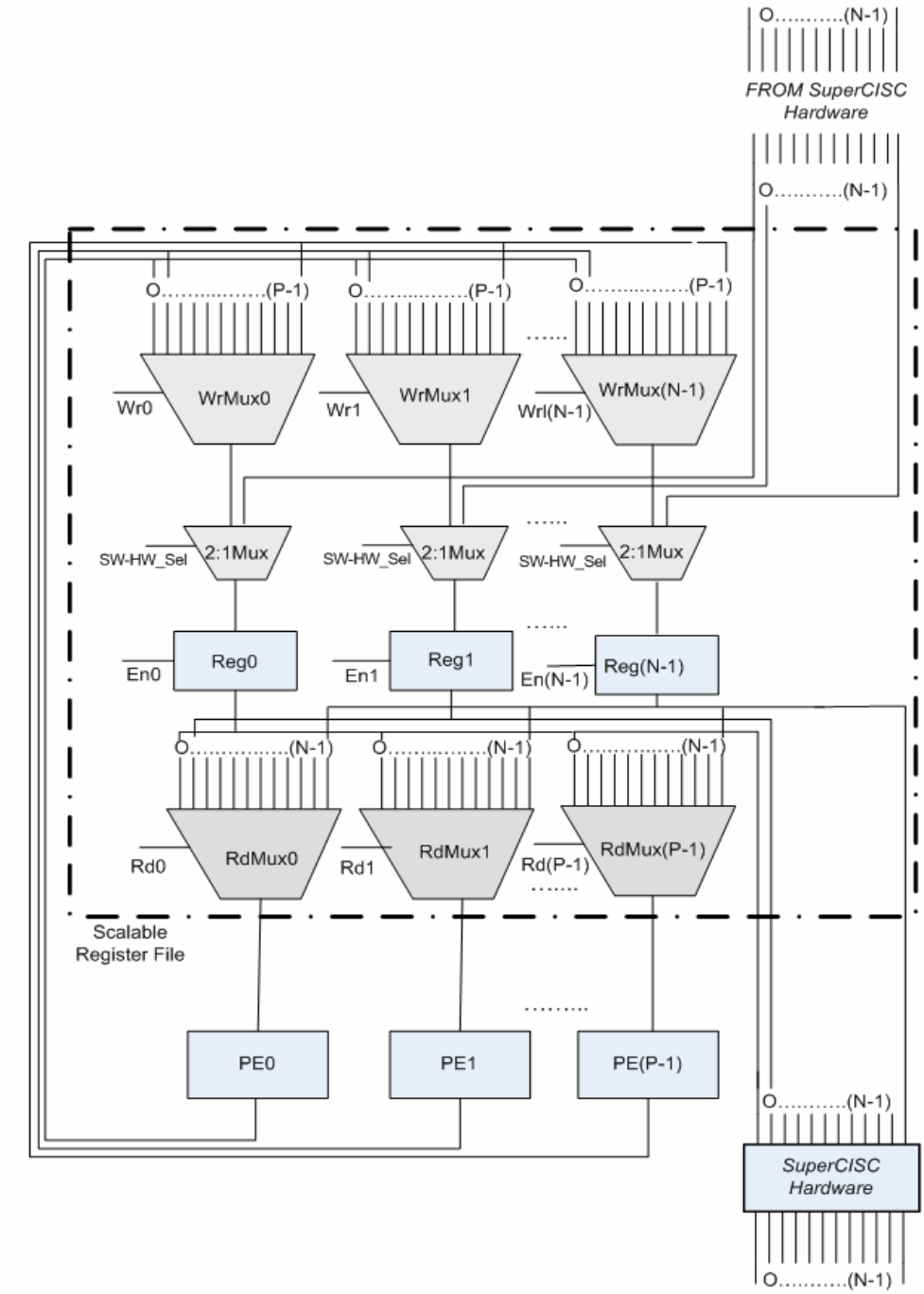


Figure 12. *N*-element register file supporting SuperCISC hardware and a *P*-wide VLIW with *P* read ports and *P* write ports.

The cost of adding SuperCISC access to the register file can be determined by plotting the performance and area of a register file without SuperCISC access with the performance and area of a register file with SuperCISC access. The difference between the values on the vertical axis represents the cost of interfacing the entire register file to SuperCISC hardware. The plot of the register file without SuperCISC access and the register file with SuperCISC access is shown in Figure 13. The average cost of the software/hardware interface to the register file is 20MHz; the average increase in area for adding SuperCISC access is 8%. Added to a 4-wide VLIW, SuperCISC hardware detracts only 29MHz from the performance of the register file, meeting the 167MHz overall system timing constraint. The added latency is absorbed by the slack time of the register file and adds nothing to the cost of the hardware/software interface.

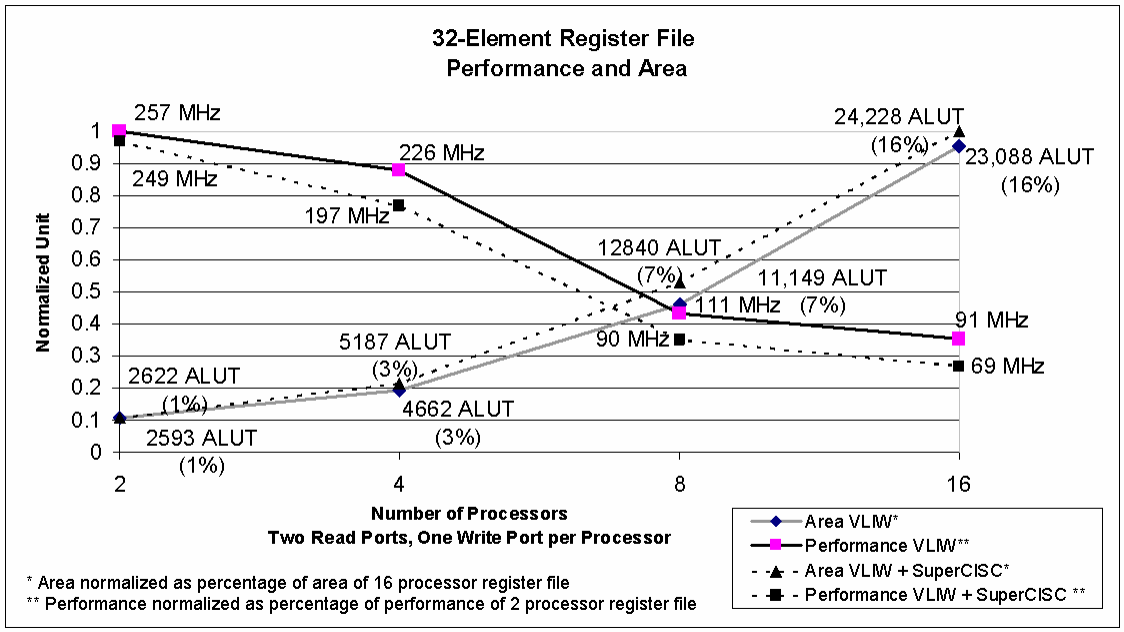


Figure 13. Scalability of a 32-element register file for *P* processors having 2*P* read and *P* write ports and full SuperCISC hardware access on an Altera Stratix II EP2S180.

3.5 SUPERCISC HARDWARE FUNCTIONS

SuperCISC hardware is an application-driven processing circuit that represents a kernel of software code as a densely-packed, asynchronous combinational logic circuit. SuperCISC hardware co-processors pursue application speedups of the 90/10 rule by executing the most run-time intensive portions of code in circuits maximized for throughput and minimized for latency. Pursuing gains of the 90/10 rule in a SuperCISC hardware function increases the overall application speedup when interfaced to the 4-way VLIW, which can only achieve a maximum speedup of 4x according to Amdahl's equation.

The design flow of SuperCISC hardware functions is a 4-step process (Figure 16) currently being automated by the University of Pittsburgh team developing a compiler for the VLIW/SuperCISC architecture. The first step is to profile the application source code and identify the run-time intensive portions of the code. The goal is to isolate the code blocks contributing to 90% or more of the execution time of the program. The next step is to partition the code into software and hardware blocks for the VLIW and SuperCISC hardware functions, respectively. The third step generates data-flow graphs (DFGs) for the hardware portions of the code. The last step is to convert the DFGs into VHDL hardware description language to interface to the VLIW architecture. It is the last two steps of DFG generation and VHDL conversion currently undergoing design automation at the University of Pittsburgh. Successful completion of this work will significantly advance application-driven processors and behavioral synthesis [32][33].

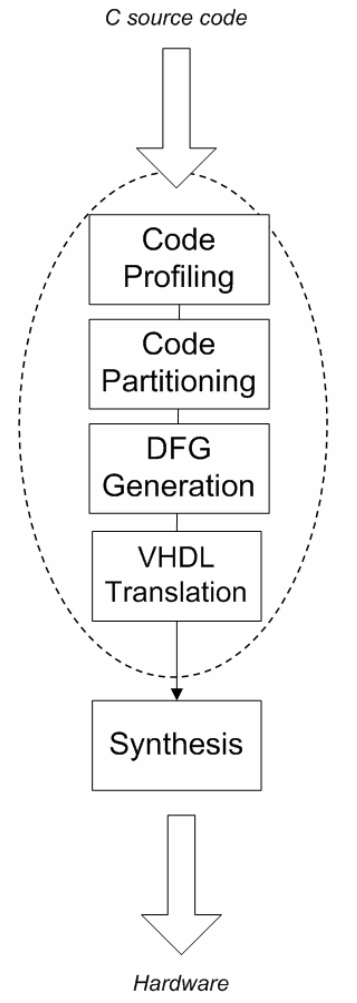


Figure 14. Four step process for the hand-design of SuperCISC hardware.

3.5.1 Candidate Selection and Code SW/HW Partitioning

Code profiling can identify, among other attributes, an application's ILP. The motivating assumption is that highly iterative applications such as those used in video encoding and signal processing support high levels of ILP that benefit from VLIW parallel computing. Candidate applications, however, often exhibit less than 2x average ILP [32]. In such cases, even wide VLIWs performance optimized for shared data components cannot realize desired speedups.

Disappointing ILP, not parallel architecture constraints, is the limiting factor to gains achievable by homogeneous parallel processors.

Figure 15 charts the average ILP discovered by Trimaran compiler [30] for five benchmark applications within the MediaBench benchmark set [32]. Little additional ILP is discovered by the compiler by increasing the compiler parameter for processing width from 4-wide to an arbitrarily-wide processor. The poor ILP discovered within these benchmark applications limits the speedup of a 4-wide VLIW to less than half its ideal of 4x over a single-issue processor running at the same speed.

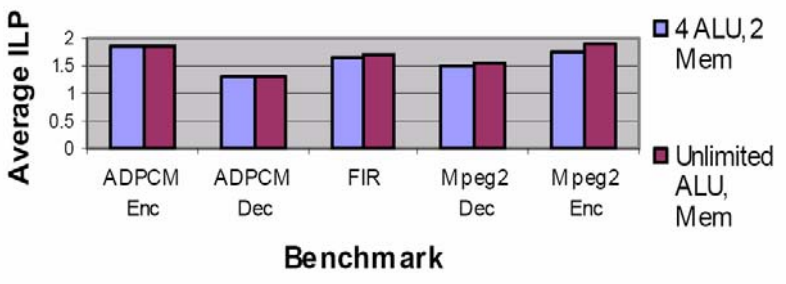


Figure 15. Instruction-level parallelism (ILP) of MediaBench benchmark applications discovered by Trimaran compiler given parameters for 4-wide VLIW with 2 memory ports and unlimited-wide VLIW with unlimited memory ports [32]

Because ILP is the critical factor limiting the gains of parallel architectures, further code analysis shows the program attributes which can lead to alternative speedups. Following the general rule that a program spends 90% of its time executing 10% of its code, the Shark profiler [31] used in this study can identify critical, time-consuming loop iterations within software source code. In an ideal implementation, minimizing the execution time of kernel portions of the code that occupy 90% of execution time would result in a 10x speedup (Equation 1), bringing the effective performance 167MHz processor to 1.67 giga-ops per second (GOPS).

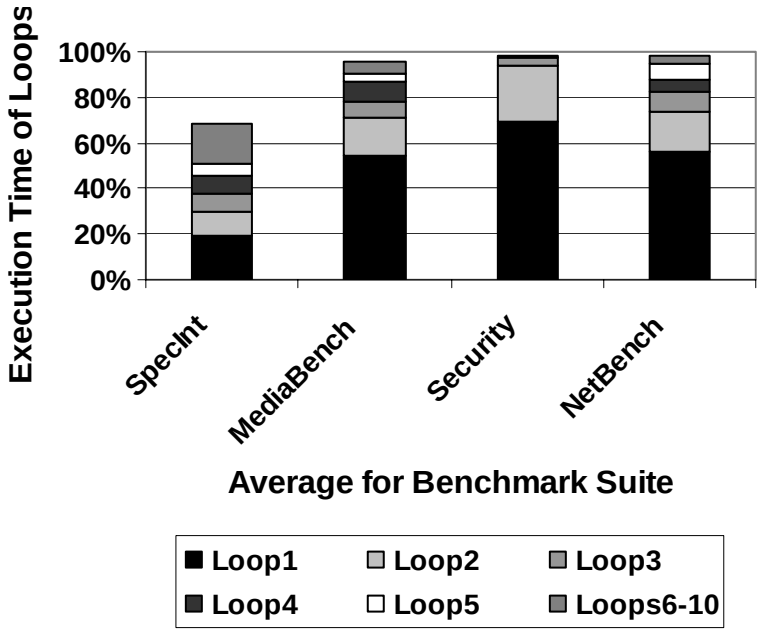


Figure 16. Execution time occupied within the top 10 loops in the code averaged across the SpecInt, MediaBench, and Netbench suites, as well as selected security applications [37]. Time-intensive loops are directed to SuperCISC hardware.

Figure 16 shows the percentage of execution time occupied by critical kernels of 4 benchmark application sets. The MediaBench benchmark set was used to generate the results presented in this thesis. Application kernels that together contribute to 90% or more of the application execution time are selected as candidate functions for SuperCISC co-processing hardware.

Using the results of the profiler, the SuperCISC design flow proceeds to the next step and the code is partitioned by hand into software and hardware segments using *pragma* statements to denote exclusion of the hardware portions of the code in the VLIW compiler. The application kernels inside the pragma statements are then isolated for manual DFG generation.

3.5.2 Data Flow Graph Generation

DFG generation from an application kernel is a process easily facilitated by hand although design automation using the intermediate representation (IR) of the source code will enable rapid graph generation. The DFGs use a library of nodes to represent basic operations in the source code. Arithmetic operations map to a node representing the operation within a hardware element. SuperCISC DFGs vary slightly from conventional DFGs in that they contain multiplexers to incorporate control flow into the graph. The presence of control structure, such as *if-then-else* statements in the source code indicate the presence of multiplexers in the SuperCISC DFG. Figure 17 shows the high-level C code for a candidate SuperCISC hardware function that performs a kernel portion of ADPCM encoder [32]. Note the presence of many *if-then-else* statements that indicates a control-intensive kernel and the instantiation of many multiplexers in the resulting DFG. Thus, the ADPCM encoder kernel will benefit from a high proportion of *control-flow efficiency* to *cycle compression* gains in the SuperCISC function kernel speedup.

```
1  // Begin Hardware Function
2  if ( bufferstep ) {
3    delta = inputbuffer & 0xf;
4  } else {
5    inputbuffer = *inp++;
6    delta=(inputbuffer >> 4) & 0xf;
7  }
8  bufferstep = !bufferstep;
9  index += indexTable[delta];
10  if ( index < 0 ) index = 0;
11  if ( index > 88 ) index = 88;
12  sign = delta & 8;
13  delta = delta & 7;
14  vpdiff = step >> 3;
15  if ( delta & 4 ) vpdiff+=step;
16  if ( delta & 2 ) vpdiff+=step>>1;
17  if ( delta & 1 ) vpdiff+=step>>2;
18  if ( sign )
19    valpred -= vpdiff;
20  else
21    valpred += vpdiff;
22  if ( valpred > 32767 )
23    valpred = 32767;
24  else if ( valpred < -32768 )
25    valpred = -32768;
26  step = stepsizeTable[index];
27  // End Hardware Function
28  *outp++ = valpred;
```

Figure 17. C-language software code for kernel portion of ADPCM encoder [32].

Each SuperCISC hardware function represents one or more iterations of a time-intensive kernels of the application. For applications with more than one kernel contributing to the bulk of execution time, more than one SuperCISC function can be invoked to contribute gains derived from the 90/10 rule. If there is loop-control surrounding a kernel functions, SuperCISC hardware can in-line the loops to form a single, large datapath. Data-dependencies between looped application kernels can produce cascading SuperCISC functions where the dependent function interconnects with the preceding SuperCISC function at the dependent datapath join.

Edges within a SuperCISC DFG indicate data dependencies and sequencing. In an FPGA, nodes can be supported by either combinational logic within the logic elements or by the embedded multiply-accumulate cores. Edges indicate the routing between the FPGA resources. Figure 18 shows the resulting data flow graph for the ADPCM encoder kernel presented in

Figure 17. Note that the DFG contains many multiplexers to represent the *if-then-else* control statements within the source code. The datapaths preceding the multiplexer and connected as inputs represent the possible outcome sequences of the branch statement. The arithmetic nodes showing constant values as an input utilize the efficiency of hardware to optimize for constant-value computation.

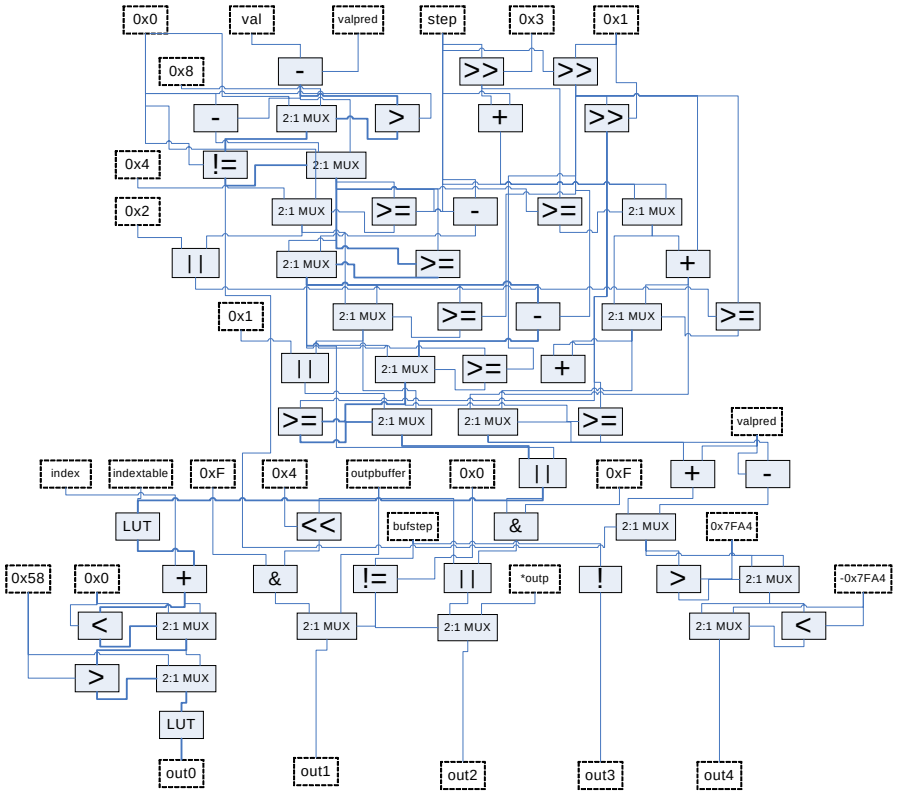


Figure 18. Data flow graph (DFG) representing SuperCISC hardware function for kernel portion of ADPCM encoder, shown in Figure 17.

The inverse discrete cosine transform (IDCT) benchmark application contains execution-time intensive kernels inside the row and column operations performed upon an 1x8 and 8x1 block of values, respectively. The IDCT benchmark application has two kernel functions, the row and

column operations, that comprise the bulk of its execution time. Therefore, the application will require two SuperCISC hardware functions to pursue the 90/10 rule to supplement the gains of the VLIW. The source code for IDCT column operation is shown in Figure 19. Note the absence of *if-then-else* statements that indicates a SuperCISC hardware function that will realize its kernel speedups from the efficiency of asynchronous computation and *cycle-compression*, explained in detail in Section 4.2.

```
1.    if (!((x1 = (blk[8*4]<<8))
2.       | (x2 = blk[8*6]) | (x3 = blk[8*2])
3.       | (x4 = blk[8*1]) | (x5 = blk[8*7])
4.       | (x6 = blk[8*5]) | (x7 = blk[8*3]))){
5.         blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=
6.         blk[8*4]=blk[8*5]=blk[8*6]=blk[8*7]=
7.         iclp[(blk[8*0]+32)>>6];
8.         return;
9.      }
10.   x0 = (blk[8*0]<<8) + 8192;
11.   /* first stage */
12.   x8 = W7*(x4+x5) + 4;
13.   x4 = (x8+(W1-W7)*x4)>>3;
14.   x5 = (x8-(W1+W7)*x5)>>3;
15.   x8 = W3*(x6+x7) + 4;
16.   x6 = (x8-(W3-W5)*x6)>>3;
17.   x7 = (x8-(W3+W5)*x7)>>3;
18.   /* second stage */
19.   x8 = x0 + x1;
20.   x0 -= x1;
21.   x1 = W6*(x3+x2) + 4;
22.   x2 = (x1-(W2+W6)*x2)>>3;
23.   x3 = (x1+(W2-W6)*x3)>>3;
24.   x1 = x4 + x6;
25.   x4 -= x6;
26.   x6 = x5 + x7;
27.   x5 -= x7;
28.   /* third stage */
29.   x7 = x8 + x3;
30.   x8 -= x3;
31.   x3 = x0 + x2;
32.   x0 -= x2;
33.   x2 = (181*(x4+x5)+128)>>8;
34.   x4 = (181*(x4-x5)+128)>>8;
35.   /* fourth stage */
36.   blk[8*0] = iclp[(x7+x1)>>14];
37.   blk[8*1] = iclp[(x3+x2)>>14];
38.   blk[8*2] = iclp[(x0+x4)>>14];
39.   blk[8*3] = iclp[(x8+x6)>>14];
40.   blk[8*4] = iclp[(x8-x6)>>14];
41.   blk[8*5] = iclp[(x0-x4)>>14];
42.   blk[8*6] = iclp[(x3-x2)>>14];
43.   blk[8*7] = iclp[(x7-x1)>>14];
```

Figure 19. C-language software code for IDCT column operation [32].

IDCT column source code is comprised entirely of arithmetic operations and a large number of shifts by a constant value. The DFG contains no multiplexers for *if-then-else* and therefore has no unused datapaths of branches not-taken to consume energy and logic area. The resulting DFG for IDCT column operation is shown in .

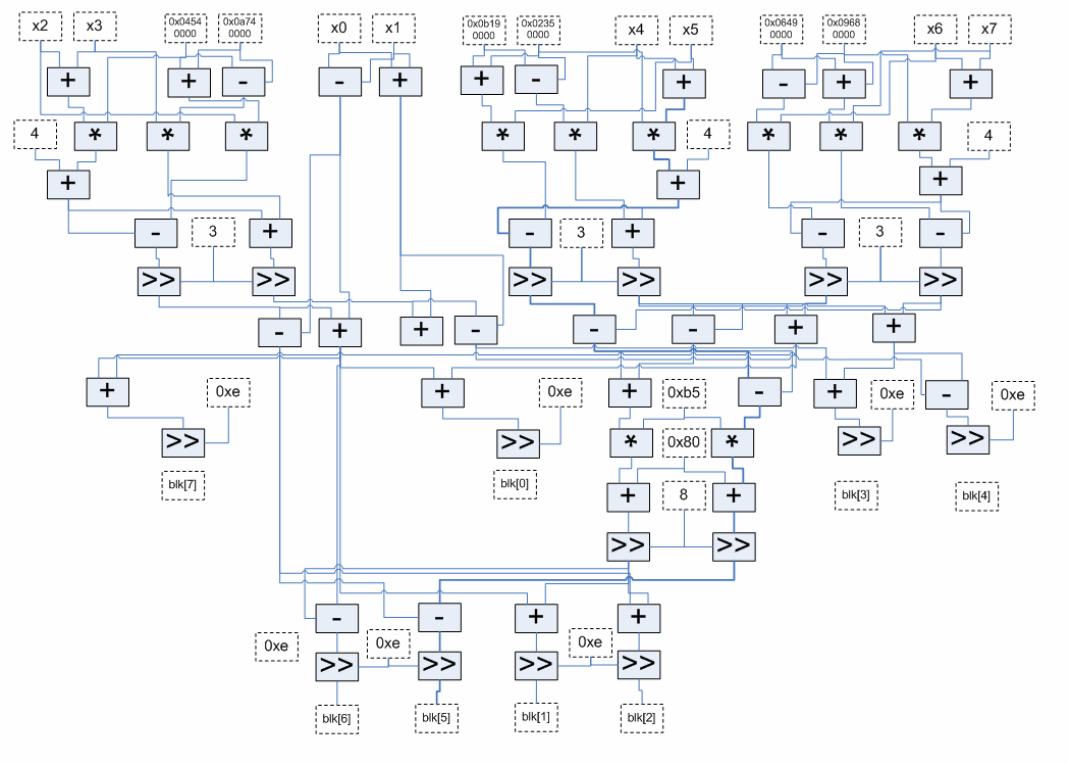


Figure 20. Data flow graph (DFG) representing SuperCISC hardware function for IDCT column operation, shown in Figure 19.

3.5.3 DFG to VHDL Conversion

From the DFGs, hardware modeling language such as VHDL is used to map the graph into a high-level circuitry representation. This step is currently undergoing design automation at the University of Pittsburgh. VHDL design modules represent the nodes in the DFGs and the edges

indicate routing and signal dependency in the FPGA hardware implementation. Figure 21 and Figure 22 show the entity and port declaration for ACPCM encoder and IDCT column, respectively. Note the reduction of input variables in IDCT column from the C source code variables to indicate hard-coding of the constant values.

```
ENTITY ADPCM_encoder IS
   PORT(
      bufstep_in : IN     std_logic_vector (31 DOWNTO 0);
      index_in   : IN     std_logic_vector (31 DOWNTO 0);
      outbuf_in  : IN     std_logic_vector (31 DOWNTO 0);
      outp_in    : IN     std_logic_vector (31 DOWNTO 0);
      step_in    : IN     std_logic_vector (31 DOWNTO 0);
      val_in     : IN     std_logic_vector (31 DOWNTO 0);
      valpred_in : IN     std_logic_vector (31 DOWNTO 0);
      reg_out    : OUT    std_logic_vector (31 DOWNTO 0);
      reg_out1   : OUT    std_logic_vector (31 DOWNTO 0);
      reg_out2   : OUT    std_logic_vector (31 DOWNTO 0);
      reg_out3   : OUT    std_logic_vector (31 DOWNTO 0);
      reg_out4   : OUT    std_logic_vector (31 DOWNTO 0)
   );

END ADPCM_encoder ;
```

Figure 21. Entity and port declarations for ADPCM encoder DFG in Figure 18.

```
ENTITY IDCT_col IS
   PORT(
      x0_in : IN     std_logic_vector (31 DOWNTO 0);
      x1_in : IN     std_logic_vector (31 DOWNTO 0);
      x2_in : IN     std_logic_vector (31 DOWNTO 0);
      x3_in : IN     std_logic_vector (31 DOWNTO 0);
      x4_in : IN     std_logic_vector (31 DOWNTO 0);
      x5_in : IN     std_logic_vector (31 DOWNTO 0);
      x6_in : IN     std_logic_vector (31 DOWNTO 0);
      x7_in : IN     std_logic_vector (31 DOWNTO 0);
      blk0  : OUT    std_logic_vector (31 DOWNTO 0);
      blk1  : OUT    std_logic_vector (31 DOWNTO 0);
      blk2  : OUT    std_logic_vector (31 DOWNTO 0);
      blk3  : OUT    std_logic_vector (31 DOWNTO 0);
      blk4  : OUT    std_logic_vector (31 DOWNTO 0);
      blk5  : OUT    std_logic_vector (31 DOWNTO 0);
      blk6  : OUT    std_logic_vector (31 DOWNTO 0);
      blk7  : OUT    std_logic_vector (31 DOWNTO 0)
   );

END IDCT_col ;
```

Figure 22. C- and port declarations for IDCT column DFG in Figure 20.

