2024

STATICALLY CONTROLLED SYNCHRONIZED LANE ARCHITECTURES

Scott K. Pomerville
Michigan Technological University, skpomerv@mtu.edu

STATICALLY CONTROLLED SYNCHRONIZED LANE ARCHITECTURES

By

Scott K. Pomerville

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2024

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Soner Önder*
Committee Member:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Jianhui Yue*
Committee Member:	*Dr. David Whalley*
Department Chair:	*Dr. Zhenlin Wang*

Dedication

To my Grandmother Janyth,

who helped inspire my love for learning. I wish you could have been here for this.

Contents

List of Figures

List of Tables

Author Contribution Statement

Small parts of the chapter 3 were published in 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2023). I was second author for this paper. Any parts of this paper included in this work were contributions to the paper written by me, and parts not written by me are not included in this work.

Parts of chapter 4 and chapter 5 were submitted to the 31st IEEE International Symposium on High-Performance Computer Architecture (HPCA 2025). I am the first author of the submitted article. I contributed the design ideas, designed the experiments and simulators, and am responsible for the evaluation of the results.

Acknowledgments

A special thanks to the people below.

My Mom and Dad, whose support has been unwavering and who were always there when I needed guidance in my life. Their love and support gave me what I needed to go forward knowing I could give this degree my all.

Amanda Erdmann, who kept me grounded when I was absorbed in work, who remained patient even when I worked late, and who would cover for all those chores that I would forget when I was near a deadline.

Dr. Gang-Ryung Uh and his late-night impromptu meetings to work on the code generation, and without whom I would not have had any code to run on my simulators.

Dr. Soner Önder's constant guidance as an advisor, his drive for excellence, and his ability to tell me what I needed to hear (even when I often didn't want to hear it). Sorry about all of the terrible jokes I wrote on the lab whiteboard.

Finally, a deeply heartfelt thanks **to all of my friends, mentors, and colleagues** who supported me throughout my academic journey. This work is a product of their collective support.

This work is supported in part by the US National Science Foundation (NSF) under grants 1901005 and 1900788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF.

Abstract

Modern superscalar processors dominate the field of computing. While dynamic execution allows for versatility in code, these processors are complex. Statically scheduled code has historically enabled simpler processor designs, but static scheduling cannot account for variables that are unknown at compile time. Furthermore, static scheduling has many inefficiencies, such as the need to insert a large number of *nops* for code in traditional Very Long Instruction Word (VLIW) processors. In this dissertation, we explore a novel architectural approach for statically scheduled code by breaking the code into several synchronous instruction streams. By representing code in a fundamentally new way, we demonstrate that we can create robust processors that can handle dynamic levels of instruction level parallelism (ILP), and demonstrate the potential it has to target traditional weaknesses typically associated with statically scheduled processors. This dissertation is an exploration of the consequences of allowing for multiple instruction streams, as well as the possibilities opened up by changing program representation to allow for several simultaneous streams of instructions.

Chapter 1

Introduction

Processors individually dealing with sequential instructions have an upper limit of performance. To make headway, then, we look to make progress on multiple instructions in parallel, leading to speedup using instruction-level parallelism (ILP). ILP is often obtained through pipelining, but to get further performance improvements, modern techniques involve simultaneously issuing multiple instructions that are not constrained by dependencies. Maximizing ILP in general programs, however, is no trivial task, and remains a long-open research problem.

The two most popular approaches are Very Long Instruction Word (VLIW) processors and Superscalar processors. However, both processors have limitations. VLIW processors are quite energy efficient, and often excel when the instruction streams are

highly regular, such as in digital signal processing or scientific computing, but they often struggle due to a lack of flexibility and significant code bloat from excess no-operation (*nop*) instructions. Superscalar processors are capable of handling irregular programs much better, but lack scalability due to reliance on complex front ends that focus on extracting parallelism from a single instruction stream.

This work proposes and expands upon a new processor architecture, which we refer to as the Synchronized Lane Architecture (SLA) processor. SLA architectures can capitalize upon advancements made in the VLIW field, but offers opportunities to address key weaknesses of VLIW processors in general-purpose computing. Our processor architecture was designed so that the back end behaves similarly to a VLIW processor, but addresses several key weaknesses of the VLIW processor by exploiting simultaneous, synchronous instruction streams.