The VHDL file is then compiled, synthesized and routed to connect the configurable logic elements that represent the SuperCISC hardware. Synthesis software, Synplify Pro or Precision, compiles the design into a file called a technology-specific netlist. Altera's Quartus II place-and-route accepts the netlist as an input and performs a simulated annealing placement algorithm to map the netlist to the FPGA. Place-and-route software furthermore analyzes the configuration to report timing characteristics, logic area utilization, memory occupation and DSP consumption.

4.0 SPEEDUP OF SUPERCISC HARDWARE FUNCTIONS

SuperCISC co-processing hardware performs complex computation in deep, highly parallel asynchronous datapaths. The SuperCISC implementation of software-base control flow as hardware-based data flow promotes *control flow efficiency*, explained in Section 4.1. The asynchronous computational units within SuperCISC hardware promotes *cycle compression*, explained in Section 4.2, a reduction of the clock-period waste suffered by many types of computation executed within a synchronous processor. The two main sources of speedup in a SuperCISC hardware function, control flow efficiency and cycle compression, allow speedups that exceed those achieved by conventional parallelization approaches. The speedups of SuperCISC hardware surpass those given by the application kernel's ILP. Using Amdahl's equation, applying SuperCISC hardware to the run-time intensive portions of an application raises speedups beyond the 4x factor of the VLIW functioning at 100% utilization on 100% of the code.

4.1 CONTROL FLOW EFFICIENCY

SuperCISC hardware functions can add efficiency to control flow, the *if-then-else* statements that control an application's execution pattern. In hardware, multiplexers control the flow of data through a function. Concurrently executing datapaths represent each possible outcome of a

control statement. The datapaths feed into a multiplexer that has a select line, controlled by the branch comparison, to choose among the inputs. The concept of pre-calculating branch outcomes as multiplexed datapaths is known as predication.

In software, an *if-then-else* statement in C-code is expressed in a stream of assembly instructions that include moves, subtractions and comparison-based branches. C code for an *if-then-else* statement with its hardware implementation as a 2:1 multiplexer is shown in Figure 23. The corresponding set of assembly instructions for the *if-then-else* statement is shown in Table 2.

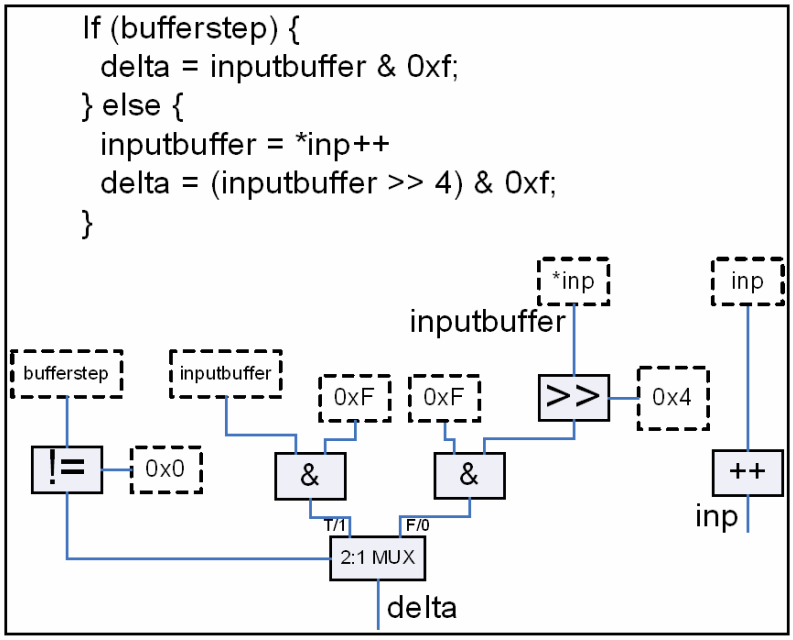


Figure 23. Data flow graph (DFG) showing 2:1 multiplexer that represents C-code for an *if-then-else* statement.

Table 2. Assembly instructions of *if-then-else* statement (shaded in grey)

Instruction	Description
MOV R1, 0	moves comparison value (zero) into R1
CMP R31, R1, R2	Compare if R2 (*bufferstep*) == 0 and store the result in R31
BNE R31, #1, (PC + #5)	Branch if comparison is false -- if R2 (*bufferstep*) != 0
ADD R6, R5, 1	Else if R2 (*bufferstep*) == 0, increment input and store in *inputbuffer*
RSL R6, #4	Right shift *inputbuffer* by 4
AND R7, R6, #0xF	Logical AND *inputbuffer* by constant
BR (PC + #2)	Skip to next unconditional execution
AND R4, #0x15	If R2 (*bufferstep*) != 0, execute this instruction
MV Rx, #immed	Next unconditional execution

A multiplexer can execute control flow significantly faster than a software equivalent executing on a single processor. On an EP2S180, a 32-bit 2:1 multiplexer supporting a 1-way comparison (LT, GT, NE) with default condition can operate at 322MHz, and a 2-way comparison (LTE, GTE) with default condition clocks in at 206MHz. The same operation consumes a minimum of 4 cycles when executed in software, reducing the effective performance of a 167MHz single processor to 42MHz. This results in a 11x speedup of a multiplexer with a 2-way comparison over a single processor.

A 4-way VLIW can execute an *if-then-else* statement in slightly fewer cycles than a single-issue CPU. A 2:1 multiplexer exhibits a 4x speedup over a 4-way VLIW to execute the same operation in software. A comparison of the number of cycles to execute a 2:1 multiplexer equivalent is shown in Figure 24.

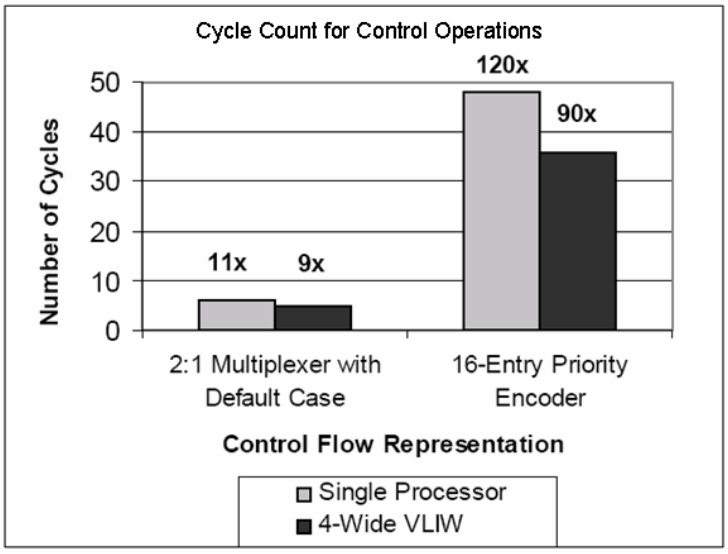


Figure 24. Cycle count for *if-then-else* statement executed on a single processor and on a 4-way VLIW. Speedup of 2:1 multiplexer hardware implementation of the same control flow control flow is shown in bold above the cycle count.

The speedup of *if-then-else* statements in hardware is also due the full rendering of branch prediction. Without branch prediction, a VLIW *branch-not-taken* outcome flushes a minimum of 3 instructions from the pipeline, wasting 3 cycles. SuperCISC hardware, however, exhibits fully rendered branch prediction by saturating multiplexer inputs with complete sequences of all possible outcomes.

A wide multiplexer indicates a burst of parallelism within a CISC datapath. Complex branch instructions with more than 2 possible outcomes can be fully predicated by a wide multiplexer. This level of parallelism for complex control flow is supported by the abundance of configurable logic area on the FPGA for scaling the multiplexer.

Performance gain of executing complex control flow in SuperCISC hardware is supported by the results of a 16-condition priority encoder used in the G721 kernel. A value is compared to a constant set of 16 values, or bins, and the appropriate output routed accordingly.

In software, this operation consumes 48 instructions, as shown in Table 3. A 16-entry priority encoder implemented in Stratix II FPGA hardware executes a control statement 120x faster than the same control statement implemented in software to run on a single-issue processor at 167MHz on the same FPGA platform.

Table 3. Assembly instructions to execute 16-entry priority encoder in software

Instruction	#Needed	Description
MOV R_x, Immed_i	16x	moves comparison value into R_x
SUB R31, R_i, R_x	16x	Subtracts comparison value from variable in register R_i and stores result in R31
BE R31, 0	16x	Branches if R31 equal to zero

A 16-entry priority encoder with fixed comparisons has a performance of 422MHz on an Altera EP2S180, the maximum clock speed of any logic element on the chip. Assembly code for a 16-entry priority encoder consumes 48 cycles reducing the effective performance of a 167MHz single processor to 3MHz. A 4-wide VLIW can implement the priority encoder in 36 cycles for an effective fMax of 5MHz.

4.2 CYCLE TIME COMPRESSION

SuperCISC hardware realizes another contribution to its speedup through back-to-back compression of synchronous arithmetic nodes. Even in a pipelined processor architecture that executes one operation per cycle, some arithmetic functions, such as shift by constant value, are

relatively simple bit manipulations and complete within a fraction of the 6ns clock period on the 167MHz VLIW. The remaining portion of the cycle is spent waiting for the slowest path of the architecture to execute.

Minimizing idle cycle time with respect to standard processor architectures is called *cycle time compression.* Cycle time compression is what allows a SuperCISC function to execute a sequence of arithmetic operations at a fraction of latency for the same sequence performed on a pipelined RISC processor. The 4-way VLIW proposed in this thesis executes all arithmetic operations at a constant rate of 167MHz. A SuperCISC function, however, can execute an arithmetic operation according to the node's maximum clock speed within the chip before proceeding to the next node.

On an EP2S180, a 32-bit adder clocks in at 346MHz, a 2x increase over the performance of the same operation executed on a 167MHz processor targeted to the same device. Table 4 shows the clock speed of some constituent nodes of hardware functions if they had registered inputs and outputs and were functioning in isolation on the Altera EP2S180 FPGA. Note that operations having one fixed operand exhibit faster performance than those having two unfixed operands. All operations are capable of executing at a clock speed that exceeds that of the 167MHz VLIW.

Table 4. Table of SuperCISC node isolated for performance and area utilization on an Altera EP2S180F

Operation (32-bit operands)	Area (LUTs)	Performance in Isolation (MHz)	Latency (ns)
Adder	96	346	3
Subtractor	96	337	3
Multiplier	0	322	3
Variable Shift	135	282	4
Fixed Shift	48	422	2
2-Way Variable Comparator	76	241	4
1-Way Variable Comparator	66	332	3
1-Way Fixed Comparator	33	384	3
1-Way Fixed Comparator	25	386	3
Logical INV	64	366	3
Logical AND/OR/XOR	96	422	2
89-Element LUT	408	229	4
16-Element Priority Encoder	47	422	2
2:1 Multiplexer	64	422	2

SuperCISC hardware combines the computational nodes isolated for performance in Table 4 in an asynchronous combination within a multi-cycle path relative to the VLIW. The performance savings of a SuperCISC hardware function over a synchronous CPU can be estimated. First, for the software-implementation latency, multiply the number of assembly instructions for the function by the cycle time of the processor. Next, for the hardware-based implementation, estimate the latency for each datapath of the SuperCISC hardware. Do this by summing the known latencies for each arithmetic node along each edge of the graph from input to output. Choose the longest latency to represent the critical path. Subtract the SuperCISC hardware

latency from the CPU latency. Divide this difference by the CPU cycle time to determine the number of cycles saved to perform the operation within a SuperCISC hardware function.

Figure 25 shows the cycle time utilization of SuperCISC computational nodes if they were functioning in isolation relative to the cycle time of a 167MHz CPU on the same target device. Logic operations realize the most benefit from SuperCISC hardware, with the fixed-operand shift achieving the most compression. Note that multipliers require no logic area utilization due to their implementation within digital signal processing (DSP) blocks. A 32-bit multiplier requires 8 DSP units, or 1% of the 768 total 9x9 multipliers available on an EP2S180.

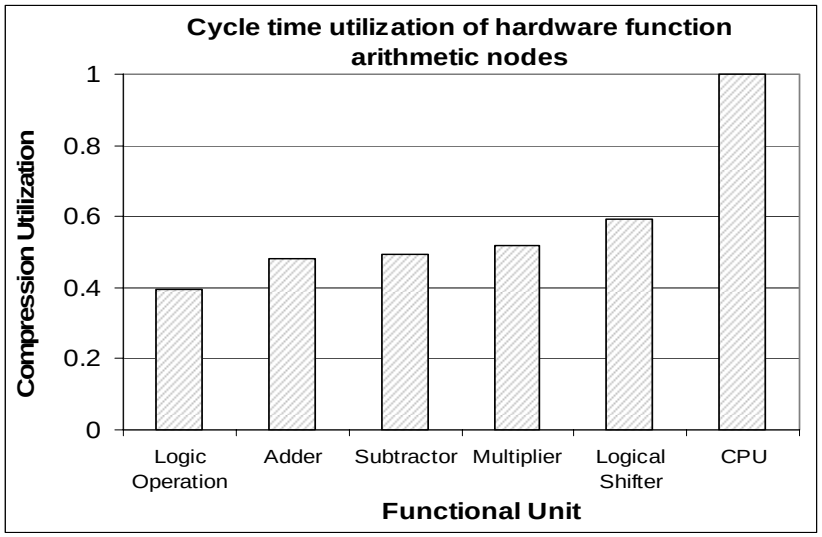


Figure 25. Cycle time utilization of SuperCISC arithmetic nodes normalized to the cycle time of a 167 MHz processor on an Altera EP2S180

SuperCISC computational nodes realize the most cycle compression from fixed operand computation. A fixed operand computation can execute up to 2.5x faster than its 2-variable equivalent. Figure 27 shows the speedup of 1-fixed operand comparators and logical shifters as compared to equivalent 2-variable operations on a 167MHz VLIW versus a SuperCISC

computational node. The figure indicates that a computation having one fixed operand (i.e. shift operand by a constant value of 4) executes in hardware about 1.5x faster than a similar operation having no fixed operands (i.e. shift operand by a variable value).

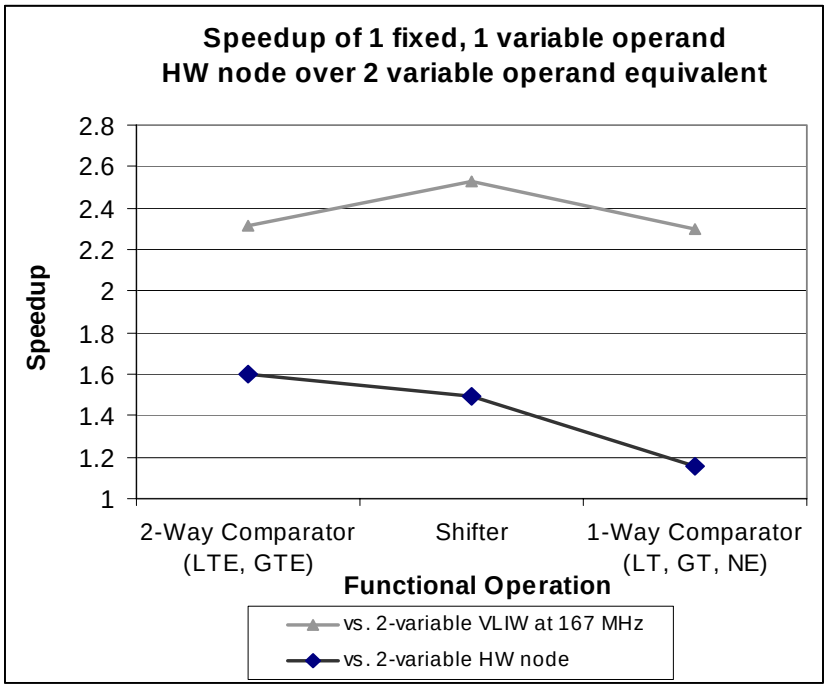


Figure 26. Speedup of fixed operand computation executed in SuperCISC computational node versus 2 variable computation executed in SuperCISC node and versus execution on 167 MHz VLIW on an Altera EP2S180

5.0 SYSTEM MODELING

The VLIW/SuperCISC software/hardware processor was designed for synthesis in VHDL and for run-time and behavioral information using C++ and the Synopsys SystemC library. The VHDL model provides the encoded information that allows the VLIW/SuperCISC architecture to be compiled for configuration of the FPGA. The same VHDL files could also be used for a custom-ASIC implementation. The SystemC encoded architecture is used for collecting statistics of the run-time behavior of the benchmark application.

5.1 VHDL MODELING

The VLIW/SuperCISC software/hardware architecture was designed in VHDL hardware description language. VHDL stands for very high-speed integrated circuit VHSIC hardware description language [34][35]. VHDL is used for both synthesizable and behavioral modeling of digital electronic hardware. VHDL is regulated by the Institute of Electrical and Electronics Engineers (IEEE). IEEE 1164 defines the most current standard and standard logic libraries available to VHDL.

VHDL does not require knowledge of the underlying primitive FPGA logic used to implement the design. The VLIW/SuperCISC VHDL files are parsed within a technology-specific process called *synthesis*, in which VHDL text files are compiled into a gate-level

implementation called a *netlist*. The VLIW/SuperCISC processor was synthesized using Synplicity Synplify as well as Altera QuartusII software to compile a netlist. Analysis of the synthesized design can provide an estimate of the performance prior to place-and-route of the design.

The netlist is passed to place and route software issued by a device vendor, such as Quartus II for the Altera family of devices. Quartus II maps the netlist to a specified FPGA and configures row and column interconnection signals between LABs, RAM and DSP blocks. Quartus II analyzes the configuration, post-place and route, and reports static timing data and FPGA area and resource utilization. Designs for the VLIW and SuperCISC were hand-optimized through an iterative process. Figure 27 shows the design flow of the VLIW/SuperCISC software/hardware processor from conception, to VHDL modeling, to synthesis, place and route, repeated through iterative design optimizations.

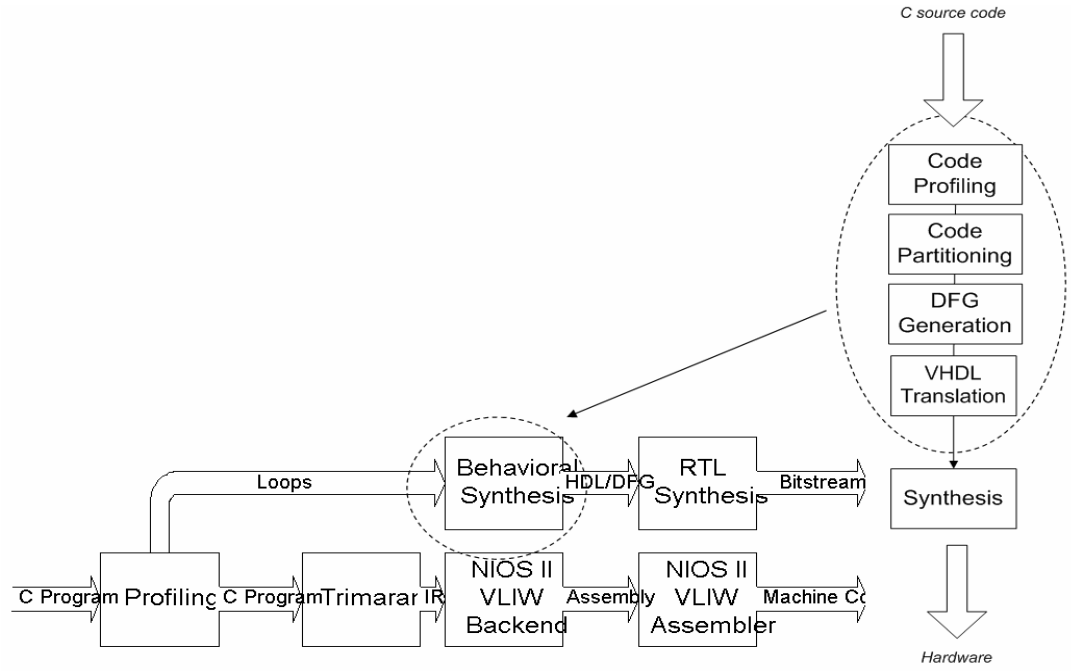


Figure 27. Design flow for VLIW/SuperCISC architecture [32][33].

5.2 SYSTEMC MODELING

The VLIW architecture was modeled at the system level in C-language source code using the Open SystemC Initiative (OSCI) *SystemC* hardware modeling library. The SystemC library extends C++ for modeling concurrent behavior, time-sequenced operations, hardware data-types, hierarchical system modeling and simulation [38]. SystemC library enables rapid design time of a behavioral model of the VLIW architecture. Run-time simulation and verification information of the system is reported to the standard console and recorded in output files. In addition to performing system-level run-time analysis, data gathered through a SystemC simulation can be used to explore issues of dynamic switching power-reduction for making design decisions about an application-driven architecture.

A hierarchical SystemC VLIW was created for this thesis as a cycle-accurate representation of the VHDL model. The SystemC VLIW consists of a main.cpp file that constitutes the top level of the design hierarchy. Instantiated classes within main.cpp represent the secondary tiers of the hierarchy and stages of the VLIW pipeline (instruction fetch, decode, execute, memory, and register file writeback). A display class is instantiated to read VLIW system signals, collect and interpret the information, and display the information to the user.

The SystemC VLIW processor provides data about an application's run-time profile. The SystemC VLIW simulator outputs counts of occurrences of different instruction types (memory load/store, arithmetic, control) recorded from the run-time behavior of the application. Run-time information of the target application can support the results of the code profiler when designing an application-driven architecture.

Figure 28 shows the run-time profile of application instruction-types for several benchmarks compiled for the VLIW SystemC VLIW. Note the predominance of memory load/store instructions that point to the value of implementing a shared address space in main memory for the SuperCISC hardware. Identifying run-time instruction-type dominance within a target application helps to point to future directions of the VLIW project. Figure 28 illustrates wht motivation for shared memory architecture to accelerate loads and stores of a memory-intensive applications.

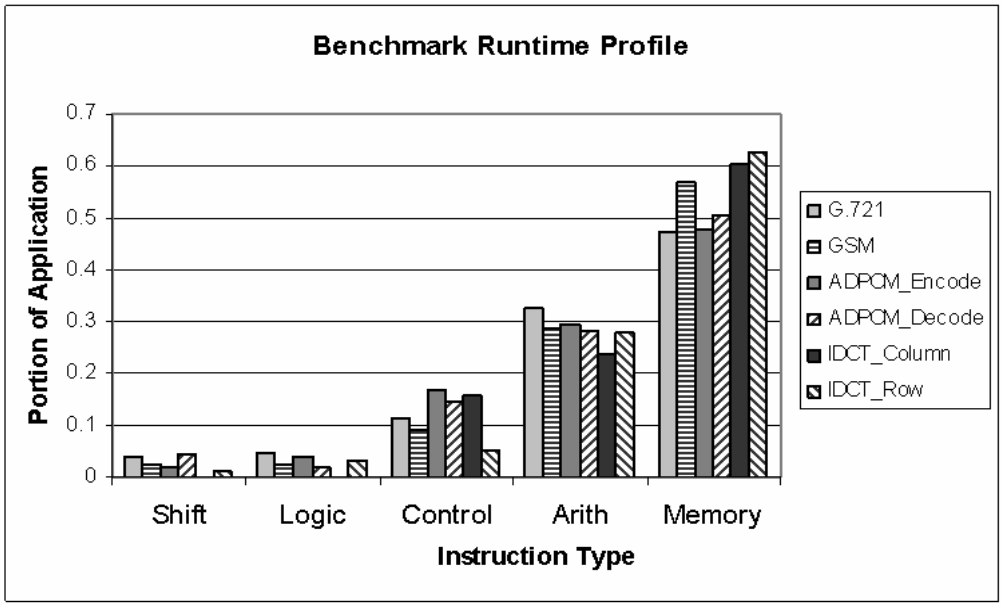


Figure 28. Run-time profile of benchmark applications compiled for the VLIW processor only of the SystemC simulator model.

The SystemC library can be used to create a system-level simulator for collection of data about static and dynamic power consumption [39][40][41]. One way to collect data about the switching activity of the system to reduce power consumption is to link the SystemC bit-level data types to a switching counter. On each cycle, a *bittwiddle()* function would count the number

of bits that switch in a given set of signals. The simulator writes the count to a 2D array with a time stamp and the value. This data can be correlated to energy consumption by time period to estimate the dynamic switching power. Design decisions for power reductions could be based upon SystemC reports of the switching behavior for a specific application. A SystemC simulator can be used in future directions of the VLIW/SuperCISC processor in the area of power-reduction for application-driven architectures.

6.0 PERFORMANCE RESULTS

Performance results for the VLIW/SuperCISC software processor/hardware accelerant architecture were obtained using Quartus II's post-place and route static timing analysis of the Synplicity synthesized netlist. The VLIW processor was isolated from the SuperCISC hardware and analyzed to determine a baseline performance for the VLIW processor on an Altera Stratix II EP2S180 FPGA.

The SuperCISC hardware functions were analyzed independently of one another with the understanding that their complexity would lead to longer latencies that require multiple VLIW clock cycles for the SuperCISC co-processors to complete execution. Static timing analysis measures the latency and slack time from the output of a clocked register, through any combinational logic, RAM block, or embedded multiplier, to the input of the next clocked stage. The path of the data vector showing the longest latency is called the *critical path* of the design. The critical path dictates the maximum clock frequency of a design sharing a clock tree.

6.1 VLIW PERFORMANCE

Iterations of the design flow have produced a 4-way VLIW with 32-bit NiosII extensible RISC ISA processors to achieve a peak performance of 167MHz. A program that can support an ILP of 4 at all times would execute on the 4-way VLIW at a rate of 668MOPS. This throughput

would raises the effective speed of the 4-way VLIW to exceed that of many embedded processors such as the ARM1020 and ARM11 [42].

Achieving optimal performance of the VLIW is not the objective of the performance profiling – achieving an *acceptable* performance (above 150MHz) was sought to attain a baseline performance for the VLIW software processor against which to compare the performance of SuperCISC hardware accelerants and evaluate their speedups.

6.1.1 VLIW Performance Profile

The performance profile of the VLIW shows the critical path to be between the 512Kb memory output to the register file, the write-back path of a LOAD operation. The 6ns latency from the memory bank to the register file could be cut by adding a register stage.

The next-to-critical path is set by the ROM decoder. In an optimal processor design on an FPGA, the maximum clock frequency would be set by the DSP blocks in which 32-bit integer multiplication takes place. DSP blocks are statically configured on the chip and therefore performance is only subject to the bit-width of the operation. Because multiplication on the VLIW is set to 32-bit inputs, the maximum performance cannot exceed 322 MHz.

Table 5. Timing results for paths within 4-way VLIW on Altera EP2S180

Path	Maximum Clock Speed	Latency	Stage of Pipeline
512Kb Memory Output to 32-element Register File Output	167 MHz	6ns	Writeback (LOAD)
Instruction ROM Output to Decoder Output	196 MHz	5ns	Decode
ALU Output to 32-element Register File Output	249 MHz	4 ns	Writeback (ALU)
ALU Output to 512Kb Memory Output	309 MHz	3ns	Memory (STORE)
Register File Output to Multiplier Output	318 MHz	3ns	Execute (MULT)

Table 5 shows the maximum clock speed given by the critical path at the top of the table. Timing results for other paths within the VLIW show latency for other stages of the VLIW pipeline. The performance of a 32-bit integer multiplier is shown as reference for the ideal performance of the VLIW.

6.1.2 Area and Resource Utilization

The proposed VLIW utilizes only 4% of the total logic cells of an EP2S180 FPGA, 25% of which is allocated to the shared register file. A single NiosII processor occupies 2928 LUTs, while 4 Nios II processors in VLIW formation occupy only twice the area of a single processor, or 5932 LUTs. The remaining 96% of the EP2S180 logic area is available for SuperCISC hardware functions.

The VLIW occupies 72, or 8% of the DSP blocks on an EP2S180. It would appear that the width of a parallel processor is limited by the number of DSP blocks rather than configurable logic, but tests conducted for this thesis show that synthesis tools direct resources sharing that

over-saturates the DSP blocks with more multiply blocks than empirical limits could support. In cases where a design that demanded more DSP resources than embedded within the FPGA was targeted to the device, the place and route software

6.2 SUPERCISC HARDWARE PERFORMANCE

SuperCISC hardware co-processors were isolated from the VLIW processor and analyzed independently of one another to determine execution latency on an EP2S180 FPGA. The latency of the SuperCISC hardware is used as a measurement against which to compare the execution times of benchmark applications on a host processor sharing the same technology platform.