While VLIW processors represent parallelism through bundling independent operations into packs of operations in a single stream of instructions, the SLA paradigm instead represents parallelism using streams of operations. By decoupling these instruction streams, we provide mechanisms for streams to be dynamically started and resumed, eliminating many NOPs and allowing for instruction streams to execute independent of other streams.

The SLA processor has several key features that make it distinct from prior processors. Similar to a VLIW processor, the SLA processor is split into sections of execution

units. We refer to these units in the SLA processor as lanes, where each lane behaves similarly to a traditional pipelined processor, and is responsible for an individial instruction stream. Unlike a VLIW processor, each lane is provided it's own program counter so that each lane can track a separate instruction stream. Since one stream must always be active, we also provide a special lane, called lane 0, that is responsible for managing other instruction streams. Finally, we provide a new ISA design that encodes synchronization using a single bit embedded in each instruction, which we call the suspend/resume *(sr)* bit, and demonstrate that it is sufficient for handling instruction stream synchronization in most cases.

The SLA processor behaves similarly to a VLIW processor by ensuring lanes have the same lock-step behavior. However, we allow the design to stop lanes when not executing code to reduce *nops* in code and improve instruction cache (icache) occupancy. We demonstrate that this approach not only maintains performance, but also improves energy usage.

The SLA processor architecture is orthogonal to many techniques that already exist, and provides the opportunity to design hybrid architectures where the SLA design can effectively tune a processor to various widths during periods of varying IPC. Furthermore, we explore variations of the SLA processor in which lanes are small VLIW-like processors instead of the single-lane approach. This wide-lane approach allows for much of the same benefit that the SLA processor provides, without the

need to have as many meta instructions and hardware structures.

We demonstrate that allowing for multiple instruction streams opens many potential design paradigms, since this design can represent parallelism more efficiently than a single instruction stream.

In the rest of this work, we intend to expand upon what we have discussed in the introduction. Chapter 2 expands the background of current processor designs, past efforts to improve VLIW performance, and techniques for static scheduling that have shown success in the past. Chapter 3 presents the challenges of creating a processor simulator when a reliable compiler is not available, and techniques for pinpointing the point of failure when both the compiler and the simulator could be the source. Chapter 4 defines the fundamental structure of the SLA architecture, which consists of the instruction set, the scheduling, and the processor hardware requirements. Chapter 5 explores a configuration of the SLA architecture that maintains the same maximum width of an original SLA processor while reducing the overhead of maintaining multiple program counters by lowering width-granularity in the processor, instead defining lanes as small VLIW-like processors that process packs of operations. Finally, Chapter 6 concludes the work with an analysis of what is discussed.

Chapter 2

Background

2.1 Existing Architectural Approaches

Post-Moore's law, parallelism, the concept of making progress on multiple instructions at once, has reigned as the king of improving performance. Parallelism can be achieved through many different techniques, such as thread-level parallelism or distributed computing solutions, but the most transparent approach to parallelism, as well as the most universally applicable, has been instruction-level parallelism. Compilers and processors work together to optimize and schedule code to maximize ILP in an attempt to make single threads as performant as possible. Modern processors have moved beyond single-stage processors to simple pipelined processors, and from

pipelined processors to processors with wider issue widths containing multiple execution units. However, program dependencies limit which instructions can be sent to the execution units in a specific order to ensure program correctness. The two major architecture paradigms that have found the most success in showing performance while maintaining correctness have been superscalar processors and very-long-instruction-word (VLIW) processors.

2.1.1 Superscalar Processors

Superscalar processors dominate the current research space due to their performance and ability to exploit ILP. To accomplish dynamic issuing of instructions, modern processors have a complex front end designed to determine available instructions and send them into the pipeline as resources become available such that the dependencies in the program are not violated. Expanding these front ends to allow for larger amounts of parallelism when available can be quite powerful, but it is a complex solution that offers diminishing returns. First, larger fetch windows lead to larger misprediction penalties and more frequent branch mispredictions. Second, reordering instructions does enable potential speedup, but it is not guaranteed if the instructions which execute early are not on the critical path. Third, when waiting on outstanding memory instructions which have missed in a cache, all dependent instructions must wait, leading to large chains of dependent instructions, consuming many resources.