The latency of SuperCISC hardware is not subject to an optimization flow because of the inherent efficiency of the 32-bit design. SuperCISC hardware accelerants instantiate computational resources dictated by the data-flow graph of the kernel to be mapped, thereby minimizing the amount of resources dedicated to each task. The latency of some operations could be reduced by customizing bit-width to suit application-specific operations upon *short* (16 bit) and *char* (8 bit) data types.

The SuperCISC hardware functions created in representation of benchmark code kernels exhibit a latency that ranges from 16ns to 28ns and reduce sequences of complex operations to the equivalent of a few VLIW clock cycles. Table 6 shows the latency and number of VLIW clock cycles for the kernel to execute for six SuperCISC functions.

Table 6. Timing results for SuperCISC hardware execution of listed portions of benchmark applications on Altera EP2S180

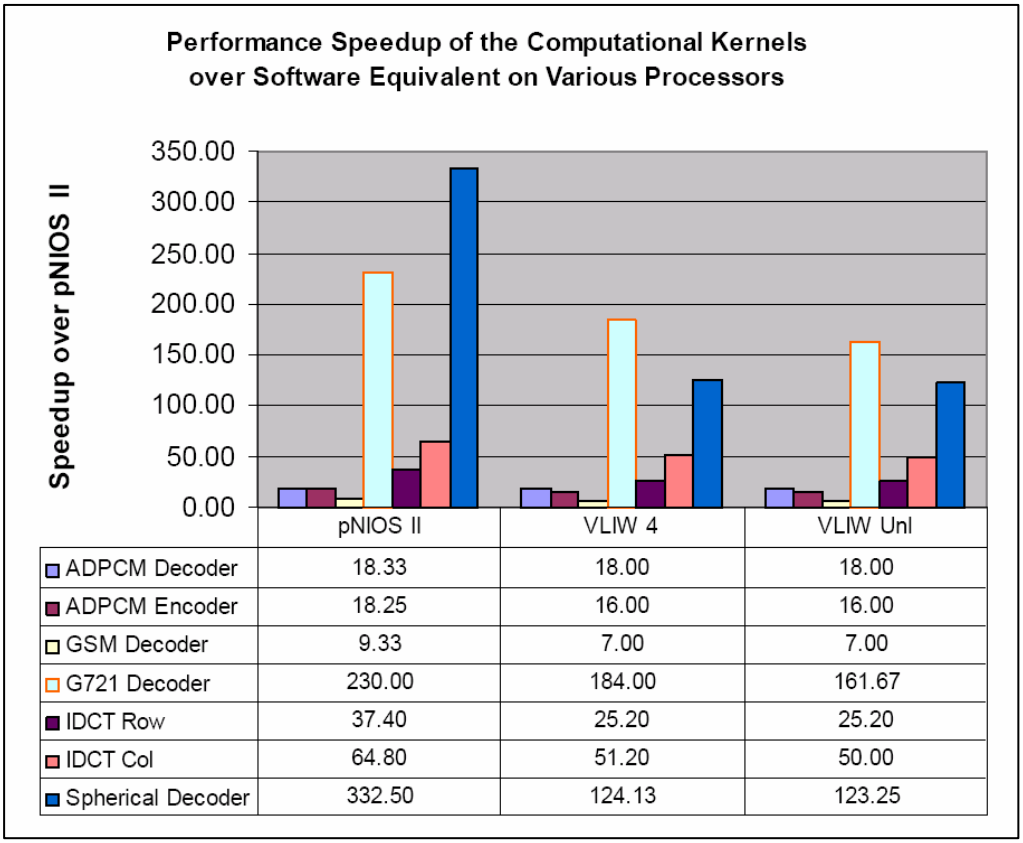
SuperCISC Function	Latency	VLIW stall cycles needed	# of VLIW instructions to execute same kernel	Speedup over VLIW
G.721 Decoder	24ns	4	736	184x
IDCT Column	22ns	4	204	51x
IDCT Row	21ns	4	100	25x
ADPCM Decoder	16ns	3	54	18x
ADPCM Encoder	28ns	5	80	16x
GSM Decoder	18ns	3	21	7x

6.2.1 Overall Application Speedups

SuperCISC hardware accelerants are designed to pursue gains of the 90/10 rule, targeting 90% of the execution time on a single-issue CPU by mapping the source code (often 10% of the source code) to hardware. Ideally, SuperCISC hardware exhibits 12x speedup to create an overall application speedup of 10x when applied to the VLIW, but many of the SuperCISC hardware profiled for this work exceed 12 speedup. Speedups in excess of 12x can be achieved by mapping kernels that together contribute more than 90% to the execute time on a single-issue CPU.

SuperCISC hardware functions have produced maximum kernel speedups of 330x over a single-issue 167MHz processor [32]. The speedup possible within a SuperCISC function is contingent upon the number of nodes, the complexity of the node functions, the amount of control flow that can be implemented to expand the candidate DFG, and the width of the input operand vector. Figure 29 shows the speedup of SuperCISC hardware executing a single

application kernel of a benchmark versus several 167MHz processor configurations targeted on the same FPGA technology platform executing the same kernel function in software.

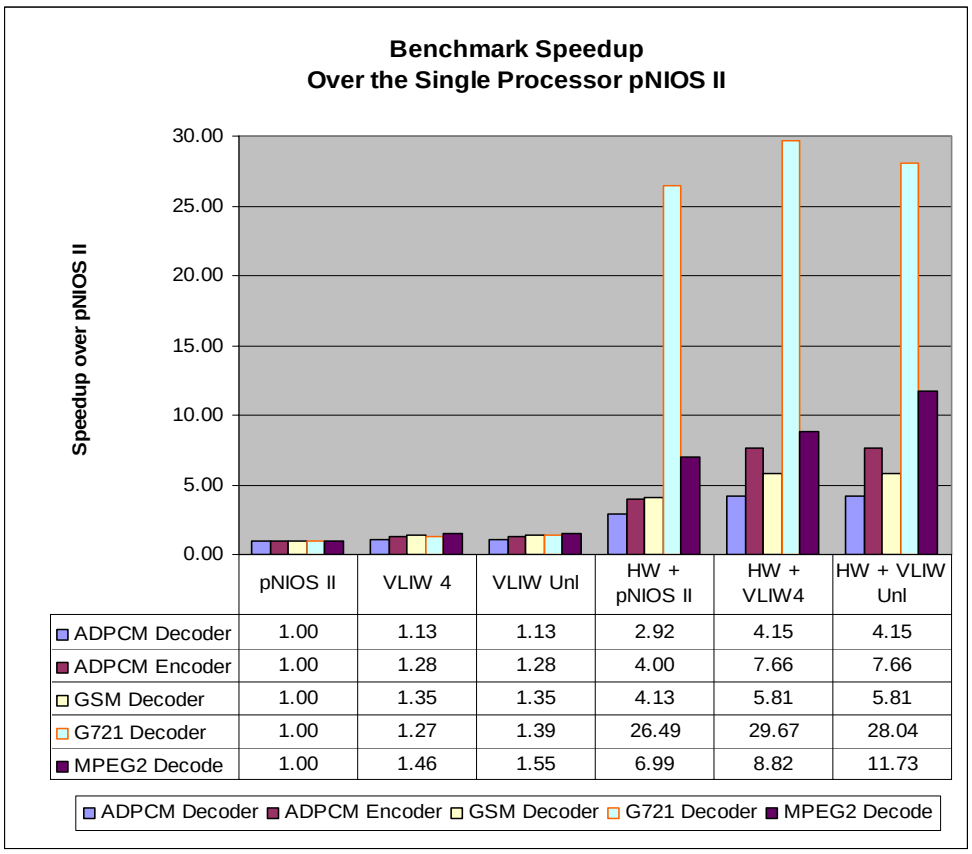


	pNIOS II	VLIW 4	VLIW Unl
ADPCM Decoder	18.33	18.00	18.00
ADPCM Encoder	18.25	16.00	16.00
GSM Decoder	9.33	7.00	7.00
G721 Decoder	230.00	184.00	161.67
IDCT Row	37.40	25.20	25.20
IDCT Col	64.80	51.20	50.00
Spherical Decoder	332.50	124.13	123.25

Figure 29. Speedup for single application kernels of SuperCISC functions versus several 167MHz processor architectures [32].

SuperCISC hardware functions raise the application speedup of a 4-way VLIW from 1.3x, less than half its ideal, to an average 10.2x speedup. This factor exceeds the arbitrary target speedup of 10x selected for the hybrid architecture. This target speedup is calculated by Amdahl's equation, inserting variables for a VLIW accelerating 10% of the single-issue run-time by a factor of 4x, coupled with SuperCISC hardware accelerating 90% of the single-issue run-time by

a factor of 12x. Figure 30 summarizes application speedup given by architectures from a 167MHz single-issue CPU to a 167MHz 4-way VLIW supported by SuperCISC hardware.



	pNIOS II	VLIW 4	VLIW Unl	HW + pNIOS II	HW + VLIW4	HW + VLIW Unl
ADPCM Decoder	1.00	1.13	1.13	2.92	4.15	4.15
ADPCM Encoder	1.00	1.28	1.28	4.00	7.66	7.66
GSM Decoder	1.00	1.35	1.35	4.13	5.81	5.81
G721 Decoder	1.00	1.27	1.39	26.49	29.67	28.04
MPEG2 Decode	1.00	1.46	1.55	6.99	8.82	11.73

Figure 30. Overall application speedup of various processor configurations compared to single-issue 167MHz architecture [32].

6.2.2 Area Utilization

All benchmarked SuperCISC functions utilize less than 1% of the logic area on an Altera EP2S180. Given the small area requirements of SuperCISC hardware, it is more meaningful to look at the logic area with respect to speedup. Figure 31 rank orders the area utilization and

kernel speedup of SuperCISC hardware accelerants, normalized to the area and speedup of the SuperCISC co-processor having the fewest logic elements, the G.721 decoder with 780 logic elements.

6.3 SUPERCISC SPEEDUP VERSUS AREA INCREASE

Hardware functions demand relatively little area compared with the potential for application speedup. Expensing an average FPGA logic area increase of 0.2x to the proposed VLIW, SuperCISC hardware functions contribute to a 10.2x speedup of the total application. In contrast, to achieve a 1.3x speedup over a single processor, a 4-way VLIW requires a 3x increase in area. The benefit (speedup) to cost (area) ratio of a SuperCISC function is 17x over that of a 4-way VLIW architecture. Figure 31 shows the speedup and area factor of SuperCISC hardware functions for each benchmark application.

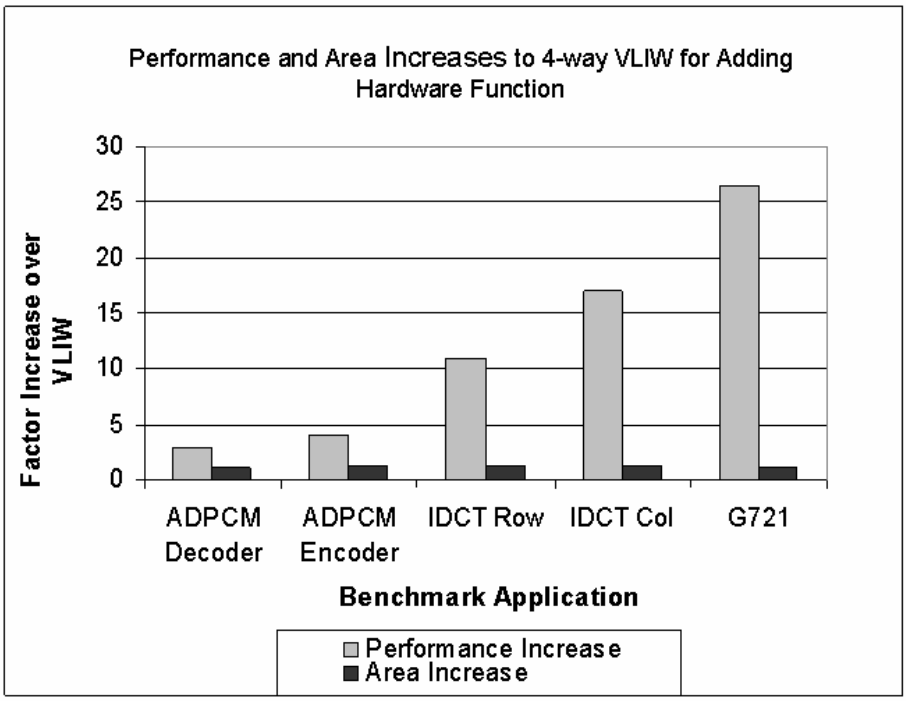


Figure 31. Performance and area increase for VLIW supported by SuperCISC hardware versus VLIW without SuperCISC support on an Altera EP2S180

Special on-chip resources such as embedded multipliers and multiply-accumulate (MAC) units are in limited supply, but devices such the EP2S180 boast 768 9-bit multipliers. IDCT column operation requires 72, or 9% of the multipliers, but less than 1% of the logic area. Empirical calculations indicate that the EP2S180 can support no more than 11 IDCT column hardware functions. Testing of the device limits, however, show that a configurable device can support many more functions than the number of multipliers would indicate. This effect is due to routing optimizations that direct resource sharing. Therefore, the truest indication of an FPGA's capacity to support quantities of SuperCISC hardware functions is given by the logic area utilization and not the occupation of embedded multipliers.

7.0 FUTURE DIRECTIONS

Work on the VLIW/SuperCISC software processor/hardware accelerated architecture shows many directions for future contributions to the area of application-driven computing. From the compiler perspective, the automated design flow can support configurable width of the VLIW software processor. The width of the VLIW can be passed as a parameter set by an ILP value of the target application. From the architecture perspective, memory data stores can supplement the register file as a shared data interface between software and hardware processing components. The following section provides data pursuant to using main memory as an additional interface between the VLIW and SuperCISC hardware.

7.1 SHARED MEMORY

Before designing the memory architecture for a highly-parallel processor, one must consider the system and application requirements for data capacity, parallel data access, and timing constraints. A memory must be capacious enough to hold data for an entire application. It must also provide data access to all the specified processors and perform at a rate to meet of exceed a baseline operating frequency, 167MHz in the case of our VLIW. Understanding the capabilities and limitations of the memory on a configurable logic device is critical to predicting synthesis results in creating a memory architecture that balances the competing requirements.

A memory designed for parallel processors may be either shared or distributed. A shared memory, sometimes called a global memory, is generally large and limited in bandwidth, but requires no coherence strategy. Multi-processor architectures that share main memory address space generally achieve better throughput than multi-processors that do not share an address space [36] and support parallel programming. A distributed memory, sometimes called a mobile memory, is generally governed by one main memory having one or more smaller, local memories and requires a coherence strategy. For a massively parallel processor to access all the data needed from memory, there is a tradeoff between the data access latency of a limited-bandwidth shared memory and the complexity of designing a coherence strategy for a distributed memory.

A shared memory may interleave access in time or provide real parallelism to incorporate the SuperCISC hardware functions into the shared address space. An architecture that supports time-interleaved memory sharing can be compiler driven, implementing interleaved access at the instruction level, or provide a conflict strategy to interleave access at the hardware level. In the current VLIW/SuperCISC architecture, writes of one processing unit may be read on the next cycle by all other processing units that have access to the shared address space.

A simple strategy for shared address-space sharing in which processing elements have a common instruction stream, stack and data implies that shared variables have the same meaning within the application thread to each software or hardware processor within the multi-processor architecture [36]. A more complex strategy for a shared memory architecture may provide for a global address space and partitions for private address spaces for hardware processing units and support of multi-threaded programming.

To meet capacity specifications, the memory design may instantiate several different memory block types available on the chip. The Altera Stratix II EP2S180 offers three kinds of memory: 930 M512 blocks with 512 bits plus parity, 768 M4K blocks with 4K bits plus parity, and 9 M-RAM blocks with 512K bits plus parity. The M512 and M4K blocks are widely distributed in columns throughout the chip, making them suitable for smaller, local memories in a distributed memory architecture. The M-RAM blocks provide the capacity for a shared memory.

Meeting memory access specifications is closely correlated to meeting timing specification. Therefore, is it useful to consider the effect of port-width on performance. Peak memory performance of 348 MHz is achieved on the EP2S180 by limiting utilization to access only one memory block. Increasing the M-RAM capacity and port size beyond a single block limit of 512Kbits and 64-bit dual ports requires striping across the blocks at some expense to the performance of the architecture. Similarly, the M4K blocks only achieve peak performance when limited to 4Kbits and dual 16-bit ports. Figure 32 below shows performance data for configurations of address space and dual-port size for the M4K and M-RAM block types. The shaded area indicates configurations that meet or exceed 167 MHz timing constraints for the VLIW/SuperCISC software/hardware processor architecture.

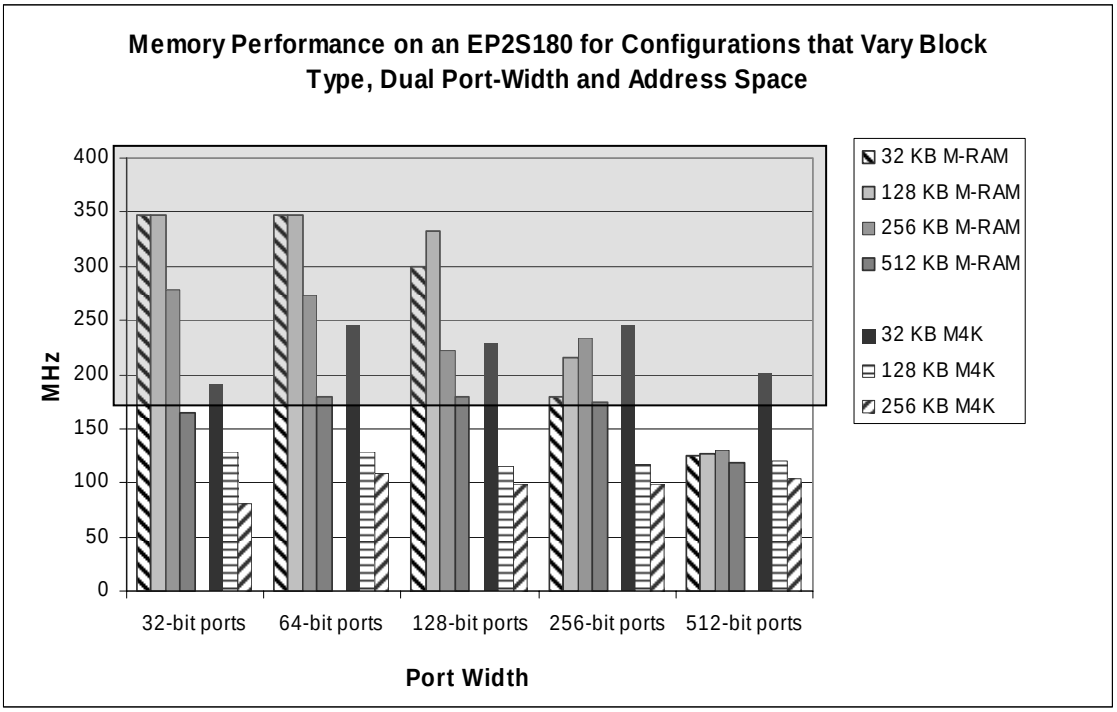


Figure 32. Memory performance on an EP2S180 for M-RAM and M4K blocks of varied dual-port size and address space. The shaded area indicates memory configurations that meet 167MHz timing constraints.

While widening port size may enable vector access to memory that meets timing constraints, the new configuration may not meet target memory 'bandwidth speedup.' Ideally, if a single 32-bit dual-ported memory performs at 100MHz, a 2x bandwidth speedup would indicate that a 64-bit dual ported memory performs at the same speed of 100MHz. This is often not the case. As a general rule, as the port-size increases on a memory of fixed-address space, the bandwidth 'speedup' falls short of the ideal gain. While this tradeoff is often not important for system designs that have a global clock and a single timing constraint, it does emphasize the sacrifice made to accommodate wide vector ports. Figure 33 shows the 'bandwidth speedup' of vectorized dual-ported memory configurations over that of a 32-bit dual-port memory having a base speedup of 1x. Speedups that exceed target are assumed to be due to random-variant, sub-

optimal, exit points of the Quartus II simulated annealing place-and-route algorithm for the 32-bit dual-ported memory referenced as a baseline.

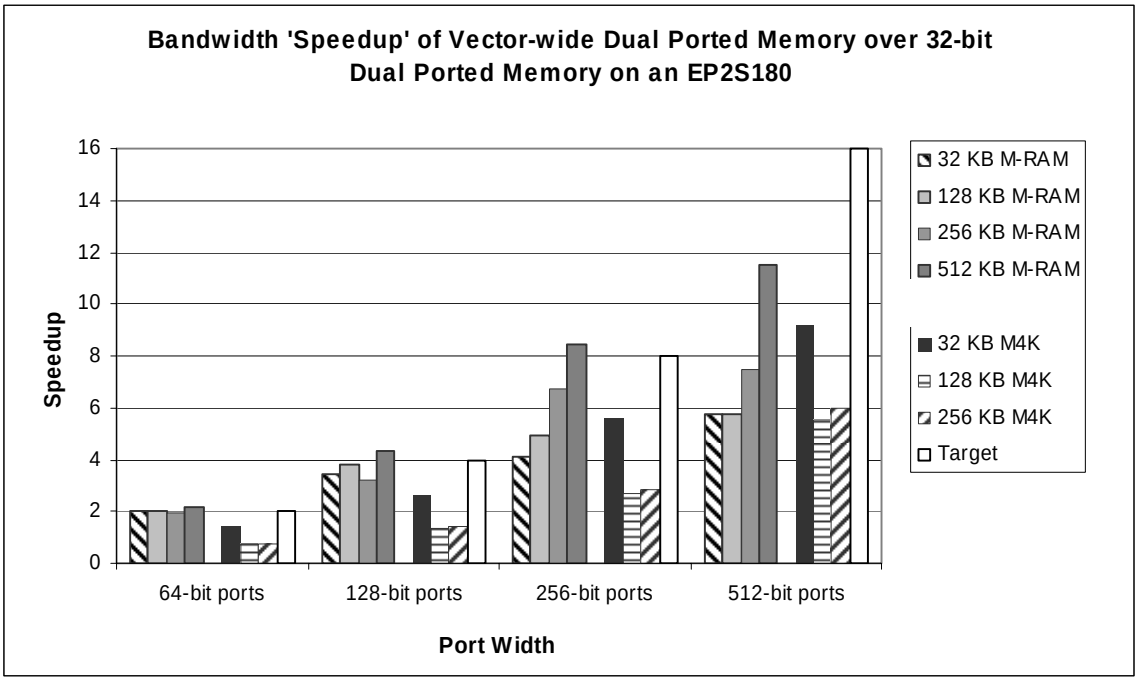


Figure 33. Bandwidth speedup of vector-wide dual ported memory over 32-bit dual ported memory on an EP2S180. The speedup of a 32-bit dual ported memory is implied to be 1x. 'Target' indicates ideal bandwidth speedup.

A shared memory with multiple-word access may support independent addressing as opposed to vector-based access (one address, many continuous data values). The independently addressed memory architecture is more complex than a vector-based architecture. The memory must be composed of multiple banks with some interleaving address strategy. The more banks a memory interleaves, the higher order decoder needed, and the more complex the routing. An independently addressed memory, however, affords great access control of the SuperCISC

hardware to the main memory. A sample shared memory architecture with an independent addressing scheme and supporting dual-port access to the VLIW and 8-word access to the SuperCISC hardware functions is shown in Figure 34.

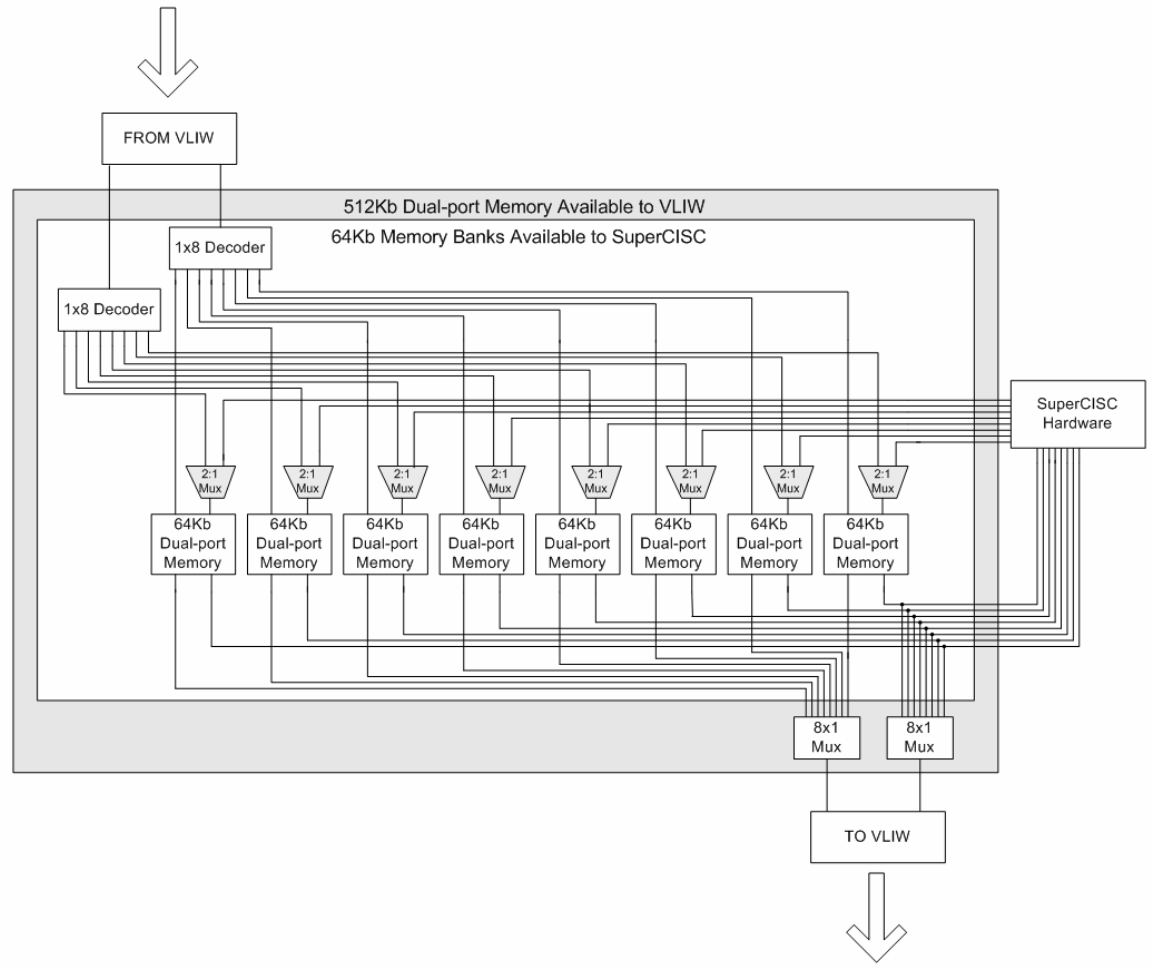


Figure 34. Sample configuration of a shared memory architecture. The VLIW accesses a 512Kb dual-ported memory. The SuperCISC hardware function accesses eight, 64Kb memory banks.

8.0 CONCLUSION

Advances in FPGA technology are making single-chip, custom processors a rewarding and increasingly feasible pursuit. The architecture proposed in this thesis is one such example - a *VLIW/SuperCISC* software processor/hardware accelerated design implemented on a Stratix II FPGA. Its performance profile and application speedups are promising enough to inspire more work in this field. As fabrication technology improves and more levels of configurable granularity emerge [7][8][19][20], experimental designs will challenge the boundaries of traditional processors.

This thesis makes several contributions in the area of VLIW processor design and hardware-accelerated architectures. Research contributions include:

1. Designing a 4-way VLIW from the ground up in VHDL hardware description language. The VLIW processor combines 4 parallel RISC processors to shared a register file, address space and instruction stream to execute up to 4 different instructions concurrently (scheduled at compile time).

2. Designing a zero-overhead implementation of a hardware/software interface. The interface consists of a shared register file and common instruction stream. The shared instruction stream allows execution of the SuperCISC hardware with zero-cycle context

switching. The SuperCISC hardware interfaces to the register file with the addition of a 2:1 multiplexer in front of each register file. The added latency is absorbed by the slack time of the register file with respect to the VLIW. Therefore, there is zero noticeable cost to the operating performance of the VLIW processor.