Finally, it is difficult to design a simple and efficient front end for superscalar processors due to the logic they need to maximize parallelism. To maintain relationships between instructions, superscalar processors pay attention to which waiting instructions' values are ready, and broadcast instruction results to all waiting instructions, in case those instructions require the computed values. This constant set of broadcasts leads to complex, energy intensive logic, but allows for significant performance gains.

2.1.2 In-Order versus Out-Of-Order Superscalar Processors

In-order superscalar processors offer a simpler alternative to the standard out-of-order superscalar processor. They have drastically simpler front ends, as logic for dynamically finding independent instructions are limited to comparing the next-to-issue instruction to in-flight instructions. The dependencies in an in-order-superscalar can thus be represented by a bit-vector that can be accessed at instruction decode, where each bit indicates processor registers and thus dependencies the instruction may have. Out-of-order processors instead use a complex set of reservation stations to both allow for register renaming and to detect when an instruction's operands are available. This logic is expensive, but offers a significant performance benefit, leading most processors to have at least it's high-performance cores to be an out-of-order implementation.

2.1.3 VLIW Processors

The most common alternative approach to superscalar processing is the very-long-instruction-word (VLIW) processor. In practice, VLIW and in-order superscalar processors behave quite similarly, with both relying on compilers to exploit parallelism. However, the instruction representation and approach to scheduling are different. VLIW processors are statically scheduled, where each instruction is a pack of several operations (sub-instructions). These processors have a significantly simplified front end which reduces power and complexity. However, there are some notable penalties with this simple approach. Since instructions must maintain a fixed width to allow for each sub-instruction to align properly to each VLIW execution unit, instructions need to be padded out to fit the entire width of the machine, even when instructions do not have a meaningful operation to execute in the given execution unit. Padding instructions out with a large number of what is equivalent to no-operations causes VLIW processors to suffer from inefficient code-space usage. Since VLIW instructions are so large and cannot easily be reduced in size, VLIW code often leads to larger code size and worse instruction cache performance. Since instruction width is critical for maintaining alignment, there is also a limitation in terms of backwards compatibility, as a larger width VLIW processor cannot run code compiled for a smaller width VLIW processor and vice versa. VLIW processors also lack the dynamic nature of a superscalar processor, since packs of instructions proceed in lock-step through the

pipeline. The VLIW instruction holds up all execution units until the slowest instruction completes, meaning a unit that executes a shorter instruction cannot be handed new operations to process until the other parts of the VLIW instruction have completed. So, despite power efficiency and simplicity advantages, these limitations in the VLIW processor have caused it to remain in use only for relatively niche cases such as signal processing, compared to the ubiquity of it's superscalar counterpart. Throughout this proposal, we will refer to a VLIW instruction as a grouping of operations, called a pack, with each individual operation in the pack being a sub-instruction. These sub-instructions can be considered as equivalent to a single instruction in a superscalar. Similarly, we can view a VLIW processor as a group of pipelined processors adjacent to each other, with each pipelined processor being responsible for it's own sub-instruction. We refer to each individual pipeline as a lane, and can refer to an individual lane by treating it as an index into the part of the instruction for which it is responsible.

2.2 EPIC Architecture

The VLIW processor's shortcomings are a well-known problem. Intel and Hewlett-Packard both recognized this, and worked to develop an architecture to resolve these issues, dubbed the Explicitly Parallel Instruction Computing (EPIC) architecture [33]. The EPIC architecture addressed backwards compatibility by allowing packs of

instructions (called bundles) to not remain aligned to the system by using meta-bits to give the processor a concept of the width of the program. Packs also had a bit to indicate if the following pack was dependent on the current pack, allowing larger width processors to issue multiple packs in a single cycle if the width of a larger processor supported it. The bundle strategy meant that EPIC processors needed complex structures to dynamically determine what instructions had to be fetched in the next cycle. The other challenge is that memory instructions, especially loads, do not have deterministic delays and can potentially cause an entire pack to stall. To maintain the lock-step nature, the EPIC Architecture made heavy use of prefetching and hints in the ISA to minimize cache misses, but the result was that the architecture was limited by cache-access time. Although the primary motivation in the creation of the architecture was to create a simpler front end to allow for faster clock speeds than an out-of-order superscalar, processors ended up being locked to the speed of the cache instead.

The creation and design of EPIC significantly advanced the compiler techniques for code motion. EPIC allowed for communication from a processor to a compiler of where in the cache hierarchy a given datum would likely be found at compile time through the use of prediction and profiling. Furthermore, the ISA provided significant tools to manage the caches to give the compiler more predictable latencies. Furthermore, the compiler was provided tools to allow for data speculation. While traditionally code must remain conservative, the EPIC architecture introduced the

ability for the compiler to generate schedules that assume certain loads and stores are to different memory locations, even if there is some small chance that they map to the same location. This speculation was achieved using a data speculative load and a data verifying load, and used the verifying load to detect if there was a memory conflict in the processor[3].

To address branches, the EPIC architecture split branches into a prepare-to-branch computation, which was responsible for target computations, a compare, which computed the branch condition, and the actual branch which indicated control-flow needed to be changed. By splitting the branch into three parts, the prepare-to-branch operation allowed for the hardware to speculatively prefetch instructions at the branch target. Furthermore, the architecture supports predicating instructions, where instructions are executed only if a provided boolean value (called the predicate) is true. By shifting from traditional control flow to predication, the design aims to mitigate complexities and delays caused by traditional branches.

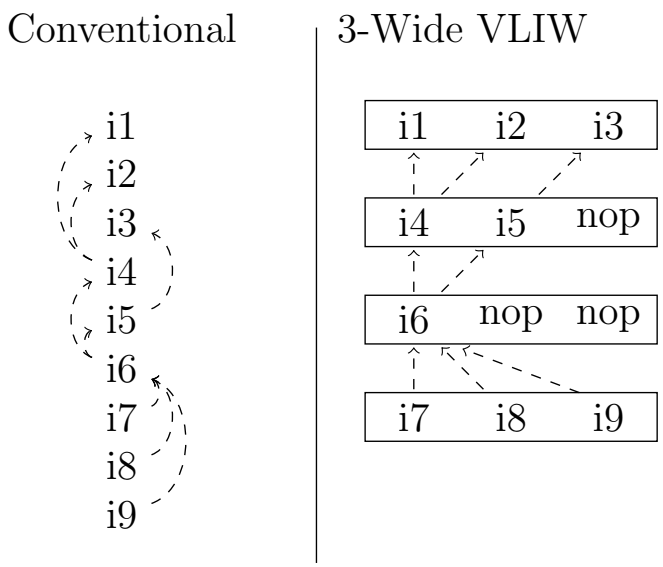


Figure 2.1: Traditional Instruction Stream vs 3-wide VLIW processor

2.3 Code Scheduling and Representation

2.3.1 Code Representation

Superscalar processors and VLIW processors represent their instructions as a single stream of instructions. For superscalar processors, each instruction represents some form of operation that the processor must handle, whether it is data movement or a computation. For VLIW processors, each instruction serves as a pack of operations. Each execution unit in the processor is responsible for a sub-instruction of each statically-scheduled instruction, and each sub-instruction serves as the equivalent of a single superscalar processor instruction. If a lane does not have any actions

scheduled for a cycle, then that lane is provided a no-operation (*nop*) sub-instruction. How the code is laid out is quite indicative of how these two processor types behave. Figure 2.1 shows how instructions may be provided as schedules. Dotted arrows represent dependencies in the code, so operation i4 can be read as dependent on i1 and i2. On the left is a conventional stream, commonly used by superscalar processors. In-order processors, such as in-order superscalar processors, execute the conventional code from top to bottom, and will not progress if the next instruction to be issued does not have available resources. Out-of-order superscalar processors dynamically select instructions that are independent and ready to issue, meaning that as long as the indicated dependency chains are not violated, it is possible to reorder instructions. For instance, an out-of-order processor may decide to execute i3 prior to i1 and i2 if resources for i1 and i2 are not available. Since a dependency is not violated, this would be a valid execution order. VLIW processors, however, have instructions that are grouped together as bundles of operations, allowing for parallelism without the need for the same complex selection logic of an out-of-order processor.

For both processors, instruction ordering is important to achieve the best possible performance. For any nontrivial code segment, there may be many valid instruction orderings that can produce an equivalent computational output. Yet, despite many correct solutions, not all instruction orderings make effective use of the processor resources that are available to them. An instruction ordering that maintains a high degree of temporal locality, for instance, can provide much better performance than

an instruction ordering that more often requests information that has since been evicted from a higher-level cache.