3. Evaluation of the scalability of a shared register file. Measuring performance decline of the register file help to make design decisions with respect to widening the VLIW processor.

4. Illustrating examples of application-specific hardware accelerants. SuperCISC hardware is an application-driven hardware accelerant that represents a kernel of software code as a densely-packed, asynchronous combinational logic circuit. SuperCISC hardware is currently designed by hand using a 4-step process that starts with application source code profiling, to hardware/software code partitioning, to data-flow graph generation, and finally translation to VHDL hardware modeling language.

5. Designing SystemC-based simulator to mimic the VLIW/hardware-accelerated hybrid processor. The SystemC VLIW processor provides data about an application's run-time profile.

6. Evaluating the costs of shared memory configurations. Two shared memory configurations were presented – an independently-address, interleaved memory architecture, and a vector-ported single-bank memory architecture.

The contributions of this work serve to further the advancement and deployment of application-driven processor design. Single-issue processors may soon approach limits of transistor size and density and will then require parallel architecture strategies and application-specific accelerants to achieve execution-time improvements. Creative strategies for processor design can merge with advances in FPGA technology to open the doors to fast and highly-feasible hardware/software co-designed architectures.

APPENDIX

A.1 SOURCE CODE FOR APPLICATION KERNEL OF ADPCM DECODER

```
1.  // KERNEL SETUP
2.  state = &decoder_state;
3.  outp2 = pcmdata_2;
4.  inp2 = (signed char *)adpcmdata;

5.  valpred = state->valprev;
6.  index = state->index;
7.  step = stepsizeTable[index];

8.  bufferstep = 0;
9.  for ( ; len > 0 ; len-- ) {

10. // BEGIN KERNEL
11. if ( bufferstep ) {
12. delta = inputbuffer & 0xf;
13. } else {
14. inputbuffer = *inp2++;
15. delta = (inputbuffer >> 4) & 0xf;
16. }
17. bufferstep = !bufferstep;
18. index += indexTable[delta];
19. if ( index < 0 ) index = 0;
20. if ( index > 88 ) index = 88;
21. sign = delta & 8;
22. delta = delta & 7;

23. vpdiff = step >> 3;
24. if ( delta & 4 ) vpdiff += step;
25. if ( delta & 2 ) vpdiff += step>>1;
26. if ( delta & 1 ) vpdiff += step>>2;

27. if ( sign )
28. valpred -= vpdiff;
29. else
30. valpred += vpdiff;

31. if ( valpred > 32767 )
32. valpred = 32767;
33. else if ( valpred < -32768 )
34. valpred = -32768;

35. step = stepsizeTable[index];
36. *outp2++ = valpred;
37. // END KERNEL
38. }

39. state->valprev = valpred;
40. state->index = index;
```

A.2 SOURCE CODE FOR APPLICATION KERNEL OF ADPCM ENCODER

```
1.  // KERNEL SETUP
2.  outp = (signed char *)adpcmdata;
3.  inp = pcmdata;
4.  valpred = state->valprev;
5.  index = state->index;
6.  step = stepsizeTable[index];
7.  bufferstep = 1;

8.  for ( ; len > 0 ; len-- ) {
9.  // BEGIN KERNEL
10. val = *inp++;
11. diff = val - valpred;
12. sign = (diff < 0) ? 8 : 0;
13. if ( sign ) diff = (-diff);
14. delta = 0;
15. vpdiff = (step >> 3);
16. if ( diff >= step ) {
17. delta = 4;
18. diff -= step;
19. vpdiff += step;
20. }
21. step >>= 1;
22. if ( diff >= step  ) {
23. delta |= 2;
24. diff -= step;
25. vpdiff += step;
26. }
27. step >>= 1;
28. if ( diff >= step ) {
29. delta |= 1;
30. vpdiff += step;
31. }
32. if ( sign )
33. valpred -= vpdiff;
34. else
35. valpred += vpdiff;

36. if ( valpred > 32767 )
37. valpred = 32767;
38. else if ( valpred < -32768 )
39. valpred = -32768;

40. delta |= sign;

41. index += indexTable[delta];
42. if ( index < 0 ) index = 0;
43. if ( index > 88 ) index = 88;
44. step = stepsizeTable[index];

45. if ( bufferstep ) {
46. outputbuffer = (delta << 4) & 0xf0;
47. } else {
48. *outp++ = (delta & 0x0f) | outputbuffer;
49. }
50. bufferstep = !bufferstep;
51. // END KERNEL
52. }
53. if ( !bufferstep )
54. *outp++ = outputbuffer;
55. state->valprev = valpred;
56. state->index = index;
```

A.3 SOURCE CODE FOR APPLICATION KERNEL OF G.721 DECODER

```
1.  an = (int)state->b[i];
2.  an = an >> 2;
3.  srn = (int)state->dq[i];
4.  anmag = (an > 0) ? an : ((-an) & 0x1FFF);
5.  anexp = quan(anmag, power2, 15) - 6;
6.  int j = 0;
7.  do {
8.  if (anmag < power2[j])
9.  break;
10. j++;
11. } while (j < 15);
12. anexp = j-6;
13. anmant = (anmag == 0) ? 32 :
14. (anexp >= 0) ? anmag >> anexp : anmag << -anexp;
15. wanexp = anexp + ((srn >> 6) & 0xF) - 13;

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

19. sezi+=(((an ^ srn) < 0) ? -retval : retval);
```

A.4 SOURCE CODE FOR APPLICATION KERNEL OF GSM DECODER

```
1.  tmp1 = rrp[i];
2.  tmp2 = v[i];
3.  tmp2 =  ( tmp1 == MIN_WORD && tmp2 == MIN_WORD
        a.  ? MAX_WORD
        b.  : 0x0FFFF & (( (long)tmp1 * (long)tmp2 + 16384) >> 15))
            ;

4.  sri  = sri - tmp2;
5.  tmp1  = ( tmp1 == MIN_WORD && sri == MIN_WORD
        a.  ? MAX_WORD
        b.  : 0x0FFFF & (( (long)tmp1 * (long)sri
                i.  + 16384) >> 15)) ;

6.  v[i+1] = v[i] + tmp1;
7.  }
8.  srp+=1;
9.  srp = vp = sri;
```

A.5 SOURCE CODE FOR APPLICATION KERNEL OF IDCT COLUMN

```
1.  if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) | (x3 = blk[8*2]) |
              i.  (x4 = blk[8*1]) | (x5 = blk[8*7]) | (x6 = blk[8*5]) |
                  (x7 = blk[8*3])))
2.  {
3.  blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=blk[8*4]=blk[8*5]=blk[8*6]=blk[8*7]
    = (blk[8*0]+32)>>6;
        a.  return(0);
4.  }

5.  x0 = (blk[8*0]<<8) + 8192;

6.  x8 = W7*(x4+x5) + 4;
7.  x4 = (x8+(W1-W7)*x4)>>3;
8.  x5 = (x8-(W1+W7)*x5)>>3;
9.  x8 = W3*(x6+x7) + 4;
10. x6 = (x8-(W3-W5)*x6)>>3;
11. x7 = (x8-(W3+W5)*x7)>>3;

12. x8 = x0 + x1;
13. x0 -= x1;
14. x1 = W6*(x3+x2) + 4;
15. x2 = (x1-(W2+W6)*x2)>>3;
16. x3 = (x1+(W2-W6)*x3)>>3;
17. x1 = x4 + x6;
18. x4 -= x6;
19. x6 = x5 + x7;
20. x5 -= x7;

21. x7 = x8 + x3;
22. x8 -= x3;
23. x3 = x0 + x2;
24. x0 -= x2;
25. x2 = (181*(x4+x5)+128)>>8;
26. x4 = (181*(x4-x5)+128)>>8;

27. temp =(x7+x1)>>14;

28. temp = (x3+x2)>>14;
29. temp = (x0+x4)>>14;
30. temp = (x8+x6)>>14;
31. temp = (x8-x6)>>14;
32. temp = (x0-x4)>>14;
33. temp = (x3-x2)>>14;
34. temp = (x7-x1)>>14;

35. blk[8*0] = iclp[1*1];
36. blk[8*1] = iclp[2*i];
37. blk[8*2] = iclp[3*i];
38. blk[8*3] = iclp[4*i];
39. blk[8*4] = iclp[5*i];
40. blk[8*5] = iclp[6*1];
41. blk[8*6] = iclp[7*i];
42. blk[8*7] = iclp[8*i];
```

A.6 SOURCE CODE FOR APPLICATION KERNEL OF IDCT ROW

```
1.  if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x3 = blk[2]) |
2.  (x4 = blk[1]) | (x5 = blk[7]) | (x6 = blk[5]) | (x7 = blk[3])))
3.  {
4.  blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=blk[6]=blk[7]=blk[0]<<3;
5.  return (0);
6.  }

7.  x0 = (blk[0]<<11) + 128; /* for proper rounding in the fourth stage
    */

8.  x8 = W7*(x4+x5);
9.  x4 = x8 + (W1-W7)*x4;
10. x5 = x8 - (W1+W7)*x5;
11. x8 = W3*(x6+x7);
12. x6 = x8 - (W3-W5)*x6;
13. x7 = x8 - (W3+W5)*x7;

14. x8 = x0 + x1;
15. x0 -= x1;
16. x1 = W6*(x3+x2);
17. x2 = x1 - (W2+W6)*x2;
18. x3 = x1 + (W2-W6)*x3;
19. x1 = x4 + x6;
20. x4 -= x6;
21. x6 = x5 + x7;
22. x5 -= x7;

23. x7 = x8 + x3;
24. x8 -= x3;
25. x3 = x0 + x2;
26. x0 -= x2;
27. x2 = (181*(x4+x5)+128)>>8;
28. x4 = (181*(x4-x5)+128)>>8;

29. blk[0] = (x7+x1)>>8;
30. blk[1] = (x3+x2)>>8;
31. blk[2] = (x0+x4)>>8;
32. blk[3] = (x8+x6)>>8;
33. blk[4] = (x8-x6)>>8;
34. blk[5] = (x0-x4)>>8;
35. blk[6] = (x3-x2)>>8;
36. blk[7] = (x7-x1)>>8;
```

A.7 SYSTEMC VLIW SOURCE FILE: MAIN.CPP

```
#define SC_USER_DEFINED_MAX_NUMBER_OF_PROCESSES
#define SC_VC6_MAX_NUMBER_OF_PROCESSES 80
#include "systemc.h"

//#include <climits>
//#include <cstdlib>
//#include <time.h>
//#include <sys/times.h>
#include <fstream>
#include "reg32.h"
#include "stimulus.h"
#include "display.h"
#include "alu.h"
#include "regfile.h"
#include "ram.h"
#include "decoder.h"
#include "mux_2to1.h"
#include "byte_sel.h"
#include "pc.h"
#include "I_cache.h"
#include <fstream>
#include "directives.h"
#include "mem_addr_conv.h"



int sc_main(int ac, char *av[])
{


  int flen;
  cout << endl;
  cout << "Enter hex file length (in lines): ";
  cin >> flen;
  sc_signal<int>   filelen;
  filelen.write(flen);

  // GLOBALS
  // clock(name, period, duty cycle, 1st edge, 1st value)
  sc_clock         clock("clock",0.25,0.5,0,true);
  sc_signal<bool>  reset;
  sc_signal<bool>  en;
  sc_signal<int>   regin;
  sc_signal<int>   regout;


  // ALU
  sc_signal<int>   op_a;
```

```
sc_signal<int>   op_b;
sc_signal<int>   alu_out;
sc_signal<int>   opcode;
sc_signal<bool>signed_op1;
sc_signal<bool>signed_op2;
sc_signal<int>   op_a1;
sc_signal<int>   op_b1;
sc_signal<int>   alu_out1;
sc_signal<int>   opcode1;
sc_signal<bool>signed_op11;
sc_signal<bool>signed_op21;
sc_signal<int>   op_a2;
sc_signal<int>   op_b2;
sc_signal<int>   alu_out2;
sc_signal<int>   opcode2;
sc_signal<bool>signed_op12;
sc_signal<bool>signed_op22;
sc_signal<int>   op_a3;
sc_signal<int>   op_b3;
sc_signal<int>   alu_out3;
sc_signal<int>   opcode3;
sc_signal<bool>signed_op13;
sc_signal<bool>signed_op23;

// REGFILE
sc_signal<bool>wr_reg;
sc_signal<bool>immed;
sc_signal<int>   s1;
sc_signal<int>   s2;
sc_signal<int>   rc_0;
sc_signal<int>   immed32;
sc_signal<int>   reg_din;
sc_signal<int>   dest;
sc_signal<bool>wr_reg1;
sc_signal<bool>immed1;
sc_signal<int>   s3;
sc_signal<int>   s4;
sc_signal<int>   rc_1;
sc_signal<int>   immed321;
sc_signal<int>   reg_din1;
sc_signal<int>   dest1;
sc_signal<bool>wr_reg2;
sc_signal<bool>immed2;
sc_signal<int>   s5;
sc_signal<int>   s6;
sc_signal<int>   rc_2;
sc_signal<int>   immed322;
sc_signal<int>   reg_din2;
sc_signal<int>   dest2;
sc_signal<bool>wr_reg3;
sc_signal<bool>immed3;
sc_signal<int>   s7;
sc_signal<int>   s8;
sc_signal<int>   rc_3;
sc_signal<int>   immed323;
```

```
sc_signal<int>   reg_din3;
sc_signal<int>   dest3;


// RAM
sc_signal<bool>wr_mem;
sc_signal<int>   bytelane;
sc_signal<int>   reg_to_mem;
sc_signal<int>   mem_dout;
sc_signal<int>   mem_din;
sc_signal<int>   mem_addr;
sc_signal<bool>wr_mem1;
sc_signal<int>   bytelane1;
sc_signal<int>   reg_to_mem1;
sc_signal<int>   mem_dout1;
sc_signal<int>   mem_din1;
sc_signal<int>   mem_addr1;

// DECODER
sc_signal<long>inst;
sc_signal<long>inst1;
sc_signal<long>inst2;
sc_signal<long>inst3;
sc_signal<int>   load_bytelane;
sc_signal<int>   store_bytelane;
sc_signal<int>   load_bytelane1;
sc_signal<int>   store_bytelane1;
sc_signal<int>   br_addr;
sc_signal<bool>branch;
sc_signal<bool>branch_check;
sc_signal<bool>jump;
sc_signal<bool>fn_call;
sc_signal<bool>fn_callr;
sc_signal<bool>fn_return;

// PROGRAM COUNTER
sc_signal<int>   addr;
sc_signal<bool>br_taken;
sc_signal<bool>fn_taken;
sc_signal<bool>stall;
sc_signal<bool>stall1;
sc_signal<bool>stall2;
sc_signal<bool>stall3;

// BTYE SEL
sc_signal<int>   mem_to_reg;
sc_signal<int>   mem_to_reg1;

// MUX
sc_signal<bool>alu_or_mem;
sc_signal<bool>alu_or_mem1;

// REG32
sc_signal<int>   alu_to_reg;
sc_signal<int>   alu_to_reg1;
```

```
sc_signal<int>   alu_to_reg2;
sc_signal<int>   alu_to_reg3;
sc_signal<int>   reg_to_memin;
sc_signal<int>   reg_to_memin1;

sc_signal<int>   prev_addr;


//// UNUSED SIGNALS///////
sc_signal<int>   br_addr1;
sc_signal<bool>branch1;
sc_signal<bool>branch_check1;
sc_signal<bool>br_taken1;
sc_signal<bool>jump1;
sc_signal<bool>fn_call1;
sc_signal<bool>fn_callr1;
sc_signal<bool>fn_return1;

sc_signal<int>   br_addr2;
sc_signal<bool>branch2;
sc_signal<bool>branch_check2;
sc_signal<bool>br_taken2;
sc_signal<bool>jump2;
sc_signal<bool>fn_call2;
sc_signal<bool>fn_callr2;
sc_signal<bool>fn_return2;
sc_signal<bool>alu_or_mem2;
sc_signal<int>   load_bytelane2;
sc_signal<int>   store_bytelane2;
sc_signal<bool>wr_mem2;
sc_signal<int>   bytelane2;

sc_signal<int>   br_addr3;
sc_signal<bool>branch3;
sc_signal<bool>branch_check3;
sc_signal<bool>br_taken3;
sc_signal<bool>jump3;
sc_signal<bool>fn_call3;
sc_signal<bool>fn_callr3;
sc_signal<bool>fn_return3;
sc_signal<bool>alu_or_mem3;
sc_signal<int>   load_bytelane3;
sc_signal<int>   store_bytelane3;
sc_signal<bool>wr_mem3;
sc_signal<int>   bytelane3;


// Regval to store to mem (0-1)
reg32 store_reg("HOLD_REGVAL");
store_reg.clock(clock.signal());
store_reg.reset(reset);
store_reg.regin(reg_to_memin);
store_reg.regout(reg_to_mem);

reg32 store_reg1("HOLD_REGVAL1");
```

```
store_reg1.clock(clock.signal());
store_reg1.reset(reset);
store_reg1.regin(reg_to_memin1);
store_reg1.regout(reg_to_mem1);


// ALU Val to hold for reg (0-3)
reg32 alu_reg("HOLD_ALU_OUT");
alu_reg.clock(clock.signal());
alu_reg.reset(reset);
alu_reg.regin(alu_out);
alu_reg.regout(alu_to_reg);

reg32 alu_reg1("HOLD_ALUVAL1");
alu_reg1.clock(clock.signal());
alu_reg1.reset(reset);
alu_reg1.regin(alu_out1);
alu_reg1.regout(alu_to_reg1);

reg32 alu_reg2("HOLD_ALUVAL2");
alu_reg2.clock(clock.signal());
alu_reg2.reset(reset);
alu_reg2.regin(alu_out2);
alu_reg2.regout(reg_din2);

reg32 alu_reg3("HOLD_ALUVAL3");
alu_reg3.clock(clock.signal());
alu_reg3.reset(reset);
alu_reg3.regin(alu_out3);
alu_reg3.regout(reg_din3);

reg32 addr_reg("HOLD_ADDR");
addr_reg.clock(clock.signal());
addr_reg.reset(reset);
addr_reg.regin(addr);
addr_reg.regout(prev_addr);

// Stimulus (0)
stimulus stim("stimulus_block");
stim.reset(reset);
stim.clock(clock.signal());

// Instruction Cache (0)
I_cache ic("INSTRUCTION_CACHE");
ic.clock(clock.signal());
ic.reset(reset);
ic.addr(addr);
ic.inst(inst);
ic.inst1(inst1);
ic.inst2(inst2);
ic.inst3(inst3);


// Display (0-3 for some)
display disp("display");
```

```
disp.clock(clock.signal());
disp.reset(reset);
disp.flen(filelen);
disp.op_a(op_a);
disp.op_b(op_b);
disp.alu_out(alu_out);
disp.opcode(opcode);
disp.signed_op1(signed_op1);
disp.signed_op2(signed_op2);
disp.wr_reg(wr_reg);
disp.immed(immed);
disp.s1(s1);
disp.s2(s2);
disp.rc_0(rc_0);
disp.immed32(immed32);
disp.reg_din(reg_din);
disp.dest(dest);
disp.wr_mem(wr_mem);
disp.mem_din(mem_din);
disp.mem_dout(mem_dout);
disp.inst(inst);
disp.addr(addr);
//disp.mem_addr(mem_addr);
disp.mem_addr(alu_out);
disp.br_addr(br_addr);
disp.branch(branch);
disp.branch_check(branch_check);
disp.br_taken(br_taken);
disp.jump(jump);
disp.fn_call(fn_call);
disp.fn_callr(fn_callr);
disp.fn_return(fn_return);
disp.stall(stall);
disp.alu_or_mem(alu_or_mem);
/// DISP Sigs PE1
disp.op_a1(op_a1);
disp.op_b1(op_b1);
disp.alu_out1(alu_out1);
disp.opcode1(opcode1);
disp.signed_op11(signed_op11);
disp.signed_op21(signed_op21);
disp.wr_reg1(wr_reg1);
disp.immed1(immed1);
disp.s3(s3);
disp.s4(s4);
disp.rc_1(rc_1);
disp.immed321(immed321);
disp.reg_din1(reg_din1);
disp.dest1(dest1);
disp.wr_mem1(wr_mem1);
disp.mem_din1(mem_din1);
disp.mem_dout1(mem_dout1);
disp.inst1(inst1);
//disp.mem_addr1(mem_addr1);
disp.mem_addr1(alu_out1);
```

```
disp.stall1(stall1);
disp.alu_or_mem1(alu_or_mem1);
// disp SIGS PE2
disp.op_a2(op_a2);
disp.op_b2(op_b2);
disp.alu_out2(alu_out2);
disp.opcode2(opcode2);
disp.signed_op12(signed_op12);
disp.signed_op22(signed_op22);
disp.wr_reg2(wr_reg2);
disp.immed2(immed2);
disp.s5(s5);
disp.s6(s6);
disp.rc_2(rc_2);
disp.immed322(immed322);
disp.reg_din2(reg_din2);
disp.dest2(dest2);
disp.inst2(inst2);
disp.stall2(stall2);
// DISP Sigs PE3
disp.op_a3(op_a3);
disp.op_b3(op_b3);
disp.alu_out3(alu_out3);
disp.opcode3(opcode3);
disp.signed_op13(signed_op13);
disp.signed_op23(signed_op23);
disp.wr_reg3(wr_reg3);
disp.immed3(immed3);
disp.s7(s7);
disp.s8(s8);
disp.rc_3(rc_3);
disp.immed323(immed323);
disp.reg_din3(reg_din3);
disp.dest3(dest3);
disp.inst3(inst3);
disp.stall3(stall3);

// TEMP DISPLAY SIGNALS
disp.mem_to_reg(mem_to_reg);
disp.alu_to_reg(alu_to_reg);
disp.load_bytelane(load_bytelane);


// ALU (0-3)
alu pe("alu");
pe.reset(reset);
pe.clock(clock.signal());
pe.op_a(op_a);
pe.op_b(op_b);
pe.alu_out(alu_out);
pe.opcode(opcode);
pe.signed_op1(signed_op1);
pe.signed_op2(signed_op2);

alu pe1("alu1");
```

```
pe1.reset(reset);
pe1.clock(clock.signal());
pe1.op_a(op_a1);
pe1.op_b(op_b1);
pe1.alu_out(alu_out1);
pe1.opcode(opcode1);
pe1.signed_op1(signed_op11);
pe1.signed_op2(signed_op21);

alu pe2("alu2");
pe2.reset(reset);
pe2.clock(clock.signal());
pe2.op_a(op_a2);
pe2.op_b(op_b2);
pe2.alu_out(alu_out2);
pe2.opcode(opcode2);
pe2.signed_op1(signed_op12);
pe2.signed_op2(signed_op22);

alu pe3("alu3");
pe3.reset(reset);
pe3.clock(clock.signal());
pe3.op_a(op_a3);
pe3.op_b(op_b3);
pe3.alu_out(alu_out3);
pe3.opcode(opcode3);
pe3.signed_op1(signed_op13);
pe3.signed_op2(signed_op23);

// Select to WB from ALU or MEM (0-1)
mux_2to1 mux_reg_din("CHOOSE_REGIN");
mux_reg_din.d0(alu_to_reg);
mux_reg_din.d1(mem_to_reg);
mux_reg_din.q0(reg_din);
mux_reg_din.sel(alu_or_mem);

mux_2to1 mux_reg_din1("CHOOSE_REGIN1");
mux_reg_din1.d0(alu_to_reg1);
mux_reg_din1.d1(mem_to_reg1);
mux_reg_din1.q0(reg_din1);
mux_reg_din1.sel(alu_or_mem1);

// Select Byte to Load (0-1)
byte_sel load_din("LOADVAL_BYTELANE");
load_din.byte_lane(load_bytelane);
load_din.din(mem_dout);
load_din.dout(mem_to_reg);

byte_sel load_din1("LOADVAL_BYTELANE1");
load_din1.byte_lane(load_bytelane1);
load_din1.din(mem_dout1);
load_din1.dout(mem_to_reg1);


// Select Byte to Store (0-1)
```

```
byte_sel store_din("STOREVAL_BYTELANE");
store_din.byte_lane(store_bytelane);
store_din.din(reg_to_mem);
store_din.dout(mem_din);

byte_sel store_din1("STOREVAL_BYTELANE1");
store_din1.byte_lane(store_bytelane1);
store_din1.din(reg_to_mem1);
store_din1.dout(mem_din1);

// Register file ports (0)
regfile rf("reg_file");
rf.wr_reg(wr_reg);
rf.immed(immed);
rf.immed32(immed32);
rf.s1(s1);
rf.s2(s2);
rf.dest(dest);
rf.op_a(op_a);
rf.op_b(op_b);
rf.to_mem(reg_to_memin);
rf.reg_din(reg_din);
rf.wr_reg1(wr_reg1);
rf.immed1(immed1);
rf.immed321(immed321);
rf.s3(s3);
rf.s4(s4);
rf.dest1(dest1);
rf.op_a1(op_a1);
rf.op_b1(op_b1);
rf.to_mem1(reg_to_memin1);
rf.reg_din1(reg_din1);
rf.wr_reg2(wr_reg2);
rf.immed2(immed2);
rf.immed322(immed322);
rf.s5(s5);
rf.s6(s6);
rf.dest2(dest2);
rf.op_a2(op_a2);
rf.op_b2(op_b2);
rf.reg_din2(reg_din2);
rf.wr_reg3(wr_reg3);
rf.immed3(immed3);
rf.immed323(immed323);
rf.s7(s7);
rf.s8(s8);
rf.dest3(dest3);
rf.op_a3(op_a3);
rf.op_b3(op_b3);
rf.reg_din3(reg_din3);
rf.reset(reset);
rf.clock(clock.signal());

// RAM 2 ports
ram mem("RAM");
```

```
mem.reset(reset);
mem.clock(clock.signal());
mem.wr_mem(wr_mem);
mem.bytelane(bytelane);
//mem.addr(mem_addr);
mem.addr(alu_out);
mem.mem_din(mem_din);
mem.mem_dout(mem_dout);
mem.wr_mem1(wr_mem1);
mem.bytelane1(bytelane1);
//mem.addr1(mem_addr1);
mem.addr1(alu_out1);
mem.mem_din1(mem_din1);
mem.mem_dout1(mem_dout1);

// CONV MEM_ADDR (0-1_
mem_addr_conv addr_conv("Mem_addr_conv");
addr_conv.mem_addr_in(alu_out);
addr_conv.mem_addr_out(mem_addr);

mem_addr_conv addr_conv1("Mem_addr_conv1");
addr_conv1.mem_addr_in(alu_out1);
addr_conv1.mem_addr_out(mem_addr1);

// DECODER (0-3 but different)
decoder dec("decoder");
dec.inst(inst);
dec.prev_addr(prev_addr);
dec.reset(reset);
dec.clock(clock.signal());
dec.s1(s1);
dec.s2(s2);
dec.rc_(rc_0);
dec.dest(dest);
dec.immed32(immed32);
dec.immed(immed);
dec.signed_op1(signed_op1);
dec.signed_op2(signed_op2);
dec.br_addr(br_addr);
dec.branch(branch);
dec.branch_check(branch_check);
dec.br_taken(br_taken);
dec.fn_taken(fn_taken);
dec.jump(jump);
dec.fn_call(fn_call);
dec.fn_callr(fn_callr);
dec.fn_return(fn_return);
dec.opcode(opcode);
dec.alu_or_mem(alu_or_mem);
dec.load_bytelane(load_bytelane);
dec.store_bytelane(store_bytelane);
dec.bytelane(bytelane);
dec.wr_mem(wr_mem);
dec.wr_reg(wr_reg);
dec.stall(stall);
```

```
decoder dec1("decoder1");
dec1.inst(inst1);
dec1.prev_addr(prev_addr);
dec1.reset(reset);
dec1.clock(clock.signal());
dec1.s1(s3);
dec1.s2(s4);
dec1.rc_(rc_1);
dec1.dest(dest1);
dec1.immed32(immed321);
dec1.immed(immed1);
dec1.signed_op1(signed_op11);
dec1.signed_op2(signed_op21);
dec1.opcode(opcode1);
dec1.alu_or_mem(alu_or_mem1);
dec1.load_bytelane(load_bytelane1);
dec1.store_bytelane(store_bytelane1);
dec1.bytelane(bytelane1);
dec1.wr_mem(wr_mem1);
dec1.wr_reg(wr_reg1);
dec1.stall(stall1);
dec1.br_addr(br_addr1);
dec1.branch(branch1);
dec1.branch_check(branch_check1);
dec1.br_taken(br_taken);
dec1.fn_taken(fn_taken);
dec1.jump(jump1);
dec1.fn_call(fn_call1);
dec1.fn_callr(fn_callr1);
dec1.fn_return(fn_return1);


decoder dec2("decoder2");
dec2.inst(inst2);
dec2.prev_addr(prev_addr);
dec2.reset(reset);
dec2.clock(clock.signal());
dec2.s1(s5);
dec2.s2(s6);
dec2.rc_(rc_2);
dec2.dest(dest2);
dec2.immed32(immed322);
dec2.immed(immed2);
dec2.signed_op1(signed_op12);
dec2.signed_op2(signed_op22);
dec2.opcode(opcode2);
dec2.wr_reg(wr_reg2);
dec2.stall(stall2);
dec2.br_addr(br_addr2);
dec2.branch(branch2);
dec2.branch_check(branch_check2);
dec2.br_taken(br_taken);
dec2.fn_taken(fn_taken);
dec2.jump(jump2);
```

```
dec2.fn_call(fn_call2);
dec2.fn_callr(fn_callr2);
dec2.fn_return(fn_return2);
dec2.alu_or_mem(alu_or_mem2);
dec2.load_bytelane(load_bytelane2);
dec2.store_bytelane(store_bytelane2);
dec2.bytelane(bytelane2);
dec2.wr_mem(wr_mem2);

decoder dec3("decoder3");
dec3.inst(inst3);
dec3.prev_addr(prev_addr);
dec3.reset(reset);
dec3.clock(clock.signal());
dec3.s1(s7);
dec3.s2(s8);
dec3.rc_(rc_3);
dec3.dest(dest3);
dec3.immed32(immed323);
dec3.immed(immed3);
dec3.signed_op1(signed_op13);
dec3.signed_op2(signed_op23);
dec3.opcode(opcode3);
dec3.wr_reg(wr_reg3);
dec3.stall(stall3);
dec3.br_addr(br_addr3);
dec3.branch(branch3);
dec3.branch_check(branch_check3);
dec3.br_taken(br_taken);
dec3.fn_taken(fn_taken);
dec3.jump(jump3);
dec3.fn_call(fn_call3);
dec3.fn_callr(fn_callr3);
dec3.fn_return(fn_return3);
dec3.alu_or_mem(alu_or_mem3);
dec3.load_bytelane(load_bytelane3);
dec3.store_bytelane(store_bytelane3);
dec3.bytelane(bytelane3);
dec3.wr_mem(wr_mem3);


// Program Counter (0)
pc ctr("PROGRAM_CTR");
ctr.clock(clock.signal());
ctr.reset(reset);
ctr.addr(addr);
ctr.branch_check(branch_check);
ctr.fn_callr(fn_callr);
ctr.fn_call(fn_call);
ctr.jump(jump);
ctr.branch(branch);
ctr.br_taken(br_taken);
ctr.fn_taken(fn_taken);
ctr.fn_return(fn_return);
ctr.stall(stall);
```

```
  ctr.stall1(stall1);
  ctr.stall2(stall2);
  ctr.stall3(stall3);
  ctr.alu_out(alu_out);
  ctr.immed_addr(immed32);
  ctr.fn_addr(op_a);
  ctr.br_addr(br_addr);



  //cout << "Time for simulation = " << endl;

  sc_start(clock, 2048);
  //sc_start(clock, 100);
  return 0;             /* this is necessary */
}
```

A.8 SYSTEMC VLIW SOURCE FILE: ALU.CPP

```
#include <systemc.h>
#include "alu.h"
#include "directives.h"

void alu::entry() {

        while(1)
        {

        func = opcode.read();
        a = op_a.read();
        b = op_b.read();
        c = 0;
        cx = 0;
        ua = op_a.read();
        ub = op_b.read();
        uc = 0;
        ucx = 0;
        sign1 = signed_op1.read();
        sign2 = signed_op2.read();

        switch (func)
        {

        case ADD_I:
                c = a + b;
                break;

        case SUB_I:
                c = a - b;
                break;

        case MULT_I:
                c = a * b;
                break;

        case MULTX_I:

                if (sign1 == 1 && sign2 ==1)
                {
                        ucx = ua * ub;
                        c = ucx >> 32;
                }
                else if (sign2 == 1)
                {
                        cx = a * ub;
                        c = cx >> 32;
                }
```

```
        else
        {
                cx = a * b;
                c = cx >> 32;
        }
        break;

case DIV_I:

        if (sign1 == 1)
        {
                uc = ua / ub;
        }
        else
        {
                 c = a / b;
        }
        break;

case AND_I:
        c = a & b;
        break;

case OR_I:
        c = a | b;
        break;

case XOR_I:
        c = a ^ b;
        break;

case NOR_I:
        c = ~(a | b);
        break;

case SLL_I:
        c = a << b;
        break;

case SRL_I:
        c = a >> b;
        break;

case SLA_I:
        c = a << b;
        break;

case SRA_I:
        c = a >> b;
        break;

case STOR_I:
        c = a + b;
        break;
```

```
case STORB_I:
        c = a + b;
        break;

case STORH_I:
        c = a + b;
        break;

case STORW_I:
        c = a + b;
        break;

case LOAD_I:
        c = a + b;
        break;

case LOADB_I:
        c = a + b;
        break;

case LOADH_I:
        c = a + b;
        break;

case LOADW_I:
        c = a + b;
        break;

case CMPGTE_I:
        if (sign1 == 1)
        {
                (ua >= ub ? c=1 : c=0);
        }
        else
        {
                        (a >= b ? c=1 : c=0);
        }
        break;

case CMPGT_I:
        if (sign1 == 1)
        {
                (ua > ub ? c=1 : c=0);
        }
        else
        {
                        (a > b ? c=1 : c=0);
        }
        break;

case CMPLTE_I:
        if (sign1 == 1)
        {
                (ua <= ub ? c=1 : c=0);
        }
```

```
                else
                {
                                (a <= b ? c=1 : c=0);
                }
                break;

        case CMPLT_I:
                if (sign1 == 1)
                {
                        (ua < ub ? c=1 : c=0);
                }
                else
                {
                                (a < b ? c=1 : c=0);
                }
                break;

        case CMPEQ_I:
                if (sign1 == 1)
                {
                        (ua == ub ? c=1 : c=0);
                }
                else
                {
                                (a == b ? c=1 : c=0);
                }
                break;

        case CMPNE_I:
                if (sign1 == 1)
                {
                        (ua != ub ? c=1 : c=0);
                }
                else
                {
                                (a != b ? c=1 : c=0);
                }
                break;

        case NOOP_I:

                break;

        default:
                        c = 0;
                        break;

        }
        alu_out.write((int) c);

        wait();
        }


}
```

A.9 SYSTEMC VLIW SOURCE FILE: DECODER.CPP

```
#include <systemc.h>
#include "decoder.h"
#include "directives.h"

void decoder::entry() {

	ins = inst.read();

	if (true)
	{
		cycle++;
		// BREAK APART INSTRUCTION
		op = ins;
		ins >>= 6;
		immed5 = ins;
		immed26 = ins;
		immed16 = ins;
		tmpimmed32 = ins;
		ins >>= 5;
		arith = ins;
		ins >>= 6;
		rc = ins;
		ins >>= 5;
		rb = ins;
		ins >>= 5;
		ra = ins;
		immediate = 0;
		hazard = 0;
		br_uncond = 0;


		// INITIALIZE INTERMEDIATE SIGNALS
		t_opcode.write(NOOP_I);
		t_signed_op1.write(0);
		t_signed_op2.write(0);
		t_wr_mem1.write(0);
		t_wr_reg1.write(0);
		t_alu_or_mem1.write(0);
		t_load_bytelane1.write(0);
		t_store_bytelane1.write(0);
		t_callr.write(0);
		t_jump.write(0);
		t_branch_check1.write(0);
		t_br_addr1.write((int)immed16);
		hold_wr_mem.write(0);
		hold_wr_reg.write(0);

		// INITIALIZE OUTPUTS
		immed.write(0);
		branch.write(0);
```

```
fn_call.write(0);
fn_return.write(0);
stall.write(0);

immed32.write((int)immed16);
s1.write((int)ra);
s2.write((int)rb);
rc_.write((int)rc);
t_dest1.write((int)rc);



switch(op)
{

        case addi:
                t_opcode.write(ADD_I);
                t_dest1.write((int)rb);
                immed.write(1);
                immediate = 1;
                t_wr_reg1.write(1);
                hold_wr_reg.write(1);
                break;

        case muli:
                t_opcode.write(MULT_I);
                t_dest1.write((int)rb);
                immed.write(1);
                immediate = 1;
                t_wr_reg1.write(1);
                hold_wr_reg.write(1);
                break;

        case call:
                if (cycle > 0 )
                {
                        fn_call.write(1);
                        immed32.write((int)immed26);
                        br_uncond = 1;
                }

                break;

        case flushd:
                branch.write(1);
                immed32.write(0);
                br_uncond = 1;
                break;

        case br:
                branch.write(1);
                br_uncond = 1;
                break;

        case bge:
```

```
            t_opcode.write(CMPGTE_I);
            t_branch_check1.write(1);
            break;

case bltu:
            t_opcode.write(CMPLT_I);
            t_signed_op1.write(1);
            t_branch_check1.write(1);
            break;

case bgeu:
            t_opcode.write(CMPGTE_I);
            t_signed_op1.write(1);
            t_branch_check1.write(1);
            break;

case beq:
            t_opcode.write(CMPEQ_I);
            t_branch_check1.write(1);
            break;

case bne:
            t_opcode.write(CMPNE_I);
            t_branch_check1.write(1);
            break;

case blt:
            t_opcode.write(CMPLT_I);
            t_branch_check1.write(1);
            break;

case ldb:
            t_opcode.write(LOADB_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            t_alu_or_mem1.write(1);
            immed.write(1);
            immediate = 1;
            t_load_bytelane1.write(2);
            t_bytelane1.write(BYTE_I);
            break;

case ldw:
            t_opcode.write(LOADW_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            t_alu_or_mem1.write(1);
            immed.write(1);
            immediate = 1;
            t_bytelane1.write(WORD_I);
            break;

case ldbu:
```

```
            t_opcode.write(LOADB_I);
            t_signed_op1.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            t_alu_or_mem1.write(1);
            immed.write(1);
            immediate = 1;
            t_load_bytelane1.write(2);
            t_bytelane1.write(BYTE_I);
            break;

    case ldhu:
            t_opcode.write(LOADH_I);
            t_signed_op1.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            t_alu_or_mem1.write(1);
            immed.write(1);
            immediate = 1;
            t_load_bytelane1.write(1);
            t_bytelane1.write(HWORD_I);
            break;

    case ldh:
            t_opcode.write(LOADH_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            t_alu_or_mem1.write(1);
            immed.write(1);
            immediate = 1;
            t_load_bytelane1.write(1);
            t_bytelane1.write(HWORD_I);
            break;

    case stw:
            t_opcode.write(STORW_I);
            t_wr_mem1.write(1);
            hold_wr_mem.write(1);
            immed.write(1);
            immediate = 1;
            t_bytelane1.write(WORD_I);
            break;

    case stb:
            t_opcode.write(STORB_I);
            t_wr_mem1.write(1);
            hold_wr_mem.write(1);
            immed.write(1);
            immediate = 1;
            t_store_bytelane1.write(2);
            t_bytelane1.write(BYTE_I);
            break;
```

```
case sth:
        t_opcode.write(STORH_I);
        t_wr_mem1.write(1);
        hold_wr_mem.write(1);
        immed.write(1);
        immediate = 1;
        t_store_bytelane1.write(1);
        t_bytelane1.write(HWORD_I);
        break;

case cmpgei:
        t_opcode.write(CMPGTE_I);
        t_wr_reg1.write(1);
        hold_wr_reg.write(1);
        t_dest1.write((int)rb);
        immed.write(1);
        immediate = 1;
        break;

case cmplti:
        t_opcode.write(CMPLT_I);
        t_wr_reg1.write(1);
        hold_wr_reg.write(1);
        t_dest1.write((int)rb);
        immed.write(1);
        immediate = 1;
        break;

case cmpnei:
        t_opcode.write(CMPNE_I);
        t_wr_reg1.write(1);
        hold_wr_reg.write(1);
        t_dest1.write((int)rb);
        immed.write(1);
        immediate = 1;
        break;

case cmpgeui:
        t_opcode.write(CMPGTE_I);
        t_signed_op1.write(1);
        t_wr_reg1.write(1);
        hold_wr_reg.write(1);
        t_dest1.write((int)rb);
        immed.write(1);
        immediate = 1;
        break;

case cmpeqi:
        t_opcode.write(CMPEQ_I);
        t_wr_reg1.write(1);
        hold_wr_reg.write(1);
        t_dest1.write((int)rb);
        immed.write(1);
        immediate = 1;
```

```
            break;

case cmpltui:
            t_opcode.write(CMPLT_I);
            t_signed_op1.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            t_dest1.write((int)rb);
            immed.write(1);
            immediate = 1;
            break;

case andi:
            t_opcode.write(AND_I);
            t_dest1.write((int)rb);
            immed.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            immediate = 1;
            break;

case andhi:
            immed32.write(tmpimmed32 << 16);
            t_opcode.write(AND_I);
            t_dest1.write((int)rb);
            immed.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            immediate = 1;
            break;

case orhi:
            t_opcode.write(OR_I);
            immed32.write(tmpimmed32 << 16);
            t_dest1.write((int)rb);
            immed.write(1);
            immediate = 1;
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case ori:
            t_opcode.write(OR_I);
            t_dest1.write((int)rb);
            immed.write(1);
            immediate = 1;
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case xorhi:
            t_opcode.write(XOR_I);
            immed32.write(tmpimmed32 << 16);
            t_dest1.write((int)rb);
            immed.write(1);
```

```
		immediate = 1;
		t_wr_reg1.write(1);
		hold_wr_reg.write(1);
		break;

	case xori:
		t_opcode.write(XOR_I);
		t_dest1.write((int)rb);
		immed.write(1);
		immediate = 1;
		t_wr_reg1.write(1);
		hold_wr_reg.write(1);
		break;

	case rtype:

		switch(arith)
		{
			case add:
				t_opcode.write(ADD_I);
				t_wr_reg1.write(1);
				hold_wr_reg.write(1);
				break;

			case sub:
				t_opcode.write(SUB_I);
				t_wr_reg1.write(1);
				hold_wr_reg.write(1);
				break;

			case divu:
				t_opcode.write(DIV_I);
				t_signed_op1.write(1);
				break;

			case div:
				t_opcode.write(DIV_I);
				t_signed_op1.write(1);
				break;

			case mul:
				t_opcode.write(MULT_I);
				t_wr_reg1.write(1);
				hold_wr_reg.write(1);
				break;

			case mulxss:
				t_opcode.write(MULTX_I);
				t_wr_reg1.write(1);
				hold_wr_reg.write(1);
				break;

			case mulxsu:
				t_opcode.write(MULTX_I);
				t_signed_op2.write(1);
```

```
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case mulxuu:
            t_opcode.write(MULTX_I);
            t_signed_op1.write(1);
            t_signed_op2.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case and_r:
            t_opcode.write(AND_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case xor_r:
            t_opcode.write(XOR_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case nor_r:
            t_opcode.write(NOR_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case or_r:
            t_opcode.write(OR_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case ret:
            fn_return.write(1);
            br_uncond = 1;
            break;

case callr:
            t_callr.write(1);
            br_uncond = 1;
            break;

case flushp:
            branch.write(1);
            immed32.write(0);
            br_uncond = 1;
            break;

case jmp:
            t_jump.write(1);
            br_uncond = 1;
```

```
            break;

case cmpeq:
            t_opcode.write(CMPEQ_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case cmpne:
            t_opcode.write(CMPNE_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case cmpge:
            t_opcode.write(CMPGTE_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case cmpgeu:
            t_opcode.write(CMPGTE_I);
            t_signed_op1.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case cmplt:
            t_opcode.write(CMPLT_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case cmpltu:
            t_opcode.write(CMPLT_I);
            t_signed_op1.write(1);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case rotl:
            t_opcode.write(SLA_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            break;

case rotli:
            t_opcode.write(SLA_I);
            t_wr_reg1.write(1);
            hold_wr_reg.write(1);
            immed.write(1);
            immediate = 1;
            immed32.write((int)immed5);
            break;
```

```
		case rotr:
			t_opcode.write(SRA_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			break;

		case sll:
			t_opcode.write(SLL_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			break;

		case slli:
			t_opcode.write(SLL_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			immed.write(1);
			immediate = 1;
			immed32.write((int)immed5);
			break;

		case sra:
			t_opcode.write(SRA_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			break;

		case srai:
			t_opcode.write(SRA_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			immed.write(1);
			immediate = 1;
			immed32.write((int)immed5);
			break;

		case srl:
			t_opcode.write(SRL_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			break;

		case srli:
			t_opcode.write(SRL_I);
			t_wr_reg1.write(1);
			hold_wr_reg.write(1);
			immed.write(1);
			immediate = 1;
			immed32.write((int)immed5);
			break;

		default:
			break;

}
```

```
                        break;

                default:
                        t_opcode.write(NOOP_I);

                        break;

        } // end case
/*********************************/
        // CHECK for branching
        if (br_taken.read() ==1)
        {
                //t_wr_reg1.write(0);
                //t_wr_mem1.write(0);
        }
/*********************************/


        if(stall_cnt.read() == 0)
        {
                //CHECK FOR DATA HAZARDS
                if ((int)ra == t_dest3.read() && t_wr_reg3.read() == 1 && (int)ra != 0)
                {
                        cout <<" Data hazard on REG[" << (unsigned)ra << "] 2 stages apart, address="
<< prev_addr.read() << endl;
                        hazard = 1;
                }
                else if ((int)ra == t_dest2.read() && t_wr_reg2.read() == 1 && (int)ra != 0)
                {
                        cout <<" Data hazard on REG[" << (unsigned)ra << "] 1 stage apart, address="
<< prev_addr.read() << endl;
                        hazard = 1;

                }
                if (immediate == 0)
                {
                        if ((int)rb == t_dest3.read() && t_wr_reg3.read() == 1 && (int)rb != 0)
                        {
                                cout <<" Data hazard on REG[" << (unsigned)rb << "] 2 stages apart,
address=" << prev_addr.read() << endl;
                                hazard = 1;
                        }
                        else if ((int)rb == t_dest2.read() && t_wr_reg2.read() == 1 && (int)rb != 0)
                        {
                                cout <<" Data hazard on REG[" << (unsigned)rb << "] 1 stage apart,
address=" << prev_addr.read() << endl;
                                hazard = 1;
                        }
                }

                // CHECK FOR cond BR followed by uncond BR HAZARDS
                if (br_uncond == 1 && t_branch_check1.read() == 1)
                {
                        cout << "Branch hazard at address=" << prev_addr.read() << endl;
```

```
                        hazard = 1;
                }
                // now check the hazard signal
                if (hazard == 1)
                {
                        stall.write(1);
                        stall_cnt.write(1);
                        t_wr_reg1.write(0);
                        t_wr_mem1.write(0);
                }
        }


/*******************************/
        } // end else

/*
// NOT DEFINED
#define io
#define initd
#define custom
#define eret
#define nextpc
#define break
#define breakr
#define rdctl
#define sync 54
#define trap 45
#define wrctl 46
*/

}
// end entry()

void decoder::hold_1cycle()
{
        while(1)
        {
                // ALU SIGNALS

                opcode.write((int)t_opcode.read());
                signed_op1.write((bool)t_signed_op1);
                signed_op2.write((bool)t_signed_op2);

                t_dest2.write((int)t_dest1.read());
                t_branch_check2.write((bool)t_branch_check1.read());
                t_br_addr2.write((int)t_br_addr1.read());
                jump.write((bool)t_jump.read());
                fn_callr.write((bool)t_callr.read());
                t_alu_or_mem2.write((bool)t_alu_or_mem1.read());

                //CHECK FOR BRANCHING
                if (br_taken.read() ==1)
                {
                        //t_wr_mem2.write(0);
```

```
                        //t_wr_reg2.write(0);
                }
                else
                {
                        t_bytelane2.write((int)t_bytelane1.read());
                        t_wr_mem2.write((bool)t_wr_mem1.read());
                        t_wr_reg2.write((bool)t_wr_reg1.read());
                }
                // check the stalls
                if (stall_cnt.read() == 1)
                {
                        stall.write(1);
                        stall_cnt.write(2);
                }
                else if(stall_cnt.read() > 0)
                {
                        stall_cnt.write(0);
                        stall.write(0);
                        t_wr_mem1.write(hold_wr_mem.read());
                        t_wr_reg1.write(hold_wr_reg.read());
                }
                if (fn_taken.read() == 1)
                {
                        //cout <<" Func taken" << endl;

                                t_wr_mem1.write(0);
                                t_wr_reg1.write(0);
                }

                wait();
        }

} // end hold_1cycle()


void decoder::hold_2cycles()
{

        while(1)
        {
                branch_check.write((bool)t_branch_check2.read());
                br_addr.write((int)t_br_addr2.read());
                t_dest3.write((int)t_dest2.read());
                t_alu_or_mem3.write((bool)t_alu_or_mem2.read());

                //CHECK FOR BRANCHING
                if (br_taken.read() ==1)
                {
                        wr_mem.write(0);
                        t_wr_reg3.write(0);
                }
                else
                {
                        bytelane.write((int)t_bytelane2.read());
                        wr_mem.write((bool)t_wr_mem2.read());
```

```
                        t_wr_reg3.write((bool)t_wr_reg2.read());
                }
                wait();
        }

} // end hold_2cycles()

void decoder::hold_3cycles()
{
        while(1)
        {
                if (br_taken.read() ==1)
                {
                        wr_reg.write(0);
                }
                else
                {
                        wr_reg.write((bool)t_wr_reg3.read());
                }
                alu_or_mem.write((bool)t_alu_or_mem3.read());
                dest.write((int)t_dest3.read());
                wait();
        }

} // end hold_3cycles()
```

A.10 SYSTEMC VLIW SOURCE FILE: DIRECTIVES.H

```
#define SIZE_MEM 32768
#define MEM_ADDR_BITS 14
#define SIZE_REG 32
#define REG_ADDR_BITS 5
//#define SIZE_PROGRAM 131072
#define SIZE_PROGRAM 4096
#define SIZE_STACK 32

// ALU OPS
#define NOOP_I -1
#define ADD_I 0
#define SUB_I 1
#define  MULT_I 2
#define MULTX_I 3
#define AND_I 4
#define OR_I 5
#define XOR_I 6
#define NOR_I 7
#define SLL_I 8
#define SRL_I 9
#define SLA_I 10
#define SRA_I 11
#define STOR_I 12
#define LOAD_I 13
#define CMPGTE_I 14
#define CMPGT_I 15
#define CMPLTE_I 16
#define CMPLT_I 17
#define CMPEQ_I 18
#define CMPNE_I 19
#define DIV_I 20
#define LOADB_I 21
#define LOADH_I 22
#define LOADW_I 23
#define STORB_I 24
#define STORH_I 25
#define STORW_I 26

//MEM OPS
#define BYTE_I 0
#define HWORD_I 1
#define WORD_I 2

// Instructions(5 downto 0)
#define call 0
#define ldbu 3
#define addi 4
#define stb 5
```

```
#define br 6
#define ldb 7
#define cmpgei 8
#define ldhu 11
#define andi 12
#define sth 13
#define bge 14
#define ldh 15
#define cmplti 16
#define ori 20
#define stw 21
#define blt 22
#define ldw 23
#define cmpnei 24
#define xori 28
#define bne 30
#define cmpeqi 32
#define ldbuio 35
#define muli 36
#define stbio 37
#define beq 38
#define ldbio 39
#define cmpgeui 40
#define ldhuio 43
#define andhi 44
#define sthio 45
#define bgeu 46
#define ldhio 47
#define cmpltui 48
#define custom 50
#define orhi 52
#define stwio 53
#define bltu 54
#define ldwio 55
#define rtype 58
#define flushd 59
#define xorhi 60


// Instruction (16 downto 11) in case of rtype (0x3a)
#define add 49
#define and_r 14
#define break_r 52
#define bret 9
#define callr 29
#define cmpeq 32
#define cmpge 8
#define cmpgeu 40
#define cmplt 16
#define cmpltu 48
#define cmpne 24
#define div 37
#define divu 36
#define eret 1
#define flushp 4
```

```
#define initi 41
#define jmp 13
#define mul 39
#define mulxss 31
#define mulxsu 23
#define mulxuu 7
#define nextpc 28
#define nor_r 6
#define or_r 22
#define rdctl 38
#define ret 5
#define rotl 3
#define rotli 2
#define rotr 11
#define sll 19
#define slli 18
#define sra 59
#define srai 58
#define srl 27
#define srli 26
#define sub 57
#define sync 54
#define trap 45
#define wrctl 46
#define xor_r 30
```

A.11 SYSTEMC VLIW SOURCE FILE: ICACHE.CPP

```
#include <systemc.h>
#include "I_cache.h"

void I_cache::entry() {

        while(1)
        {


          if (reset.read() == 1)
                {
                                inst.write(icache[0]);
                                inst1.write(icache1[0]);
                                inst2.write(icache2[0]);
                                inst3.write(icache3[0]);
                }
                else
                {
                        if(addr.read() >=0)
                        {

                                inst.write(icache[(int)addr.read()]);
                                inst1.write(icache1[(int)addr.read()]);
                                inst2.write(icache2[(int)addr.read()]);
                                inst3.write(icache3[(int)addr.read()]);
                        }
                        else
                        {
                                inst.write(icache[0]);
                                inst1.write(icache1[0]);
                                inst2.write(icache2[0]);
                                inst3.write(icache3[0]);
                        }

                }
                wait();
        }


}
```

A.12 SYSTEMC VLIW SOURCE FILE: MUX_2TO1.CPP

```
#include <systemc.h>
#include "mux_2to1.h"


void mux_2to1::entry() {

	if (sel.read() ==0)
	{
		q0.write((int)d0.read());
	}
	else
	{
		q0.write((int)d1.read());
	}

}
```

A.13 SYSTEMC VLIW SOURCE FILE: PC.CPP

```
#include <systemc.h>
#include "pc.h"
#include "directives.h"

void pc::entry() {
	while(1)
	{

	if (reset.read() ==1 )
	{
		stackptr = 0;
		immedaddr = 0;
		iptr = -1;
		offset = 0;
		fnaddr = 0;
		br_taken.write(0);

		for (int i=0; i < SIZE_STACK; i++)
		{
			stack[i] = 0;
		}
	}
	// Check branches before incrementing
	else
	{
		iptr = inst_ptr.read();
		offset = (int)br_addr.read();
		immedaddr = (int)immed_addr.read();
		fnaddr = (int)fn_addr.read();
		stackptr = (int)stack_ptr.read();
		br_taken.write(0);


		if (branch_check.read() == 1 && (int)alu_out.read() > 0)
		{
			//if (offset < 0)
			//{
				offset -=1;
			//}
			iptr += (offset);
			//iptr += (1 + offset);
			br_taken.write(1);
		}
		else if (branch.read() ==1)
		{
			iptr += (1 + immedaddr);
			fn_taken.write(1);
```

```
		}
		else if (fn_callr.read() == 1)
		{
			stack[stackptr] = iptr;
			stackptr++;
			iptr = fnaddr;
			br_taken.write(1);
			fn_taken.write(1);
		}
		else if (fn_call.read() == 1)
		{
			stack[stackptr] = iptr;
			stackptr++;
			iptr = immedaddr;
			br_taken.write(1);
			fn_taken.write(1);
		}
		else if (jump.read() ==1)
		{
			iptr = fnaddr;
			br_taken.write(1);
			fn_taken.write(1);
		}
		else if (fn_return.read() ==1)
		{
			if (stackptr != 0)
			{
				stackptr--;
			}
			if (stackptr >= 0)
			{
				iptr = stack[stackptr];
			}
			fn_taken.write(1);
		}
		else if(stall.read() == 1 || stall1.read() == 1 || stall2.read() == 1 || stall3.read() == 1)
		{
			//do nothing
		}
		else
		{
			if (cycle != 0)
			{
				iptr++;
			}
		}
		cycle++;
	}
	inst_ptr.write((int)iptr);
	addr.write((int)iptr);
	stack_ptr.write((int)stackptr);
	wait();
	}
}
```