The problem of finding the best order of instructions is NP-complete, hence we often rely on heuristics to find instruction layouts that maximize performance. The art of effectively ordering instructions is referred to as instruction scheduling. Out-of-order superscalar processors are referred to as dynamically-scheduled processors, and are designed to select instructions out-of-order when in-order instructions are otherwise unavailable, improving speculation and parallelism capabilities. In-order and VLIW processors, however, have a simpler front end that cannot dynamically reorder instructions, meaning instruction ordering becomes even more important. While many scheduling algorithms exist, we cover a handful of them particularly useful for these statically scheduled processors.

2.3.2 List/Basic Block Scheduling

The simplest scheduling algorithm is often referred to as list or basic block scheduling. In list scheduling, each instruction is assigned a heuristically determined priority, based on both the compiler and the machine being targeted[14]. Then, the algorithm greedily schedules the highest-priority instruction that has not been scheduled, and

whose dependencies have already been fulfilled. Since instructions which have outstanding dependencies cannot be scheduled using this process, violations in instruction ordering cannot occur. We generate a valid schedule by repeating this process until all instructions have been scheduled. These schedules are typically applied to moving instructions within basic blocks, meaning that the window for ordering is typically quite small. Thus, limited opportunities are available for maximizing performance.

2.3.3 Trace Scheduling

Trace scheduling takes the concept of list scheduling and expands it to be run on a set of potential traces (or profiles) throughout a set of blocks in the code, while prioritizing code that is most commonly executed in the traces[11, 14]. At each entry and exit point, code can then be inserted to compensate for unique traces that do not match the commonly scheduled portions. By expanding the available distance for instructions to move, trace scheduling significantly increases opportunities for improved scheduling and can potentially collapse branches into larger basic blocks. The global perspective provided by trace scheduling allows for significantly increased opportunities over basic block scheduling.

2.3.4 Superblock/Hyperblock Scheduling

The biggest penalty of trace scheduling typically comes from potentially large expansions of compensation code[14]. Since trace scheduling requires code to compensate when a program takes a less common path through the scheduled section, many duplicates can be made of instructions throughout the program. To mitigate code expansion, the concept of superblock scheduling was developed[17]. Superblocks limit code motion and reduce potential duplication while maintaining much of the advantages of trace scheduling by focusing on traces only through superblocks. A superblock is simply a large set of basic blocks with a single entrance. All side entrances are disregarded. This approach minimizes potential duplications due to atypical back edges. A slight extension, called hyperblock scheduling, does the same thing, but adds predication to merge paths stemming from some conditional branch. The hyperblock and superblock scheduling techniques are notably implemented in the Trimaran compiler, a major compiler designed to produce code for VLIW (and later EPIC) architectures[6, 38].

2.3.5 Static and Dynamic Scheduling

Out-of-order superscalar processors dynamically schedule operations in their instruction window, and thus are able to both more effectively deal with dynamic latencies as well as support speculation. However, they cannot handle the holistic approaches that compiler scheduling can offer, meaning even out-of-order dynamically scheduled processors are reliant upon well-scheduled code. EPIC pioneered many past approaches to allow for static scheduling to emulate some of the advantages of dynamic scheduling, such as using prediction-with-compensation code in EPIC architecture and predicating instructions[3, 12, 33]. Similar approaches to combine the advantages of dynamic scheduling and static scheduling have been done to great effect, such as predicating and executing instructions prior to dependent branches[37], limiting side-effects of predicted instructions in statically scheduled machines until a branch is confirmed[39], or allowing for schedules that explicitly instruct the processor when to predict values for loads and when to check for load correctness to hide memory latencies[13]. Dynamic scheduling has significant overhead in terms of complexity, and work has also been done to offload tasks of dynamic scheduling where benefits were minimal to static scheduling in an attempt to reclaim area savings[7].

Chapter 3

ISA, Architecture, and Compiler Codesign

3.1 Introduction

Statically scheduled processors that implement novel architectural changes may need to offer a new interface for compilers to properly take advantage of the hardware. In some cases, this may only be a small addition to an existing ISA, but in situations where the hardware is significantly different an entirely new ISA may need to be developed.