A.14 SYSTEMC VLIW SOURCE FILE: RAM.CPP

```
#include <systemc.h>
#include "ram.h"
#include "directives.h"

void ram::entry() {


        while(1)
        {

         // Reset to zero
         if (reset.read() == 1)
         {
                /* for (int i=0; i<SIZE_MEM; i++)
                 {
                        mem[i] = 0;
                 }*/
         }
         else
         {
                /// PORT 0
                 en = wr_mem.read();

                 if (en == 1)
                 {
                        tmpaddr = addr.read();
                        data = mem_din.read();
                        if(bytelane.read() == BYTE_I)
                        {
                                byte0 = data;
                                mem[tmpaddr] = byte0;
                        }
                        else if (bytelane.read() == HWORD_I)
                        {
                                 byte0 = data;
                                 data >>= 8;
                                 byte1 = data;
                                 mem[tmpaddr] = byte1;
                                 mem[tmpaddr+1] = byte0;
                        }
                        else
                        {
                                 byte0 = data;
                                 data >>= 8;
                                 byte1 = data;
                                 data >>= 8;
                                 byte2 = data;
                                 data >>= 8;
                                 byte3 = data;
```

```
                        mem[tmpaddr] = byte3;
                        mem[tmpaddr+1] = byte2;
                        mem[tmpaddr+2] = byte1;
                        mem[tmpaddr+4] = byte0;
                }
}
tmpaddr = addr.read();
data = 0;
if (bytelane.read() == BYTE_I)
{
        data = mem[tmpaddr];
}
else if (bytelane.read() == HWORD_I)
{
        data = mem[tmpaddr];
        data <<= 8;
        data += mem[tmpaddr+1];
}
else
{
        data = mem[tmpaddr];
        data <<= 8;
        data += mem[tmpaddr+1];
        data <<= 8;
        data += mem[tmpaddr+2];
        data <<= 8;
        data += mem[tmpaddr+3];
}
mem_dout.write(data);

// PORT 1
en = wr_mem1.read();

if (en == 1)
{
        tmpaddr = addr1.read();
        data = mem_din1.read();
        if(bytelane1.read() == BYTE_I)
        {
                byte0 = data;
                mem[tmpaddr] = byte0;
        }
        else if (bytelane1.read() == HWORD_I)
        {
                byte0 = data;
                data >>= 8;
                byte1 = data;
                mem[tmpaddr] = byte1;
                mem[tmpaddr+1] = byte0;
        }
        else
        {
                byte0 = data;
                data >>= 8;
                byte1 = data;
```

```
                                        data >>= 8;
                                        byte2 = data;
                                        data >>= 8;
                                        byte3 = data;
                                        mem[tmpaddr] = byte3;
                                        mem[tmpaddr+1] = byte2;
                                        mem[tmpaddr+2] = byte1;
                                        mem[tmpaddr+4] = byte0;
                                }
                        }
                        tmpaddr = addr1.read();
                        mem_dout1.write(mem[tmpaddr]);

                }
                // all cycles
                wait();
                  }


}
```

A.15 SYSTEMC VLIW SOURCE FILE: REGFILE.CPP

```
#include <systemc.h>
#include "regfile.h"
#include "directives.h"

void regfile::entry() {


        while(1)
        {

         // Reset to zero
         if (reset.read() == 1)
         {
                for (int i=0; i<SIZE_REG; i++)
                {
                        reg[i] = 0;
                }
         }
         else
         {

                // WR_REG 0
                en = wr_reg.read();
                if (en == 1)
                {
                       addr = dest.read();
                        if ((int)addr != 0)
                        {
                                reg[addr] = reg_din.read();
                        }
                }
                // WR_REG 1
                en = wr_reg1.read();
                if (en == 1)
                {
                        addr = dest1.read();
                        if ((int)addr != 0)
                        {
                                reg[addr] = reg_din1.read();
                        }
                }
                // WR_REG 2
                en = wr_reg2.read();
                if (en == 1)
                {
                        addr = dest2.read();
                        if ((int)addr != 0)
                        {
                                reg[addr] = reg_din2.read();
```

```
            }
}
// WR_REG 3
en = wr_reg3.read();
if (en == 1)
{
            addr = dest3.read();
            if ((int)addr != 0)
            {
                        reg[addr] = reg_din3.read();
            }
}

/// READ REG PE0 ////////
// OPERAND B
if (immed.read() == 1)
{
            op_b.write((int)immed32);
}
else
{
           addr = s2.read();
           op_b.write(reg[addr]);
}
// OPERAND A
addr = s1.read();
op_a.write(reg[addr]);
// TO MEM
addr = s2.read();
to_mem.write(reg[addr]);

/// READ REG PE1 //////////
 // OPERAND B
 if (immed1.read() == 1)
 {
            op_b1.write((int)immed321);
 }
 else
 {
           addr = s4.read();
           op_b1.write(reg[addr]);
 }
// OPERAND A
addr = s3.read();
op_a1.write(reg[addr]);
// TO MEM
addr = s3.read();
to_mem1.write(reg[addr]);


/// READ REG PE2 //////////
 // OPERAND B
 if (immed2.read() == 1)
 {
            op_b2.write((int)immed322);
```

```
                }
                else
                {
                        addr = s6.read();
                        op_b2.write(reg[addr]);
                }
               // OPERAND A
               addr = s5.read();
               op_a2.write(reg[addr]);
               // TO MEM
               addr = s5.read();


               /// READ REG PE3 //////////
                // OPERAND B
                if (immed3.read() == 1)
                {
                         op_b3.write((int)immed323);
                }
                else
                {
                        addr = s8.read();
                        op_b3.write(reg[addr]);
                }
               // OPERAND A
               addr = s7.read();
               op_a3.write(reg[addr]);
               // TO MEM
               addr = s7.read();


        }
       // all cycles
       wait();
         }

}
```

A.16 SYSTEMC VLIW SOURCE FILE: STIMULUS.CPP

```
#include <systemc.h>
#include "regfile.h"
#include "directives.h"

void regfile::entry() {


        while(1)
        {

         // Reset to zero
         if (reset.read() == 1)
         {
                for (int i=0; i<SIZE_REG; i++)
                {
                        reg[i] = 0;
                }
         }
         else
         {

                // WR_REG 0
                en = wr_reg.read();
                if (en == 1)
                {
                        addr = dest.read();
                         if ((int)addr != 0)
                         {
                                reg[addr] = reg_din.read();
                         }
                }
                // WR_REG 1
                en = wr_reg1.read();
                if (en == 1)
                {
                         addr = dest1.read();
                         if ((int)addr != 0)
                         {
                                reg[addr] = reg_din1.read();
                         }
                }
                // WR_REG 2
                en = wr_reg2.read();
                if (en == 1)
                {
                         addr = dest2.read();
                         if ((int)addr != 0)
                         {
```

```
                reg[addr] = reg_din2.read();
        }
}
// WR_REG 3
en = wr_reg3.read();
if (en == 1)
{
        addr = dest3.read();
        if ((int)addr != 0)
        {
                reg[addr] = reg_din3.read();
        }
}

/// READ REG PE0 ////////
// OPERAND B
if (immed.read() == 1)
{
        op_b.write((int)immed32);
}
else
{
        addr = s2.read();
        op_b.write(reg[addr]);
}
// OPERAND A
addr = s1.read();
op_a.write(reg[addr]);
// TO MEM
addr = s2.read();
to_mem.write(reg[addr]);

/// READ REG PE1 //////////
 // OPERAND B
 if (immed1.read() == 1)
 {
        op_b1.write((int)immed321);
 }
 else
 {
        addr = s4.read();
        op_b1.write(reg[addr]);
 }
// OPERAND A
addr = s3.read();
op_a1.write(reg[addr]);
// TO MEM
addr = s3.read();
to_mem1.write(reg[addr]);


/// READ REG PE2 //////////
 // OPERAND B
 if (immed2.read() == 1)
 {
```

```
                    op_b2.write((int)immed322);
                }
                else
                {
                        addr = s6.read();
                        op_b2.write(reg[addr]);
                }
                // OPERAND A
                addr = s5.read();
                op_a2.write(reg[addr]);
                // TO MEM
                addr = s5.read();


                /// READ REG PE3 //////////
                 // OPERAND B
                 if (immed3.read() == 1)
                 {
                        op_b3.write((int)immed323);
                 }
                 else
                 {
                        addr = s8.read();
                        op_b3.write(reg[addr]);
                 }
                // OPERAND A
                addr = s7.read();
                op_a3.write(reg[addr]);
                // TO MEM
                addr = s7.read();


        }
        // all cycles
        wait();
          }
}
```

A.17 VHDL VLIW SOURCE FILE: TOP_SYSTEM_4PE_STRUCT.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY top_ALU IS
  PORT(
    add_sub      : IN     std_logic;
    alu_or_mem   : IN     std_logic;
    byte_lane    : IN     std_logic_vector (1 DOWNTO 0);
    clock        : IN     std_logic;
    cmp          : IN     std_logic_vector (2 DOWNTO 0);
    left_right   : IN     std_logic;
    lo_hi        : IN     std_logic;
    logic_op     : IN     std_logic_vector (1 DOWNTO 0);
    mem_data_out : IN     std_logic_vector (31 DOWNTO 0);
    op_sel       : IN     std_logic_vector (1 DOWNTO 0);
    reset        : IN     std_logic;
    rot_log_arith : IN    std_logic_vector (1 DOWNTO 0);
    s1           : IN     std_logic_vector (31 DOWNTO 0);
    s2           : IN     std_logic_vector (31 DOWNTO 0);
    s2_4_mem     : IN     std_logic_vector (31 DOWNTO 0);
    signed_op    : IN     std_logic;
    signed_op2   : IN     std_logic;
    alu_out      : OUT    std_logic_vector (31 DOWNTO 0);
    flags        : OUT    std_logic_vector (2 DOWNTO 0);
    mem_addr     : OUT    std_logic_vector (31 DOWNTO 0);
    mem_data_in  : OUT    std_logic_vector (31 DOWNTO 0);
    st_bytelane  : OUT    std_logic_vector (1 DOWNTO 0)
  );

-- Declarations

END top_ALU ;

--
-- VHDL Architecture Auto_Gen.top_ALU.struct
--
-- Created:
--         by - Dara.UNKNOWN (J11)
--         at - 17:15:35 07/22/2005
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

LIBRARY Auto_Gen;
```

```
ARCHITECTURE struct OF top_ALU IS

  -- Internal signal declarations
  SIGNAL add_in        : std_logic_vector(31 DOWNTO 0);
  SIGNAL add_out       : std_logic_vector(31 DOWNTO 0);
  SIGNAL alu_out_muxin : std_logic_vector(31 DOWNTO 0);
  SIGNAL flags_in      : std_logic_vector(2 DOWNTO 0);
  SIGNAL lo_hi_mult    : std_logic;
  SIGNAL load_out      : std_logic_vector(31 DOWNTO 0);
  SIGNAL log_in        : std_logic_vector(31 DOWNTO 0);
  SIGNAL log_out       : std_logic_vector(31 DOWNTO 0);
  SIGNAL mem_addr_in   : std_logic_vector(31 DOWNTO 0);
  SIGNAL mult_out      : std_logic_vector(31 DOWNTO 0);
  SIGNAL mux_out       : std_logic_vector(31 DOWNTO 0);
  SIGNAL s1_mult       : std_logic_vector(31 DOWNTO 0);
  SIGNAL s2_mult       : std_logic_vector(31 DOWNTO 0);
  SIGNAL shift_in      : std_logic_vector(31 DOWNTO 0);
  SIGNAL shift_out     : std_logic_vector(31 DOWNTO 0);


  -- Component Declarations
  COMPONENT ALU_adder
  PORT (
    add_sub   : IN     std_logic ;
    cmp       : IN     std_logic_vector (2 DOWNTO 0);
    s1        : IN     std_logic_vector (31 DOWNTO 0);
    s2        : IN     std_logic_vector (31 DOWNTO 0);
    signed_op : IN     std_logic ;
    alu_out   : OUT    std_logic_vector (31 DOWNTO 0);
    flags     : OUT    std_logic_vector (2 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT ALU_load
  PORT (
    byte_lane   : IN     std_logic_vector (1 DOWNTO 0);
    clock       : IN     std_logic ;
    mem_data_in : IN     std_logic_vector (31 DOWNTO 0);
    reset       : IN     std_logic ;
    signed_wb   : IN     std_logic ;
    load_data   : OUT    std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT ALU_logical
  PORT (
    logic_op : IN     std_logic_vector (1 DOWNTO 0);
    s1       : IN     std_logic_vector (31 DOWNTO 0);
    s2       : IN     std_logic_vector (31 DOWNTO 0);
    alu_out  : OUT    std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT ALU_mem_addr
  PORT (
    s1     : IN     std_logic_vector (31 DOWNTO 0);
    s2     : IN     std_logic_vector (31 DOWNTO 0);
```

```
  alu_out : OUT    std_logic_vector (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT ALU_multiplier
PORT (
  lo_hi   : IN     std_logic ;
  s1      : IN     std_logic_vector (31 DOWNTO 0);
  s2      : IN     std_logic_vector (31 DOWNTO 0);
  alu_out : OUT    std_logic_vector (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT ALU_shifter
PORT (
  left_right    : IN     std_logic ;
  rot_log_arith : IN     std_logic_vector (1 DOWNTO 0);
  s1            : IN     std_logic_vector (31 DOWNTO 0);
  s2            : IN     std_logic_vector (31 DOWNTO 0);
  alu_out       : OUT    std_logic_vector (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT ALU_store
PORT (
  add_out     : IN     std_logic_vector (31 DOWNTO 0);
  byte_lane   : IN     std_logic_vector (1 DOWNTO 0);
  clock       : IN     std_logic ;
  reset       : IN     std_logic ;
  s2_4_mem    : IN     std_logic_vector (31 DOWNTO 0);
  mem_addr    : OUT    std_logic_vector (31 DOWNTO 0);
  mem_data_in : OUT    std_logic_vector (31 DOWNTO 0);
  st_bytelane : OUT    std_logic_vector (1 DOWNTO 0)
);
END COMPONENT;
COMPONENT Generic_Reg_noenable
GENERIC (
  SizeIn  : integer;
  SizeOut : integer
);
PORT (
  A_in  : IN     std_logic_vector (SizeIn - 1 DOWNTO 0);
  clock : IN     std_logic;
  reset : IN     std_logic;
  A_out : OUT    std_logic_vector (SizeOut - 1 DOWNTO 0)
);
END COMPONENT;
COMPONENT Reg_1
PORT (
  clock : IN     std_logic ;
  d_in  : IN     std_logic ;
  reset : IN     std_logic ;
  q_out : OUT    std_logic
);
END COMPONENT;
COMPONENT Reg_32_noenable
PORT (
  clock   : IN     std_logic ;
```

```
    reg_in  : IN     std_logic_vector (31 DOWNTO 0);
    reset   : IN     std_logic ;
    reg_out : OUT    std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;

  -- Optional embedded configurations
  -- pragma synthesis_off
  FOR ALL : ALU_adder USE ENTITY Auto_Gen.ALU_adder;
  FOR ALL : ALU_load USE ENTITY Auto_Gen.ALU_load;
  FOR ALL : ALU_logical USE ENTITY Auto_Gen.ALU_logical;
  FOR ALL : ALU_mem_addr USE ENTITY Auto_Gen.ALU_mem_addr;
  FOR ALL : ALU_multiplier USE ENTITY Auto_Gen.ALU_multiplier;
  FOR ALL : ALU_shifter USE ENTITY Auto_Gen.ALU_shifter;
  FOR ALL : ALU_store USE ENTITY Auto_Gen.ALU_store;
  FOR ALL : Generic_Reg_noenable USE ENTITY Auto_Gen.Generic_Reg_noenable;
  FOR ALL : Reg_1 USE ENTITY Auto_Gen.Reg_1;
  FOR ALL : Reg_32_noenable USE ENTITY Auto_Gen.Reg_32_noenable;
  -- pragma synthesis_on


BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Block 1 eb1
  -- Non hierarchical truthtable
  ---------------------------------------------------------------------------
  eb1_truth_process: PROCESS(alu_or_mem, alu_out_muxin, load_out)
  ---------------------------------------------------------------------------
  BEGIN
    -- Block 1
    CASE alu_or_mem IS
    WHEN '0' =>
      alu_out <= alu_out_muxin;
    WHEN '1' =>
      alu_out <= load_out;
    WHEN OTHERS =>
      alu_out <= alu_out_muxin;
    END CASE;

  END PROCESS eb1_truth_process;

  -- Architecture concurrent statements



  -- HDL Embedded Block 3 eb3
  -- Non hierarchical truthtable
  ---------------------------------------------------------------------------
  eb3_truth_process: PROCESS(add_out, log_out, mult_out, op_sel, shift_out)
  ---------------------------------------------------------------------------
  BEGIN
    -- Block 1
    CASE op_sel IS
    WHEN "00" =>
      mux_out <= add_out;
```

```
  WHEN "01" =>
    mux_out <= mult_out;
  WHEN "10" =>
    mux_out <= log_out;
  WHEN "11" =>
    mux_out <= shift_out;
  WHEN OTHERS =>
    mux_out <= add_out;
  END CASE;

END PROCESS eb3_truth_process;

-- Architecture concurrent statements

-- Instance port mappings.
I0 : ALU_adder
  PORT MAP (
    add_sub   => add_sub,
    cmp       => cmp,
    s1        => s1,
    s2        => s2,
    signed_op => signed_op,
    alu_out   => add_in,
    flags     => flags_in
  );
load_reg : ALU_load
  PORT MAP (
    byte_lane   => byte_lane,
    clock       => clock,
    mem_data_in => mem_data_out,
    reset       => reset,
    signed_wb   => signed_op,
    load_data   => load_out
  );
I1 : ALU_logical
  PORT MAP (
    logic_op => logic_op,
    s1       => s1,
    s2       => s2,
    alu_out  => log_in
  );
I10 : ALU_mem_addr
  PORT MAP (
    s1      => s1,
    s2      => s2,
    alu_out => mem_addr_in
  );
I3 : ALU_multiplier
  PORT MAP (
    lo_hi   => lo_hi_mult,
    s1      => s1_mult,
    s2      => s2_mult,
    alu_out => mult_out
  );
I2 : ALU_shifter
```

```
  PORT MAP (
    left_right    => left_right,
    rot_log_arith => rot_log_arith,
    s1            => s1,
    s2            => s2,
    alu_out       => shift_in
  );
store_reg : ALU_store
  PORT MAP (
    add_out     => mem_addr_in,
    byte_lane   => byte_lane,
    clock       => clock,
    reset       => reset,
    s2_4_mem    => s2_4_mem,
    mem_addr    => mem_addr,
    mem_data_in => mem_data_in,
    st_bytelane => st_bytelane
  );
I4 : Generic_Reg_noenable
  GENERIC MAP (
    SizeIn  => 3,
    SizeOut => 3
  )
  PORT MAP (
    clock => clock,
    reset => reset,
    A_in  => flags_in,
    A_out => flags
  );
I5 : Reg_1
  PORT MAP (
    clock => clock,
    d_in  => lo_hi,
    reset => reset,
    q_out => lo_hi_mult
  );
adder_reg : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => add_in,
    reset   => reset,
    reg_out => add_out
  );
ex2_reg : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => mux_out,
    reset   => reset,
    reg_out => alu_out_muxin
  );
log_reg : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => log_in,
    reset   => reset,
```

```
      reg_out => log_out
    );
  mult_reg : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => s1,
      reset   => reset,
      reg_out => s1_mult
    );
  mult_reg1 : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => s2,
      reset   => reset,
      reg_out => s2_mult
    );
  shift_reg : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => shift_in,
      reset   => reset,
      reg_out => shift_out
    );

END struct;
```

A.18 VHDL VLIW SOURCE FILE: TOP_ALU_AND_DECODER.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY top_ALU_and_decoder IS
  PORT(
    addr          : IN     std_logic_vector (31 DOWNTO 0);
    clock         : IN     std_logic;
    inst          : IN     std_logic_vector (31 DOWNTO 0);
    mem_data_out_b : IN     STD_LOGIC_VECTOR (31 DOWNTO 0);
    reset         : IN     std_logic;
    s1            : IN     std_logic_vector (31 DOWNTO 0);
    s2            : IN     std_logic_vector (31 DOWNTO 0);
    s2_4_mem      : IN     std_logic_vector (31 DOWNTO 0);
    take_branch   : IN     std_logic;
    alu_out       : OUT    std_logic_vector (31 DOWNTO 0);
    base_addr     : OUT    std_logic_vector (31 DOWNTO 0);
    br_code       : OUT    std_logic_vector (5 DOWNTO 0);
    br_ext        : OUT    std_logic_vector (2 DOWNTO 0);
    branch_check  : OUT    std_logic;
    call          : OUT    std_logic;
    callr         : OUT    std_logic;
    flags         : OUT    std_logic_vector (2 DOWNTO 0);
    flush_lo      : OUT    std_logic;
    immed         : OUT    std_logic;
    immed_16_or_5 : OUT    std_logic;
    immed_val     : OUT    std_logic_vector (31 DOWNTO 0);
    jmp           : OUT    std_logic;
    mem_addr      : OUT    std_logic_vector (31 DOWNTO 0);
    mem_data_in   : OUT    std_logic_vector (31 DOWNTO 0);
    op_a          : OUT    std_logic_vector (4 DOWNTO 0);
    op_b          : OUT    std_logic_vector (4 DOWNTO 0);
    op_immed      : OUT    std_logic;
    rd_mem        : OUT    std_logic;
    ret           : OUT    std_logic;
    st_bytelane   : OUT    std_logic_vector (1 DOWNTO 0);
    wr_mem        : OUT    std_logic;
    wr_reg        : OUT    std_logic;
    wr_reg_addr   : OUT    std_logic_vector (4 DOWNTO 0)
  );

-- Declarations

END top_ALU_and_decoder ;

enerated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
USE ieee.std_logic_arith.all;

LIBRARY Auto_Gen;

ARCHITECTURE struct OF top_ALU_and_decoder IS

  -- Architecture declarations
  -- Internal signal declarations
  SIGNAL add_sub3       : std_logic;
  SIGNAL alu_or_mem3    : std_logic;
  SIGNAL byte_lane      : std_logic_vector(1 DOWNTO 0);
  SIGNAL cmp3           : std_logic_vector(2 DOWNTO 0);
  SIGNAL dest           : std_logic_vector(4 DOWNTO 0);
  SIGNAL left_right3    : std_logic;
  SIGNAL lo_hi3         : std_logic;
  SIGNAL logic_op3      : std_logic_vector(1 DOWNTO 0);
  SIGNAL op_sel3        : std_logic_vector(1 DOWNTO 0);
  SIGNAL operands       : std_logic_vector(14 DOWNTO 0);
  SIGNAL rot_log_arith3 : std_logic_vector(1 DOWNTO 0);
  SIGNAL signed_op6     : std_logic;
  SIGNAL signed_op7     : std_logic;

  -- Implicit buffer signal declarations
  SIGNAL immed_16_or_5_internal : std_logic;
  SIGNAL op_immed_internal      : std_logic;


  -- Component Declarations
  COMPONENT Hold_dest
  PORT (
    clock    : IN     std_logic ;
    dest     : IN     std_logic_vector (4 DOWNTO 0);
    reset    : IN     std_logic ;
    dest_reg : OUT    std_logic_vector (4 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT Operand_splitter
  PORT (
    immed        : IN     std_logic ;
    immed_16_or_5 : IN     std_logic ;
    operands     : IN     std_logic_vector (14 DOWNTO 0);
    dest         : OUT    std_logic_vector (4 DOWNTO 0);
    op_a         : OUT    std_logic_vector (4 DOWNTO 0);
    op_b         : OUT    std_logic_vector (4 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT top_ALU
  PORT (
    add_sub      : IN     std_logic ;
    alu_or_mem   : IN     std_logic ;
    byte_lane    : IN     std_logic_vector (1 DOWNTO 0);
    clock        : IN     std_logic ;
    cmp          : IN     std_logic_vector (2 DOWNTO 0);
    left_right   : IN     std_logic ;
    lo_hi        : IN     std_logic ;
```

```
  logic_op      : IN     std_logic_vector (1 DOWNTO 0);
  mem_data_out  : IN     std_logic_vector (31 DOWNTO 0);
  op_sel        : IN     std_logic_vector (1 DOWNTO 0);
  reset         : IN     std_logic ;
  rot_log_arith : IN     std_logic_vector (1 DOWNTO 0);
  s1            : IN     std_logic_vector (31 DOWNTO 0);
  s2            : IN     std_logic_vector (31 DOWNTO 0);
  s2_4_mem      : IN     std_logic_vector (31 DOWNTO 0);
  signed_op     : IN     std_logic ;
  signed_op2    : IN     std_logic ;
  alu_out       : OUT    std_logic_vector (31 DOWNTO 0);
  flags         : OUT    std_logic_vector (2 DOWNTO 0);
  mem_addr      : OUT    std_logic_vector (31 DOWNTO 0);
  mem_data_in   : OUT    std_logic_vector (31 DOWNTO 0);
  st_bytelane   : OUT    std_logic_vector (1 DOWNTO 0)
);
END COMPONENT;
COMPONENT top_decoder_4pe
PORT (
  addr          : IN     std_logic_vector (31 DOWNTO 0);
  clock         : IN     std_logic ;
  inst          : IN     std_logic_vector (31 DOWNTO 0);
  reset         : IN     std_logic ;
  take_branch   : IN     std_logic ;
  add_sub       : OUT    std_logic ;
  alu_or_mem    : OUT    std_logic ;
  base_addr     : OUT    std_logic_vector (31 DOWNTO 0);
  br_code       : OUT    std_logic_vector (5 DOWNTO 0);
  br_ext        : OUT    std_logic_vector (2 DOWNTO 0);
  branch_check  : OUT    std_logic ;
  byte_lane     : OUT    std_logic_vector (1 DOWNTO 0);
  call          : OUT    std_logic ;
  callr         : OUT    std_logic ;
  cmp           : OUT    std_logic_vector (2 DOWNTO 0);
  ex1_immed     : OUT    std_logic ;
  flush_lo      : OUT    std_logic ;
  immed_16_or_5 : OUT    std_logic ;
  immed_val     : OUT    std_logic_vector (31 DOWNTO 0);
  jmp           : OUT    std_logic ;
  left_right    : OUT    std_logic ;
  lo_hi         : OUT    std_logic ;
  logic_op      : OUT    std_logic_vector (1 DOWNTO 0);
  op_immed      : OUT    std_logic ;
  op_sel        : OUT    std_logic_vector (1 DOWNTO 0);
  operands      : OUT    std_logic_vector (14 DOWNTO 0);
  rd_mem        : OUT    std_logic ;
  ret           : OUT    std_logic ;
  rot_log_arith : OUT    std_logic_vector (1 DOWNTO 0);
  signed_op     : OUT    std_logic ;
  signed_op2    : OUT    std_logic ;
  wr_mem        : OUT    std_logic ;
  wr_reg        : OUT    std_logic
);
END COMPONENT;
```

```
  -- Optional embedded configurations
  -- pragma synthesis_off
  FOR ALL : Hold_dest USE ENTITY Auto_Gen.Hold_dest;
  FOR ALL : Operand_splitter USE ENTITY Auto_Gen.Operand_splitter;
  FOR ALL : top_ALU USE ENTITY Auto_Gen.top_ALU;
  FOR ALL : top_decoder_4pe USE ENTITY Auto_Gen.top_decoder_4pe;
  -- pragma synthesis_on

BEGIN

  -- Instance port mappings.
  I5 : Hold_dest
    PORT MAP (
      clock    => clock,
      dest     => dest,
      reset    => reset,
      dest_reg => wr_reg_addr
    );
  I14 : Operand_splitter
    PORT MAP (
      immed        => op_immed_internal,
      immed_16_or_5 => immed_16_or_5_internal,
      operands     => operands,
      dest         => dest,
      op_a         => op_a,
      op_b         => op_b
    );
  ALU : top_ALU
    PORT MAP (
      add_sub      => add_sub3,
      alu_or_mem   => alu_or_mem3,
      byte_lane    => byte_lane,
      clock        => clock,
      cmp          => cmp3,
      left_right   => left_right3,
      lo_hi        => lo_hi3,
      logic_op     => logic_op3,
      mem_data_out  => mem_data_out_b,
      op_sel       => op_sel3,
      reset        => reset,
      rot_log_arith => rot_log_arith3,
      s1           => s1,
      s2           => s2,
      s2_4_mem     => s2_4_mem,
      signed_op    => signed_op6,
      signed_op2   => signed_op7,
      alu_out      => alu_out,
      flags        => flags,
      mem_addr     => mem_addr,
      mem_data_in  => mem_data_in,
      st_bytelane  => st_bytelane
    );
  top_decode : top_decoder_4pe
    PORT MAP (
      addr         => addr,
```

```
      clock         => clock,
      inst          => inst,
      reset         => reset,
      take_branch   => take_branch,
      add_sub       => add_sub3,
      alu_or_mem    => alu_or_mem3,
      base_addr     => base_addr,
      br_code       => br_code,
      br_ext        => br_ext,
      branch_check  => branch_check,
      byte_lane     => byte_lane,
      call          => call,
      callr         => callr,
      cmp           => cmp3,
      ex1_immed     => immed,
      flush_lo      => flush_lo,
      immed_16_or_5 => immed_16_or_5_internal,
      immed_val     => immed_val,
      jmp           => jmp,
      left_right    => left_right3,
      lo_hi         => lo_hi3,
      logic_op      => logic_op3,
      op_immed      => op_immed_internal,
      op_sel        => op_sel3,
      operands      => operands,
      rd_mem        => rd_mem,
      ret           => ret,
      rot_log_arith => rot_log_arith3,
      signed_op     => signed_op6,
      signed_op2    => signed_op7,
      wr_mem        => wr_mem,
      wr_reg        => wr_reg
    );

  -- Implicit buffered output assignments
  immed_16_or_5 <= immed_16_or_5_internal;
  op_immed      <= op_immed_internal;

END struct;
```