New ISAs, in a practical sense, tend to be large undertakings that require input from

many invested parties, such as in the case of RISC-V. An equally important and challenging task, however, is the verification of the design. Verification is a fundamentally complex task that requires both an environment for generating executables in the new ISA, as well as the environment for executing the executables to locate errors.

As part of our research, we first designed and implemented a new ISA called the Statically Controlled Asynchronous Lane Architecture (SCALE) ISA. Designing the new ISA involves creating a complex tool chain to produce the code and a simulator capable of running programs compiled for the ISA.

The complexity of both the simulator and code generator leads to a chicken-and-egg problem, where verification of the simulator relies upon correctly generated code, but verification of the code generator requires a correct simulator, compiler, optimizer, and assembler. While some errors can cause a program to terminate in a quick fashion, such as a segmentation fault detected in the simulator, there are many errors which can take a significant amount of simulation time after the error occurs before a simulated program fails. Detection of the cause of an error becomes even more difficult when a program terminates gracefully, but output is simply incorrect.

Due to the complexity of each interlocking part, finding the first point in which a program deviates from expected behavior becomes an important and challenging first step for locating the errors in the system.

To address the issue of locating points of failure, we provide a workflow for generating a new ISA using MIPS as an intermediate ISA. This workflow was implemented by a team comprising of members from Michigan Technological University and Florida State University. In this workflow, we first demonstrate a technique for quickly pinpointing where a simulator diverges from the expected control path. Next, we expand upon the divergence detection technique to allow for comparisons from a single pipelined processor to a VLIW processor. Finally, we expand the technique to allow for designs that support moving from a VLIW processor to processors where instruction ordering and instruction types may be entirely different, despite the program being equivalent.

3.2 Designing a Bilingual Assembler

To validate our new ISA, we first created a *bilingual assembler* with a verified ISA, alongside our target SCALE ISA. Described by Mortensen et. al [25], a bilingual assembler is one that accepts two instruction sets as valid. For our verified ISA, we used MIPS as it was the closest to our target ISA. To create a bilingual assembler, we first defined our target ISA, and provided macros that mapped all MIPS instructions into SCALE instruction equivalents. We then developed a simple functional simulator that accepts code using the SCALE ISA. The simulator is a bare minimum simulator to read in instructions, process them, and emit the results. All simulated hardware

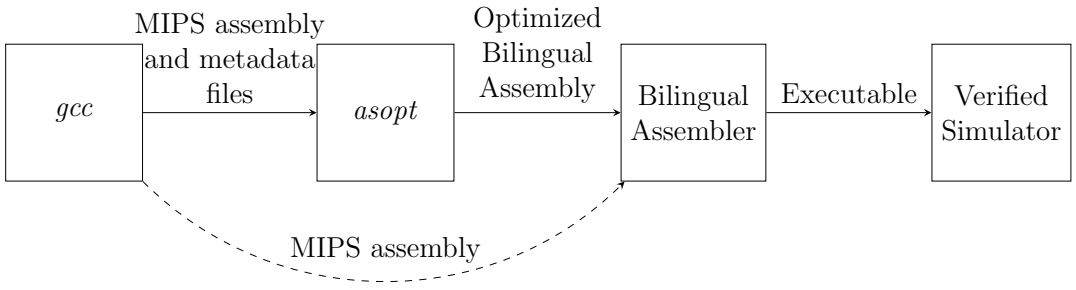


Figure 3.1: Workflow of Code Generation

was the minimum for what was required for the simulation to run.

During this first step, we tested the developed ISA using verified MIPS programs. We implemented each MIPS instruction as a macro of one or more SCALE instructions, and used the designed MIPS to SCALE macros in the assembler. By transforming the MIPS ISA to a SCALE ISA using mechanical macros, the processor simulator running the SCALE ISA was able to reach a level of robustness prior to adding any further tools.

Once the simulator and basic assembler had a degree of confidence, we provided the tools to a team responsible for our assembly optimizer, which we refer to as *asopt*. *Asopt* takes both the MIPS assembly, as well as some metadata files, that are produced by *gcc* to output a modified assembly file. The metadata files are used for register liveness analysis across function calls, and *asopt* modifications are dependent on flags and configurations. Different flags for *asopt* specify the types of optimizations to be performed. By using gcc, we can compile files using a variety of source languages

as well as leverage optimizations performed by the gcc compiler. Figure 3.1 shows how the workflow of code reaches the simulator and the role asopt plays as a translator for MIPS to SCALE. Prior to the introduction of asopt, we instead followed the control flow indicated by the dashed line, where we directly provided the assembly produced by *gcc* to the bilingual assembler.