A.19 VHDL VLIW SOURCE FILE: TOP_REGISTER_32X32X4W.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY top_register_32x32x4w IS
  PORT(
    clock      : IN    std_logic;
    immed_0    : IN    std_logic;
    immed_1    : IN    std_logic;
    immed_2    : IN    std_logic;
    immed_3    : IN    std_logic;
    immed_val_0 : IN    std_logic_vector (31 DOWNTO 0);
    immed_val_1 : IN    std_logic_vector (31 DOWNTO 0);
    immed_val_2 : IN    std_logic_vector (31 DOWNTO 0);
    immed_val_3 : IN    std_logic_vector (31 DOWNTO 0);
    op_a_0     : IN    std_logic_vector (4 DOWNTO 0);
    op_a_1     : IN    std_logic_vector (4 DOWNTO 0);
    op_a_2     : IN    std_logic_vector (4 DOWNTO 0);
    op_a_3     : IN    std_logic_vector (4 DOWNTO 0);
    op_b_0     : IN    std_logic_vector (4 DOWNTO 0);
    op_b_1     : IN    std_logic_vector (4 DOWNTO 0);
    op_b_2     : IN    std_logic_vector (4 DOWNTO 0);
    op_b_3     : IN    std_logic_vector (4 DOWNTO 0);
    reset      : IN    std_logic;
    wr_addr_a  : IN    std_logic_vector (4 DOWNTO 0);
    wr_addr_b  : IN    std_logic_vector (4 DOWNTO 0);
    wr_addr_c  : IN    std_logic_vector (4 DOWNTO 0);
    wr_addr_d  : IN    std_logic_vector (4 DOWNTO 0);
    wr_data_a  : IN    std_logic_vector (31 DOWNTO 0);
    wr_data_b  : IN    std_logic_vector (31 DOWNTO 0);
    wr_data_c  : IN    std_logic_vector (31 DOWNTO 0);
    wr_data_d  : IN    std_logic_vector (31 DOWNTO 0);
    wr_reg_a   : IN    std_logic;
    wr_reg_b   : IN    std_logic;
    wr_reg_c   : IN    std_logic;
    wr_reg_d   : IN    std_logic;
    s1_0       : OUT   std_logic_vector (31 DOWNTO 0);
    s1_1       : OUT   std_logic_vector (31 DOWNTO 0);
    s1_2       : OUT   std_logic_vector (31 DOWNTO 0);
    s1_3       : OUT   std_logic_vector (31 DOWNTO 0);
    s2_0       : OUT   std_logic_vector (31 DOWNTO 0);
    s2_1       : OUT   std_logic_vector (31 DOWNTO 0);
    s2_2       : OUT   std_logic_vector (31 DOWNTO 0);
    s2_3       : OUT   std_logic_vector (31 DOWNTO 0);
    s2_4_mem_0  : OUT   std_logic_vector (31 DOWNTO 0);
    s2_4_mem_1  : OUT   std_logic_vector (31 DOWNTO 0);
    s2_4_mem_2  : OUT   std_logic_vector (31 DOWNTO 0);
    s2_4_mem_3  : OUT   std_logic_vector (31 DOWNTO 0)
  );
```

```
-- Declarations

END top_register_32x32x4w ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

LIBRARY Auto_Gen;

ARCHITECTURE struct OF top_register_32x32x4w IS

  -- Internal signal declarations
  SIGNAL B1_0      : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_0_out   : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_1      : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_1_out   : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_2      : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_2_out   : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_3      : std_logic_vector(31 DOWNTO 0);
  SIGNAL B1_3_out   : std_logic_vector(31 DOWNTO 0);
  SIGNAL enable_bus : std_logic_vector(31 DOWNTO 0);
  SIGNAL mux_in     : std_logic_vector(127 DOWNTO 0);
  SIGNAL mux_out    : std_logic_vector(255 DOWNTO 0);
  SIGNAL mux_out1   : std_logic_vector(1023 DOWNTO 0);
  SIGNAL reg_out    : std_logic_vector(1023 DOWNTO 0);
  SIGNAL s1_regin_0 : std_logic_vector(31 DOWNTO 0);
  SIGNAL s1_regin_1 : std_logic_vector(31 DOWNTO 0);
  SIGNAL s1_regin_2 : std_logic_vector(31 DOWNTO 0);
  SIGNAL s1_regin_3 : std_logic_vector(31 DOWNTO 0);
  SIGNAL sel_bus    : std_logic_vector(39 DOWNTO 0);
  SIGNAL select_bus : std_logic_vector(63 DOWNTO 0);


  -- Component Declarations
  COMPONENT Enable_decoder_4W
  PORT (
    wr_addr_a  : IN     std_logic_vector (4 DOWNTO 0);
    wr_addr_b  : IN     std_logic_vector (4 DOWNTO 0);
    wr_addr_c  : IN     std_logic_vector (4 DOWNTO 0);
    wr_addr_d  : IN     std_logic_vector (4 DOWNTO 0);
    wr_reg_a   : IN     std_logic ;
    wr_reg_b   : IN     std_logic ;
    wr_reg_c   : IN     std_logic ;
    wr_reg_d   : IN     std_logic ;
    enable_bus : OUT    std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT Mux_2x1
  PORT (
    muxin_a : IN     std_logic_vector (31 DOWNTO 0);
    muxin_b : IN     std_logic_vector (31 DOWNTO 0);
    sel     : IN     std_logic ;
    muxout  : OUT    std_logic_vector (31 DOWNTO 0)
```

```
);
END COMPONENT;
COMPONENT Reg_32_noenable
PORT (
  clock   : IN     std_logic ;
  reg_in  : IN     std_logic_vector (31 DOWNTO 0);
  reset   : IN     std_logic ;
  reg_out : OUT    std_logic_vector (31 DOWNTO 0)
);
END COMPONENT;
COMPONENT Select_decoder_4W
PORT (
  wr_addr_a  : IN     std_logic_vector (4 DOWNTO 0);
  wr_addr_b  : IN     std_logic_vector (4 DOWNTO 0);
  wr_addr_c  : IN     std_logic_vector (4 DOWNTO 0);
  wr_addr_d  : IN     std_logic_vector (4 DOWNTO 0);
  wr_reg_a   : IN     std_logic ;
  wr_reg_b   : IN     std_logic ;
  wr_reg_c   : IN     std_logic ;
  wr_reg_d   : IN     std_logic ;
  select_bus : OUT    std_logic_vector (63 DOWNTO 0)
);
END COMPONENT;
COMPONENT mux_bank_32x32x8
PORT (
  mux_in  : IN     std_logic_vector (1023 DOWNTO 0);
  sel_bus : IN     std_logic_vector (39 DOWNTO 0);
  mux_out : OUT    std_logic_vector (255 DOWNTO 0)
);
END COMPONENT;
COMPONENT mux_bank_32x4x1
PORT (
  mux_in  : IN     std_logic_vector (127 DOWNTO 0);
  sel_bus : IN     std_logic_vector (63 DOWNTO 0);
  mux_out : OUT    std_logic_vector (1023 DOWNTO 0)
);
END COMPONENT;
COMPONENT reg_bank_32x32
PORT (
  clock      : IN     std_logic ;
  enable_bus : IN     std_logic_vector (31 DOWNTO 0);
  reg_in     : IN     std_logic_vector (1023 DOWNTO 0);
  reset      : IN     std_logic ;
  reg_out    : OUT    std_logic_vector (1023 DOWNTO 0)
);
END COMPONENT;

-- Optional embedded configurations
-- pragma synthesis_off
FOR ALL : Enable_decoder_4W USE ENTITY Auto_Gen.Enable_decoder_4W;
FOR ALL : Mux_2x1 USE ENTITY Auto_Gen.Mux_2x1;
FOR ALL : Reg_32_noenable USE ENTITY Auto_Gen.Reg_32_noenable;
FOR ALL : Select_decoder_4W USE ENTITY Auto_Gen.Select_decoder_4W;
FOR ALL : mux_bank_32x32x8 USE ENTITY Auto_Gen.mux_bank_32x32x8;
FOR ALL : mux_bank_32x4x1 USE ENTITY Auto_Gen.mux_bank_32x4x1;
```

```
  FOR ALL : reg_bank_32x32 USE ENTITY Auto_Gen.reg_bank_32x32;
  -- pragma synthesis_on


BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Text Block 1 eb1
  B1_0 <= mux_out(31 downto 0);
  s1_regin_0 <= mux_out(63 downto 32);
  B1_1 <= mux_out(95 downto 64);
  s1_regin_1 <= mux_out(127 downto 96);
  B1_2 <= mux_out(159 downto 128);
  s1_regin_2 <= mux_out(191 downto 160);
  B1_3 <= mux_out(223 downto 192);
  s1_regin_3 <= mux_out(255 downto 224);

  sel_bus <= op_a_3 & op_b_3 & op_a_2 & op_b_2 & op_a_1 & op_b_1 & op_a_0 & op_b_0;
  mux_in <= wr_data_d & wr_data_c & wr_data_b & wr_data_a;

  -- HDL Embedded Text Block 3 eb3
  s2_4_mem_0 <= B1_0_out;

  -- HDL Embedded Text Block 4 eb4
  s2_4_mem_1 <= B1_1_out;

  -- HDL Embedded Text Block 6 eb6
  s2_4_mem_2 <= B1_2_out;

  -- HDL Embedded Text Block 8 eb8
  s2_4_mem_3 <= B1_3_out;


  -- Instance port mappings.
  I4 : Enable_decoder_4W
    PORT MAP (
      wr_addr_a  => wr_addr_a,
      wr_addr_b  => wr_addr_b,
      wr_addr_c  => wr_addr_c,
      wr_addr_d  => wr_addr_d,
      wr_reg_a   => wr_reg_a,
      wr_reg_b   => wr_reg_b,
      wr_reg_c   => wr_reg_c,
      wr_reg_d   => wr_reg_d,
      enable_bus => enable_bus
    );
  I13 : Mux_2x1
    PORT MAP (
      muxin_a => B1_0_out,
      muxin_b => immed_val_0,
      sel     => immed_0,
      muxout  => s2_0
    );
  I14 : Mux_2x1
    PORT MAP (
      muxin_a => B1_1_out,
```

```
    muxin_b => immed_val_1,
    sel     => immed_1,
    muxout  => s2_1
  );
I15 : Mux_2x1
  PORT MAP (
    muxin_a => B1_2_out,
    muxin_b => immed_val_2,
    sel     => immed_2,
    muxout  => s2_2
  );
I16 : Mux_2x1
  PORT MAP (
    muxin_a => B1_3_out,
    muxin_b => immed_val_3,
    sel     => immed_3,
    muxout  => s2_3
  );
s0a : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => s1_regin_0,
    reset   => reset,
    reg_out => s1_0
  );
s0b : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => B1_0,
    reset   => reset,
    reg_out => B1_0_out
  );
s1a : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => s1_regin_1,
    reset   => reset,
    reg_out => s1_1
  );
s1b : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => B1_1,
    reset   => reset,
    reg_out => B1_1_out
  );
s2a : Reg_32_noenable
  PORT MAP (
    clock   => clock,
    reg_in  => s1_regin_2,
    reset   => reset,
    reg_out => s1_2
  );
s2b : Reg_32_noenable
  PORT MAP (
```

```
      clock   => clock,
      reg_in  => B1_2,
      reset   => reset,
      reg_out => B1_2_out
    );
  s3a : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => s1_regin_3,
      reset   => reset,
      reg_out => s1_3
    );
  s3b : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => B1_3,
      reset   => reset,
      reg_out => B1_3_out
    );
  I12 : Select_decoder_4W
    PORT MAP (
      wr_addr_a  => wr_addr_a,
      wr_addr_b  => wr_addr_b,
      wr_addr_c  => wr_addr_c,
      wr_addr_d  => wr_addr_d,
      wr_reg_a   => wr_reg_a,
      wr_reg_b   => wr_reg_b,
      wr_reg_c   => wr_reg_c,
      wr_reg_d   => wr_reg_d,
      select_bus => select_bus
    );
  I5 : mux_bank_32x32x8
    PORT MAP (
      mux_in  => reg_out,
      sel_bus => sel_bus,
      mux_out => mux_out
    );
  I0 : mux_bank_32x4x1
    PORT MAP (
      mux_in  => mux_in,
      sel_bus => select_bus,
      mux_out => mux_out1
    );
  regbank : reg_bank_32x32
    PORT MAP (
      clock      => clock,
      enable_bus => enable_bus,
      reg_in     => mux_out1,
      reset      => reset,
      reg_out    => reg_out
    );

END struct;
```

A.20 VHDL VLIW SOURCE FILE: DECODER_NIOS.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Decoder_NIOS IS
  PORT(
    addr       : IN     std_logic_vector (31 DOWNTO 0);
    clock      : IN     std_logic;
    opcode     : IN     std_logic_vector (5 DOWNTO 0);
    other      : IN     std_logic_vector (5 DOWNTO 0);
    reset      : IN     std_logic;
    take_branch : IN     std_logic;
    base_addr   : OUT    std_logic_vector (31 DOWNTO 0);
    ctl_ex1     : OUT    std_logic_vector (16 DOWNTO 0);
    ctl_ex2     : OUT    std_logic_vector (14 DOWNTO 0);
    ctl_op      : OUT    std_logic_vector (4 DOWNTO 0);
    ctl_wb      : OUT    std_logic_vector (1 DOWNTO 0);
    flush_lo    : OUT    std_logic
  );

-- Declarations

END Decoder_NIOS ;

--
-- VHDL Architecture Auto_Gen.Decoder_NIOS.struct
--
-- Created:
--         by - Dara.UNKNOWN (J11)
--         at - 14:38:27 07/22/2005
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.numeric_std.all;

LIBRARY Auto_Gen;

ARCHITECTURE struct OF Decoder_NIOS IS

  -- Architecture declarations
  -- Instructions(5 downto 0)
       Constant call : std_logic_vector(6 downto 0) := "0000000";
       Constant ldbu : std_logic_vector(6 downto 0) :=  "0000011";
       Constant addi : std_logic_vector(6 downto 0) :=  "0000100";
       Constant stb : std_logic_vector(6 downto 0) :=  "0000101";
       Constant br : std_logic_vector(6 downto 0) :=  "0000110";
```

```
Constant ldb : std_logic_vector(6 downto 0) :=  "0000111";
Constant cmpgei : std_logic_vector(6 downto 0) :=  "0001000";
Constant ldhu : std_logic_vector(6 downto 0) :=  "0001011";
Constant andi : std_logic_vector(6 downto 0) :=  "0001100";
Constant sth : std_logic_vector(6 downto 0) :=  "0001101";
Constant bge : std_logic_vector(6 downto 0) :=  "0001110";
Constant ldh : std_logic_vector(6 downto 0) :=  "0001111";
Constant cmplti : std_logic_vector(6 downto 0) :=  "0010000";
Constant ori : std_logic_vector(6 downto 0) :=  "0010100";
Constant stw : std_logic_vector(6 downto 0) :=  "0010101";
Constant blt : std_logic_vector(6 downto 0) :=  "0010110";
Constant ldw : std_logic_vector(6 downto 0) :=  "0010111";
Constant cmpnei : std_logic_vector(6 downto 0) :=  "0011000";
Constant xori : std_logic_vector(6 downto 0) :=  "0011100";
Constant bne : std_logic_vector(6 downto 0) :=  "0011110";
Constant cmpeqi : std_logic_vector(6 downto 0) :=  "0100000";
Constant ldbuio : std_logic_vector(6 downto 0) :=  "0100011";
Constant muli : std_logic_vector(6 downto 0) :=  "0100100";
Constant stbio : std_logic_vector(6 downto 0) :=  "0100101";
Constant beq : std_logic_vector(6 downto 0) :=  "0100110";
Constant ldbio : std_logic_vector(6 downto 0) :=  "0100111";
Constant cmpgeui : std_logic_vector(6 downto 0) :=  "0101000";
Constant ldhuio : std_logic_vector(6 downto 0) :=  "0101011";
Constant andhi : std_logic_vector(6 downto 0) :=  "0101100";
Constant sthio : std_logic_vector(6 downto 0) :=  "0101101";
Constant bgeu : std_logic_vector(6 downto 0) :=  "0101110";
Constant ldhio : std_logic_vector(6 downto 0) :=  "0101111";
Constant cmpltui : std_logic_vector(6 downto 0) :=  "0110000";
Constant custom : std_logic_vector(6 downto 0) :=  "0110010";
Constant orhi : std_logic_vector(6 downto 0) :=  "0110100";
Constant stwio : std_logic_vector(6 downto 0) :=  "0110101";
Constant bltu : std_logic_vector(6 downto 0) :=  "0110110";
Constant ldwio : std_logic_vector(6 downto 0) :=  "0110111";
Constant rtype : std_logic_vector(6 downto 0) :=  "0111010";
Constant flushd : std_logic_vector(6 downto 0) :=  "0111011";
Constant xorhi : std_logic_vector(6 downto 0) :=  "0111100";


-- Instruction (16 downto 11) in case of rtype (0x3a)
Constant add : std_logic_vector(6 downto 0) := "1110001";
Constant and_rs : std_logic_vector(6 downto 0) := "1001110";
Constant break : std_logic_vector(6 downto 0) := "1110100";
Constant bret : std_logic_vector(6 downto 0) := "1001001";
Constant callr : std_logic_vector(6 downto 0) := "1011101";
Constant cmpeq : std_logic_vector(6 downto 0) := "1100000";
Constant cmpge : std_logic_vector(6 downto 0) := "1001000";
Constant cmpgeu : std_logic_vector(6 downto 0) := "1101000";
Constant cmplt : std_logic_vector(6 downto 0) := "1010000";
Constant cmpltu : std_logic_vector(6 downto 0) := "1110000";
Constant cmpne : std_logic_vector(6 downto 0) := "1011000";
Constant div : std_logic_vector(6 downto 0) := "1100101";
Constant divu : std_logic_vector(6 downto 0) := "1100100";
Constant eret : std_logic_vector(6 downto 0) := "1000001";
Constant flushp : std_logic_vector(6 downto 0) := "1000100";
Constant initd : std_logic_vector(6 downto 0) := "1110011";
```

```
Constant jmp : std_logic_vector(6 downto 0) := "1001101";
Constant mul : std_logic_vector(6 downto 0) := "1100111";
Constant mulxss : std_logic_vector(6 downto 0) := "1011111";
Constant mulxsu : std_logic_vector(6 downto 0) := "1010111";
Constant mulxuu : std_logic_vector(6 downto 0) := "1000111";
Constant nextpc : std_logic_vector(6 downto 0) := "1011100";
Constant nor_rs : std_logic_vector(6 downto 0) := "1000110";
Constant or_rs : std_logic_vector(6 downto 0) := "1010110";
Constant rdctl : std_logic_vector(6 downto 0) := "1100110";
Constant ret : std_logic_vector(6 downto 0) := "1000101";
Constant rotl : std_logic_vector(6 downto 0) := "1000011";
Constant roli : std_logic_vector(6 downto 0) := "1000010";
Constant rotr : std_logic_vector(6 downto 0) := "1001011";
Constant shll : std_logic_vector(6 downto 0) := "1010011";
Constant slli : std_logic_vector(6 downto 0) := "1010010";
Constant shra : std_logic_vector(6 downto 0) := "1111011";
Constant srai : std_logic_vector(6 downto 0) := "1111010";
Constant shrl : std_logic_vector(6 downto 0) := "1011011";
Constant srli : std_logic_vector(6 downto 0) := "1011010";
Constant sub : std_logic_vector(6 downto 0) := "1111001";
Constant sync : std_logic_vector(6 downto 0) := "1110110";
Constant trap : std_logic_vector(6 downto 0) := "1101101";
Constant wrctl : std_logic_vector(6 downto 0) := "1101110";
Constant xor_rs : std_logic_vector(6 downto 0) := "1011110";

-- Signal constants
Constant and_op : std_logic_vector := "00";
Constant or_op : std_logic_vector := "01";
Constant nor_op : std_logic_vector := "10";
Constant xor_op : std_logic_vector := "11";
Constant alu_op : std_logic := '0';
Constant mem_op : std_logic := '1';
Constant left : std_logic := '0';
Constant right : std_logic := '1';
Constant add_op : std_logic := '0';
Constant sub_op : std_logic := '1';
Constant is_signed : std_logic := '0';
Constant is_unsigned : std_logic := '1';
Constant lo : std_logic := '0';
Constant hi : std_logic := '1';
Constant immediate : std_logic := '1';
Constant immed_5 : std_logic := '1';
Constant immed_16 : std_logic := '0';
Constant eq : std_logic_vector(2 downto 0) := "001";
Constant ne : std_logic_vector(2 downto 0) := "010";
Constant lt : std_logic_vector(2 downto 0) := "011";
Constant gt : std_logic_vector(2 downto 0) := "100";
Constant lte : std_logic_vector(2 downto 0) := "101";
Constant gte : std_logic_vector(2 downto 0) := "110";
Constant is_branch : std_logic := '1';
Constant rotation : std_logic_vector(1 downto 0) := "00";
Constant logical : std_logic_vector(1 downto 0) := "01";
Constant arithmetic : std_logic_vector(1 downto 0) := "10";
Constant adder_op : std_logic_vector(1 downto 0) := "00";
Constant mult_op : std_logic_vector(1 downto 0) := "01";
```

```
    Constant logical_op : std_logic_vector(1 downto 0) := "10";
    Constant shifter_op : std_logic_vector(1 downto 0) := "11";
    Constant call_op : std_logic := '1';
    Constant ret_op : std_logic := '1';
    Constant write : std_logic := '1';
    Constant read : std_logic := '1';
    Constant lane_0 : std_logic_vector(1 downto 0) := "00";
    Constant lane_1 : std_logic_vector(1 downto 0) := "01";
    Constant lane_3 : std_logic_vector(1 downto 0) := "10";
    Constant x1 : std_logic := '0';
    Constant x2 : std_logic_vector(1 downto 0) := "00";
    Constant x3 : std_logic_vector(2 downto 0) := "000";

    -- Instruction Stages
    Constant fetch : std_logic_vector(5 downto 0) := "000001";
    Constant operand : std_logic_vector(5 downto 0) := "000010";
    Constant execute1 : std_logic_vector(5 downto 0) := "000100";
    Constant execute2 : std_logic_vector(5 downto 0) := "001000";
    Constant writeback : std_logic_vector(5 downto 0) := "010000";
  -- Non hierarchical truthtable declarations


  -- Non hierarchical truthtable declarations


  -- Non hierarchical truthtable declarations


  -- Non hierarchical truthtable declarations



-- Internal signal declarations
SIGNAL addr1           : std_logic_vector(31 DOWNTO 0);
SIGNAL addr2           : std_logic_vector(31 DOWNTO 0);
SIGNAL alu_or_mem      : std_logic;
SIGNAL and_opcode      : std_logic_vector(5 DOWNTO 0);
SIGNAL branch_check    : std_logic;
SIGNAL byte_lane       : std_logic_vector(1 DOWNTO 0);
SIGNAL call_func       : std_logic;
SIGNAL callr_func      : std_logic;
SIGNAL ex1             : std_logic_vector(16 DOWNTO 0);
SIGNAL ex1_1           : std_logic_vector(16 DOWNTO 0);
SIGNAL ex1_add_sub     : std_logic;
SIGNAL ex1_cmp         : std_logic_vector(2 DOWNTO 0);
SIGNAL ex1_immed       : std_logic;
SIGNAL ex2             : std_logic_vector(14 DOWNTO 0);
SIGNAL ex2_op_sel      : std_logic_vector(1 DOWNTO 0);
SIGNAL flush           : std_logic;
SIGNAL flush_call      : std_logic;
SIGNAL flush_callr     : std_logic;
SIGNAL flush_hi        : std_logic;
SIGNAL flush_jmp       : std_logic;
SIGNAL flush_regout    : std_logic;
SIGNAL flush_ret       : std_logic;
```

```
SIGNAL is_rtype         : std_logic;
SIGNAL jmp_func         : std_logic;
SIGNAL left_right       : std_logic;
SIGNAL lo_hi            : std_logic;
SIGNAL logic_op         : std_logic_vector(1 DOWNTO 0);
SIGNAL op               : std_logic_vector(4 DOWNTO 0);
SIGNAL op_NOT_3a        : std_logic_vector(5 DOWNTO 0);
SIGNAL op_immed         : std_logic;
SIGNAL op_immed_16_or_5 : std_logic;
SIGNAL op_immed_26      : std_logic;
SIGNAL op_lo_hi         : std_logic;
SIGNAL op_signed_op     : std_logic;
SIGNAL opcode_0         : std_logic_vector(6 DOWNTO 0);
SIGNAL opcode_1         : std_logic_vector(6 DOWNTO 0);
SIGNAL opcode_2         : std_logic_vector(6 DOWNTO 0);
SIGNAL opcode_3         : std_logic_vector(6 DOWNTO 0);
SIGNAL return_func      : std_logic;
SIGNAL rot_log_arith    : std_logic_vector(1 DOWNTO 0);
SIGNAL signed_op        : std_logic;
SIGNAL signed_op2       : std_logic;
SIGNAL wb               : std_logic_vector(1 DOWNTO 0);
SIGNAL wr_mem           : std_logic;
SIGNAL wr_reg           : std_logic;

-- Implicit buffer signal declarations
SIGNAL flush_lo_internal : std_logic;


-- Component Declarations
COMPONENT Generic_Reg_noenable
GENERIC (
  SizeIn  : integer;
  SizeOut : integer
);
PORT (
  A_in  : IN     std_logic_vector (SizeIn - 1 DOWNTO 0);
  clock : IN     std_logic;
  reset : IN     std_logic;
  A_out : OUT    std_logic_vector (SizeOut - 1 DOWNTO 0)
);
END COMPONENT;
COMPONENT Reg_1
PORT (
  clock : IN     std_logic ;
  d_in  : IN     std_logic ;
  reset : IN     std_logic ;
  q_out : OUT    std_logic
);
END COMPONENT;
COMPONENT Reg_32_noenable
PORT (
  clock   : IN     std_logic ;
  reg_in  : IN     std_logic_vector (31 DOWNTO 0);
  reset   : IN     std_logic ;
  reg_out : OUT    std_logic_vector (31 DOWNTO 0)
```

```
  );
  END COMPONENT;

  -- Optional embedded configurations
  -- pragma synthesis_off
  FOR ALL : Generic_Reg_noenable USE ENTITY Auto_Gen.Generic_Reg_noenable;
  FOR ALL : Reg_1 USE ENTITY Auto_Gen.Reg_1;
  FOR ALL : Reg_32_noenable USE ENTITY Auto_Gen.Reg_32_noenable;
  -- pragma synthesis_on


BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Text Block 1 eb1
  process(clock, flush)
  begin
  if (flush = '1') then
    ctl_op <= (others => '0');
  elsif (rising_edge(clock)) then
    ctl_op <= op;
  end if;
  end process;

  -- HDL Embedded Block 2 decoder_op
  -- Non hierarchical truthtable
  ---------------------------------------------------------------------------
  decoder_op_truth_process: PROCESS(opcode_0)
  ---------------------------------------------------------------------------
  BEGIN
    -- Block 1
    CASE opcode_0 IS
    WHEN call =>
      op_lo_hi <= x1;
      op_immed_26 <= immediate;
      op_immed_16_or_5 <= x1;
      op_immed <= x1;
      op_signed_op <= x1;
    WHEN ldbu =>
      op_lo_hi <= x1;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= immed_16;
      op_immed <= immediate;
      op_signed_op <= is_unsigned;
    WHEN addi =>
      op_lo_hi <= x1;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= immed_16;
      op_immed <= immediate;
      op_signed_op <= is_signed;
    WHEN stb =>
      op_lo_hi <= x1;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= immed_16;
      op_immed <= immediate;
      op_signed_op <= is_signed;
```

```
WHEN br =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN ldb =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN cmpgei =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN ldhu =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_unsigned;
WHEN andi =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN sth =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN bge =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN ldh =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN cmplti =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN ori =>
```

```
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= x1;
WHEN stw =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN blt =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN ldw =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN cmpnei =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN xori =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= x1;
WHEN bne =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN cmpeqi =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN ldbuio =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_16;
  op_immed <= immediate;
  op_signed_op <= is_unsigned;
WHEN muli =>
  op_lo_hi <= x1;
```

```
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_signed;
WHEN stbio =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_signed;
WHEN beq =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= x1;
   op_signed_op <= is_signed;
WHEN ldbio =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_signed;
WHEN cmpgeui =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_unsigned;
WHEN ldhuio =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_unsigned;
WHEN andhi =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_signed;
WHEN sthio =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= immed_16;
   op_immed <= immediate;
   op_signed_op <= is_signed;
WHEN bgeu =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
   op_immed_16_or_5 <= x1;
   op_immed <= x1;
   op_signed_op <= is_unsigned;
WHEN ldhio =>
   op_lo_hi <= x1;
   op_immed_26 <= x1;
```

```
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_signed;
  WHEN cmpltui =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_16;
    op_immed <= immediate;
    op_signed_op <= is_unsigned;
  WHEN custom =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= x1;
  WHEN initd =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= x1;
  WHEN orhi =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_16;
    op_immed <= immediate;
    op_signed_op <= is_signed;
  WHEN stwio =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_16;
    op_immed <= immediate;
    op_signed_op <= is_signed;
  WHEN bltu =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_unsigned;
  WHEN ldwio =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_16;
    op_immed <= immediate;
    op_signed_op <= is_signed;
  WHEN add =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_signed;
  WHEN and_rs =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
```

```
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN break =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN bret =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN callr =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN cmpeq =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN cmpge =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN cmpgeu =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_unsigned;
WHEN cmplt =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN cmpltu =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_unsigned;
WHEN cmpne =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
```

```
  op_signed_op <= is_signed;
WHEN div =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN divu =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_unsigned;
WHEN eret =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN flushp =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN jmp =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN mul =>
  op_lo_hi <= lo;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN mulxss =>
  op_lo_hi <= hi;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN mulxsu =>
  op_lo_hi <= hi;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN mulxuu =>
  op_lo_hi <= hi;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_unsigned;
```

```
WHEN nextpc =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN nor_rs =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN or_rs =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN rdctl =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN ret =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= x1;
WHEN rotl =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN roli =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= immed_5;
  op_immed <= immediate;
  op_signed_op <= is_signed;
WHEN rotr =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN shll =>
  op_lo_hi <= x1;
  op_immed_26 <= x1;
  op_immed_16_or_5 <= x1;
  op_immed <= x1;
  op_signed_op <= is_signed;
WHEN slli =>
```

```
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_5;
    op_immed <= immediate;
    op_signed_op <= is_unsigned;
  WHEN shra =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_signed;
  WHEN srai =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_5;
    op_immed <= immediate;
    op_signed_op <= is_unsigned;
  WHEN shrl =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_signed;
  WHEN srli =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= immed_5;
    op_immed <= immediate;
    op_signed_op <= is_unsigned;
  WHEN sub =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= is_signed;
  WHEN sync =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= x1;
  WHEN trap =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= x1;
  WHEN wrctl =>
    op_lo_hi <= x1;
    op_immed_26 <= x1;
    op_immed_16_or_5 <= x1;
    op_immed <= x1;
    op_signed_op <= x1;
  WHEN flushd =>
    op_lo_hi <= lo;
```

```
      op_immed_26 <= x1;
      op_immed_16_or_5 <= x1;
      op_immed <= x1;
      op_signed_op <= x1;
    WHEN xor_rs =>
      op_lo_hi <= x1;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= x1;
      op_immed <= x1;
      op_signed_op <= is_signed;
    WHEN xorhi =>
      op_lo_hi <= x1;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= immed_16;
      op_immed <= immediate;
      op_signed_op <= is_signed;
    WHEN OTHERS =>
      op_lo_hi <= lo;
      op_immed_26 <= x1;
      op_immed_16_or_5 <= x1;
      op_immed <= x1;
      op_signed_op <= x1;
    END CASE;

  END PROCESS decoder_op_truth_process;

  -- HDL Embedded Text Block 3 eb2
  --ex1(0) <= ex1_immed;
  --ex1(1) <= ex1_add_sub;
  --ex1(3 downto 2) <= rot_log_arith;
  --ex1(4) <= signed_op;
  --ex1(5) <= signed_op2;
  --ex1(7 downto 6) <= logic_op;
  --ex1(8) <= lo_hi;
  --ex1(11 downto 9) <= ex1_cmp;
  --ex1(12) <= call_func;
  --ex1(13) <= callr_func;
  --ex1(14) <= jmp_func;
  --ex1(15) <= return_func;
  --ex1(16) <= left_right;
  ex1 <= left_right & return_func & jmp_func & callr_func & call_func & ex1_cmp & lo_hi & logic_op &
signed_op2 & signed_op & rot_log_arith & ex1_add_sub & ex1_immed;


  --op(0) <= ex1_immed;
  --op(1) <= op_immed_16_or_5;
  --op(2) <= immed_26;
  --op(3) <= lo_hi_op;
  --op(4) <= signed_op;
  op <= op_signed_op & op_lo_hi & op_immed_26 & op_immed_16_or_5 & op_immed;

  --ex2(1 downto 0) <= ex2_op_sel;
  --ex2(2) <= branch_check;
  --ex2(3) <= wr_mem;
  --ex2(9 downto 4) <= opcode;
```

```
--ex2(12 downto 10) <= other(2 downto 0);
--ex2(14 downto 13) <= byte_lane;
--ex2(15) <= rd_mem;
ex2 <= byte_lane & other(2 downto 0) & opcode & wr_mem & branch_check & ex2_op_sel;

--wb(0) <= alu_or_mem;
--wb(1) <= wr_reg;
wb <= wr_reg & alu_or_mem;

-- HDL Embedded Text Block 4 eb3
process(clock, reset)
begin
if (reset = '1') then
  ctl_ex1 <= (others => '0');
--elsif (flush = '1') then
  --ex1_1 <= (others => '0');
  --ex1_1(15 downto 12) <= ex1(15 downto 12);
elsif (rising_edge(clock)) then
  ctl_ex1 <= ex1;
end if;
end process;

-- HDL Embedded Text Block 6 eb5
process(clock, reset)
begin
if (reset = '1') then
  ctl_ex2 <= (others => '0');
elsif (rising_edge(clock)) then
  ctl_ex2 <= ex2;
end if;
end process;

-- HDL Embedded Text Block 9 eb8
process(clock, reset)
begin
if (reset = '1') then
  ctl_wb <= (others => '0');
elsif (rising_edge(clock)) then
  ctl_wb <= wb;
end if;
end process;

-- HDL Embedded Text Block 13 eb12
op_NOT_3a <= "000101";
process(is_rtype, opcode, other)
begin
  if (is_rtype = '1') then
    opcode_0 <= is_rtype & other;
  else
    opcode_0 <= is_rtype & opcode;
  end if;
end process;

-- HDL Embedded Text Block 14 eb13
flush_ret <= ex1_1(15);
```

```
flush_jmp <= ex1_1(14);
flush_callr <= ex1_1(13);
flush_call <= ex1_1(12);

-- HDL Embedded Block 15 decoder_ex1
-- Non hierarchical truthtable
-------------------------------------------------------------------------
decoder_ex1_truth_process: PROCESS(flush_hi, opcode_1)
-------------------------------------------------------------------------
BEGIN
  -- Block 1
  IF (flush_hi = '1') THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = call) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= call_op;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldbu) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = addi) THEN
    ex1_add_sub <= add_op;
```

```
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = stb) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = br) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldb) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpgei) THEN
```

```
    ex1_add_sub <= sub_op;
    ex1_cmp <= gte;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldhu) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = andi) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= and_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = sth) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
```

```
ELSIF (opcode_1 = bge) THEN
  ex1_add_sub <= sub_op;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= x1;
  lo_hi <= x1;
  rot_log_arith <= x2;
  ex1_immed <= x1;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = ldh) THEN
  ex1_add_sub <= add_op;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= x1;
  lo_hi <= x1;
  rot_log_arith <= x2;
  ex1_immed <= immediate;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = cmplti) THEN
  ex1_add_sub <= sub_op;
  ex1_cmp <= lt;
  logic_op <= x2;
  left_right <= x1;
  lo_hi <= x1;
  rot_log_arith <= x2;
  ex1_immed <= immediate;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = ori) THEN
  ex1_add_sub <= x1;
  ex1_cmp <= x3;
  logic_op <= or_op;
  left_right <= x1;
  lo_hi <= x1;
  rot_log_arith <= x2;
  ex1_immed <= immediate;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= x1;
```

```
    signed_op2 <= x1;
  ELSIF (opcode_1 = stw) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = blt) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldw) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpnei) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= ne;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
```

```
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = xori) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= or_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = bne) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpeqi) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= eq;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldbuio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
```

```
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = muli) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = stbio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = beq) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldbio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
```

```
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpgeui) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= gte;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldhuio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = andhi) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= and_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = sthio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
```

```
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = bgeu) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldhio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpltui) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= lt;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = custom) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
```

```
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = initd) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = orhi) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= or_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = stwio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = bltu) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
```

```
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ldwio) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = add) THEN
    ex1_add_sub <= add_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = and_rs) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= and_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = break) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
```

```
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = bret) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = callr) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= call_op;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpeq) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= eq;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpge) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= gte;
    logic_op <= x2;
    left_right <= x1;
```

```
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpgeu) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= gte;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmplt) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= lt;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpltu) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= lt;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = cmpne) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= ne;
    logic_op <= x2;
```

```
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = div) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = divu) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = eret) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = flushp) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
```

```
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = jmp) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= call_op;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = mul) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= lo;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= is_signed;
  ELSIF (opcode_1 = mulxss) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= hi;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= is_signed;
  ELSIF (opcode_1 = mulxsu) THEN
    ex1_add_sub <= x1;
```

```
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= hi;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= is_unsigned;
  ELSIF (opcode_1 = mulxuu) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= hi;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= is_unsigned;
  ELSIF (opcode_1 = nextpc) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = nor_rs) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= nor_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= is_signed;
  ELSIF (opcode_1 = or_rs) THEN
```

```
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= or_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= is_signed;
  ELSIF (opcode_1 = rdctl) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = ret) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= ret_op;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = rotl) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= left;
    lo_hi <= x1;
    rot_log_arith <= rotation;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
```

```
ELSIF (opcode_1 = roli) THEN
  ex1_add_sub <= x1;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= left;
  lo_hi <= x1;
  rot_log_arith <= rotation;
  ex1_immed <= immediate;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = rotr) THEN
  ex1_add_sub <= x1;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= right;
  lo_hi <= x1;
  rot_log_arith <= rotation;
  ex1_immed <= x1;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = shll) THEN
  ex1_add_sub <= x1;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= left;
  lo_hi <= x1;
  rot_log_arith <= logical;
  ex1_immed <= x1;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_signed;
  signed_op2 <= x1;
ELSIF (opcode_1 = slli) THEN
  ex1_add_sub <= x1;
  ex1_cmp <= x3;
  logic_op <= x2;
  left_right <= left;
  lo_hi <= x1;
  rot_log_arith <= logical;
  ex1_immed <= immediate;
  call_func <= x1;
  callr_func <= x1;
  jmp_func <= x1;
  return_func <= x1;
  signed_op <= is_unsigned;
```

```
    signed_op2 <= x1;
  ELSIF (opcode_1 = shra) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= right;
    lo_hi <= x1;
    rot_log_arith <= arithmetic;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = srai) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= right;
    lo_hi <= x1;
    rot_log_arith <= arithmetic;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = shrl) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= right;
    lo_hi <= x1;
    rot_log_arith <= logical;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = srli) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= right;
    lo_hi <= x1;
    rot_log_arith <= logical;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
```

```
    signed_op <= is_unsigned;
    signed_op2 <= x1;
  ELSIF (opcode_1 = sub) THEN
    ex1_add_sub <= sub_op;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = sync) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = trap) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = wrctl) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
```

```
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = flushd) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= lo;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  ELSIF (opcode_1 = xor_rs) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= xor_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSIF (opcode_1 = xorhi) THEN
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= xor_op;
    left_right <= x1;
    lo_hi <= x1;
    rot_log_arith <= x2;
    ex1_immed <= immediate;
    call_func <= x1;
    callr_func <= x1;
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= is_signed;
    signed_op2 <= x1;
  ELSE
    ex1_add_sub <= x1;
    ex1_cmp <= x3;
    logic_op <= x2;
    left_right <= x1;
    lo_hi <= lo;
    rot_log_arith <= x2;
    ex1_immed <= x1;
    call_func <= x1;
    callr_func <= x1;
```

```
    jmp_func <= x1;
    return_func <= x1;
    signed_op <= x1;
    signed_op2 <= x1;
  END IF;

END PROCESS decoder_ex1_truth_process;

-------------------------------------------------------------------------
decoder_ex2_truth_process: PROCESS(opcode_2)
-------------------------------------------------------------------------
BEGIN
  -- Block 1
  CASE opcode_2 IS
  WHEN call =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldbu =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_0;
  WHEN addi =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN stb =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= write;
    byte_lane <= lane_0;
  WHEN br =>
    ex2_op_sel <= adder_op;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldb =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_0;
  WHEN cmpgei =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldhu =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_1;
  WHEN andi =>
```

```
   ex2_op_sel <= logical_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN sth =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= write;
   byte_lane <= lane_1;
WHEN bge =>
   ex2_op_sel <= adder_op;
   branch_check <= is_branch;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN ldh =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= lane_1;
WHEN cmplti =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN ori =>
   ex2_op_sel <= logical_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN stw =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= write;
   byte_lane <= lane_3;
WHEN blt =>
   ex2_op_sel <= adder_op;
   branch_check <= is_branch;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN ldw =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= lane_3;
WHEN cmpnei =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN xori =>
   ex2_op_sel <= logical_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN bne =>
```

```
    ex2_op_sel <= adder_op;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpeqi =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldbuio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_1;
  WHEN muli =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN stbio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= write;
    byte_lane <= lane_0;
  WHEN beq =>
    ex2_op_sel <= adder_op;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldbio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_0;
  WHEN cmpgeui =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldhuio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_1;
  WHEN andhi =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN sthio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= write;
    byte_lane <= lane_1;
  WHEN bgeu =>
```

```
    ex2_op_sel <= adder_op;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldhio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_1;
  WHEN cmpltui =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN custom =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN initd =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN orhi =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN stwio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= write;
    byte_lane <= lane_3;
  WHEN bltu =>
    ex2_op_sel <= adder_op;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ldwio =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= lane_3;
  WHEN add =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN and_rs =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN break =>
```

```
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN bret =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN callr =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpeq =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpge =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpgeu =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmplt =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpltu =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN cmpne =>
    ex2_op_sel <= adder_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN div =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN divu =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN eret =>
```

```
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN flushp =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN jmp =>
    ex2_op_sel <= x2;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN mul =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN mulxss =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN mulxsu =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN mulxuu =>
    ex2_op_sel <= mult_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN nextpc =>
    ex2_op_sel <= x2;
    branch_check <= is_branch;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN nor_rs =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN or_rs =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN rdctl =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN ret =>
```

```
   ex2_op_sel <= x2;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN rotl =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN roli =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN rotr =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN shll =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN slli =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN shra =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN srai =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN shrl =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN srli =>
   ex2_op_sel <= shifter_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN sub =>
   ex2_op_sel <= adder_op;
   branch_check <= x1;
   wr_mem <= x1;
   byte_lane <= x2;
WHEN sync =>
```

```
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN trap =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN wrctl =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN flushd =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN xor_rs =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN xorhi =>
    ex2_op_sel <= logical_op;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  WHEN OTHERS =>
    ex2_op_sel <= x2;
    branch_check <= x1;
    wr_mem <= x1;
    byte_lane <= x2;
  END CASE;

END PROCESS decoder_ex2_truth_process;

decoder_wb_truth_process: PROCESS(opcode_3)
-------------------------------------------------------------------------------
BEGIN
  -- Block 1
  CASE opcode_3 IS
  WHEN call =>
    alu_or_mem <= x1;
    wr_reg <= x1;
  WHEN ldbu =>
    alu_or_mem <= mem_op;
    wr_reg <= write;
  WHEN addi =>
    alu_or_mem <= alu_op;
    wr_reg <= write;
  WHEN stb =>
    alu_or_mem <= x1;
    wr_reg <= x1;
```

```
WHEN br =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldb =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN cmpgei =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN ldhu =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN andi =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN sth =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN bge =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldh =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN cmplti =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN ori =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN stw =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN blt =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldw =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN cmpnei =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN xori =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN bne =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN cmpeqi =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN ldbuio =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN muli =>
```

```
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN stbio =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN beq =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldbio =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN cmpgeui =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN ldhuio =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN andhi =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN sthio =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN bgeu =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldhio =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN cmpltui =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN custom =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN initd =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN orhi =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN stwio =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN bltu =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ldwio =>
  alu_or_mem <= mem_op;
  wr_reg <= write;
WHEN add =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN and_rs =>
  alu_or_mem <= alu_op;
```

```
  wr_reg <= write;
WHEN break =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN bret =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN callr =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN cmpeq =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN cmpge =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN cmpgeu =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN cmplt =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN cmpltu =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN cmpne =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN div =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN divu =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN eret =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN flushp =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN jmp =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN mul =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN mulxss =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN mulxsu =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN mulxuu =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
```

```
WHEN nextpc =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN nor_rs =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN or_rs =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN rdctl =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN ret =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN rotl =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN roli =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN rotr =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN shll =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN slli =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN shra =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN srai =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN shrl =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN srli =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN sub =>
  alu_or_mem <= alu_op;
  wr_reg <= write;
WHEN sync =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN trap =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN wrctl =>
  alu_or_mem <= x1;
  wr_reg <= x1;
WHEN flushd =>
```

```
    alu_or_mem <= x1;
    wr_reg <= x1;
  WHEN xor_rs =>
    alu_or_mem <= alu_op;
    wr_reg <= write;
  WHEN xorhi =>
    alu_or_mem <= alu_op;
    wr_reg <= write;
  WHEN OTHERS =>
    alu_or_mem <= x1;
    wr_reg <= x1;
  END CASE;

END PROCESS decoder_wb_truth_process;

-- ModuleWare code(v1.0) for instance 'I4' of 'or'
flush_hi <= reset OR take_branch;

-- ModuleWare code(v1.0) for instance 'I8' of 'or'
flush_lo_internal <= flush_hi OR flush_ret OR flush_jmp OR flush_callr
                     OR flush_call;

-- ModuleWare code(v1.0) for instance 'I11' of 'or'
flush <= flush_lo_internal OR flush_regout;

-- ModuleWare code(v1.0) for instance 'I1' of 'tand'
I1combo: PROCESS (and_opcode)
VARIABLE dtemp : std_logic;
VARIABLE itemp : std_logic_vector(5 DOWNTO 0);
BEGIN
  itemp := and_opcode;
  dtemp := '1';
  FOR i IN 5 DOWNTO 0 LOOP
    dtemp := dtemp AND itemp(i);
  END LOOP;
  is_rtype <= dtemp;
END PROCESS I1combo;

-- ModuleWare code(v1.0) for instance 'I10' of 'xor'
and_opcode <= opcode XOR op_NOT_3a;

-- Instance port mappings.
I0 : Generic_Reg_noenable
  GENERIC MAP (
    SizeIn  => 7,
    SizeOut => 7
  )
  PORT MAP (
    clock => clock,
    reset => reset,
    A_in  => opcode_0,
    A_out => opcode_1
  );
I2 : Generic_Reg_noenable
  GENERIC MAP (
```

```
      SizeIn  => 7,
      SizeOut => 7
    )
    PORT MAP (
      clock => clock,
      reset => reset,
      A_in  => opcode_1,
      A_out => opcode_2
    );
  I3 : Generic_Reg_noenable
    GENERIC MAP (
      SizeIn  => 7,
      SizeOut => 7
    )
    PORT MAP (
      clock => clock,
      reset => reset,
      A_in  => opcode_2,
      A_out => opcode_3
    );
  I9 : Reg_1
    PORT MAP (
      clock => clock,
      d_in  => flush_lo_internal,
      reset => reset,
      q_out => flush_regout
    );
  I5 : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => addr1,
      reset   => flush_lo_internal,
      reg_out => addr2
    );
  I6 : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => addr,
      reset   => flush_lo_internal,
      reg_out => addr1
    );
  I7 : Reg_32_noenable
    PORT MAP (
      clock   => clock,
      reg_in  => addr2,
      reset   => reset,
      reg_out => base_addr
    );

  -- Implicit buffered output assignments
  flush_lo <= flush_lo_internal;

END struct;
```

BIBLIOGRAPHY

1. W.K. Jenkins, "A Highly Efficient Residue-Combinatorial Architecture for Digital Filters," *Proceedings of the IEEE,* v.66, n.6, pp. 700-2, June 1978.

2. A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, K.L. Kranz, J.D. Kubiatowicz, B.H. Lim, K. Mackenzie and D. Yeung, "The MIT Alewife Machine," *Proceedings of the IEEE,* v.87, n.3, pp. 430-44, March 1999.

3. M.B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffman, P. Johnson and J. Kim, "Evaluation of the Raw Microprocessor: An Exposed Wire-delay Architecture for ILP and Streams," *Proceedings of the 2004 IEEE Symposium on Computer Architecture*, pp. 2-13, June 2004.

4. D. Sima, "Decisive Aspects in the Evolution of Microprocessors," *Proceedings of the IEEE,* v.92 n.12, pp. 1896-1926, Dec. 2004.

5. K.Y. Tong, V. Kheterpal, V. Rovner, L. Pileggi and H. Schmit, "Regular Logic Fabrics for a Via Patterned Gate Array," *Proceedings of the 2003 IEEE International Conference on Custom Integrated Circuits*, pp. 53-56, Sept. 2003.

6. H. Qi, A. Jiang and J. Wei, "IP Reusable Design Methodology," *Proceedings of the 2001 IEEE International Conference on ASIC,* pp. 756-759, Oct. 2001.

7. J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12-21, Napa Valley, CA, April 1997.

8. T.J. Callahan, J.R. Hauser and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Computer*, pp. 62-69, Vol. 33 No. 4, April 2000.

9. X. Wang and S.G. Ziavras, "A Configurable Multiprocessor and Dynamic Load Balancing for Parallel LU Factorization," *Proceedings of the 2004 IEEE Symposium on Parallel and Distributed Processing,* April 2004.

10. R.R. Hoare and S.C. Tung, “Combining Mentor Graphics’ HDL Designer FPGA Flow with a Reconfigurable System on a Programmable Chip, Educational Opportunity or Insanity?” *Proceedings of the 2003 IEEE International Conference on Microelectronic Systems Education*, June 2003.

11. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd Ed.*, Morgan-Kauffman, 1996.

12. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kauffman, 1996.

13. Altera Inc. Nios II Embedded Processor Literature: Nios II Instruction Set Reference, *http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf*.

14. M. Berekovic, P. Pirsch, T. Selinger, K.I. Wels, C. Miro, A. Lafage, C. Heer and G. Ghigo, “Co-processor Architecture for MPEG-4 Main Profile Visual Compositing,” *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems*, pp. 180-183, Geneva, Switzerland, May 2000.

15. D. Rizzo and O. Colavin, “A Video Compression Case Study on a Reconfigurable VLIW Architecture,” *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pp. 540-546, Paris, France, March 2002.

16. C. Kozyrakis and D. Patterson, “Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks,” *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 283-294, Istanbul, Turkey, November 2002.

17. P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Homewood, “Lx: A Technology Platform for Customizable VLIW Embedded Processing,” *Proceedings if the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

18. D. Kusic, R. Hoare, A.K. Jones, J. Fazekas and J. Foster, "Extracting Speedup From C-code With Poor Instruction-level Parallelism," Intl. Parallel and Distributed Processing Sym. (IPDPS), *Workshop on Massively Parallel Processing (WMPP)*, Denver, CO, April 2005.

19. S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi and R.R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler,” *IEEE Computer*, pp. 70-77, Vol. 33. No. 4, April 2000.

20. S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor and R. Laufer, “PipeRench: A Coprocessor for Streaming Multimedia Acceleration,” *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 28-39, Atlanta, GA, May 1999.

21. B. A. Levine, H. Schmit, “Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable,” *FCCM*, 2003.

22. D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. "Architecture Design of Reconfigurable Pipelined Datapaths," *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.

23. E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1996.

24. U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, B. Khailany, "The Imagine Stream Processor," IEEE *International Conference on Computer Design: VLSI in Computers and Processors*, Sept. 2002.

25. John D. Owens, Scott Rixner, Ujval Kapasi, Peter Mattson, Brian Towles, and William J. Dally, "Media Processing Applications on the Imagine Stream Processor," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Sept. 2002.

26. S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 87-96.

27. R. Hoare, S. Tung, K. Werger, "A 64-Way SIMD Processing Architecture on an FPGA," *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2003, pp. 345-350.

28. S. Dutta, A. Wolfe, W. Wolf and K. O'Connor, "Design Issues for Very-Long-Instruction-Word VLSI Video Signal Processors," *IEEE Workshop on VLSI Signal Processing*, San Francisco, Oct. 1996.

29. Altera Inc. Stratix II Device Family Data Sheet: Stratix II Device Handbook, *http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf*.

30. Trimaran Compiler: Web-based information, *http://www.trimaran.org/*.

31. Shark Code Profiler: Apple Developer's Connection, *http://developer.apple.com/tools/sharkoptimize.html*.

32. 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," *Int. Symposium on Field Programmable Gate Arrays (FPGA'05)*, Monterey, CA, Feb. 2005.

33. 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 J. of Applied Signal Processing*, 2005.

34. P.J. Ashenden, "Modeling Digital Systems Using VHDL," *IEEE Potentials*, v.17 n.2, pp. 27-30, Apr.-May 1998.

35. R. Waxman, J.H. Aylor and E. Marschner, "The VHSIC Hardware Description Language (IEEE Standard 1076)," *Digest of Papers from the IEEE Computer Society International Conference*, pp. 310-315, March 1998.

36. D.E. Culler, J.P. Singh and A. Gupta, *Parallel Computer Architecture,* San Francisco: Morgan Kaufmann Publishers, Inc, 1999. pp. 28-37, pp. 453-467.

37. D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, G. Stitt, "Profiling Tools for Hardware/Software Partitioning of Embedded Applications," *Proceedings of the 2003 ACM SiGPLAN Conference on Languages, Compilers and Tools for Embedded Systems*, San Diego, CA, June 2003.

38. Open Source Initiative, *http://www.systemc.org*.

39. A. Bona, V. Zaccaria and R. Zafalon, "System Level Power Modeling and Simulation of High-end Instrustial Network-on-chip," *Proceedings of the 2004 IEEE Design, Automation Test in Europe Conference and Exhibition*, pp. 318-323, Feb. 2004.

40. U. Neffe, K. Rothbart, C. Steger, R. Weiss, E. Rieger and A. Muhlberger, "Energy Estimation Based on Hierarchical Bus Models for Power-aware Smart Cards," *Proceedings of the 2004 IEEE Design, Automation and Test in Europe Conference and Exhibition*, pp. 300 - 305, Feb. 2004.

41. C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri and M. Bocchi, "A Cycle-Accurate ISS for a Dynamically Reconfigurable Processor Architecture," *Proceedings of the 2005 IEEE Parallel and Distributed Processing Symposium*, Apr. 2004.

42. ARM Processor Core summary page, http://www.arm.com/products/CPUs.