Initially, *asopt* only provided MIPS code to the bilingual assembler, and since the assembler provided translations from MIPS to SCALE using mechanical macros, the optimizer was able to implement one SCALE instruction at a time. As more SCALE instructions were implemented, *asopt* could phase out MIPS instructions as they became redundant. Once instructions were implemented, we could verify instruction correctness by disabling old MIPS instructions from the bilingual assembler.

3.3 Fast Divergence Detection

When developing for two processors that have different architectures, but the same source program, detecting divergences quickly can be tracked through monitoring any unexpected changes that occur in the execution of the program. There are several variables that can be tracked throughout the lifetime of a program despite differences in architecture. Chief among these is the control flow of a program in the simulator. In programs without randomness, the path taken through a program is deterministic,

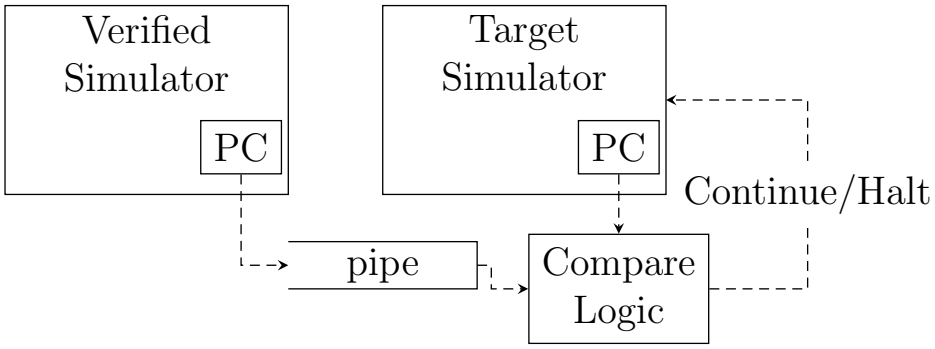


Figure 3.2: Basic Layout for Functional Verification Detection

and minor changes in a program can drastically alter control flow. For these deterministic programs, we can design an error isolator for a target simulator by comparing the program counter of retiring instructions against a dynamically created trace of the program from a functional simulation to determine proper execution.

Figure 3.2 shows how fast divergence detection works. Since traces are often quite large, dynamically generating the trace with a verified simulator allows us to compare proper execution with a target simulator. The verified simulator, at retire, emits program PCs to a named blocking pipe. The target simulator, at retire, reads the PCs and verifies that the values match with what the target simulator has calculated. If at any point values diverge, the simulation can immediately pause to allow for inspection at the point of failure.

Note that this technique can be extended with a second named blocking pipe to ensure both simulations behave in lock-step, allowing for inspection of the verified simulator at the same point of failure if needed, such as comparing register files.

When comparing different ISAs, different program instruction counts can cause counters to become desynchronized. If the two ISAs have a one-to-one mapping of instructions, then comparing PCs are sufficient. However, when that is not an explicit guarantee, one ISA may split a computation into a different combination of instructions compared to another ISA or vice versa, which may change instruction counts. Since instruction spaces are no longer identical in layout, comparing program counters becomes impossible. In these cases, progress must be monitored through other means.

When PC comparisons are not practical, we instead use a technique we call *coarse-grain tracking*. In coarse-grain tracking, we track the program progress through a combination of tracking side-effects in the program, and relating them to function call and return order. At retire for store instructions, we emit stored values and addresses to our verified simulator, and use the same design in Figure 3.2 to track when emitted values no longer match, similar to comparing PCs. Similarly, we make a note of each time an instruction for entering or returning from a function is retired, though we do not compare destinations, since the simulator does not have a mapping of where functions are in the code space. While the coarse-grained approach is not as responsive as tracking program-counters, control flow and value divergence is typically quickly reflected through stored values and function calls.

3.4 Considerations for comparing execution between a simple instruction streams vs nonstandard instruction streams

In most cases, PCs are more responsive for divergence detection than side-effect tracking, since changes are typically reflected more quickly. In architectures where instructions are represented in bundles of operations, like a VLIW processor, steps are taken to enforce equivalence during simulator development and assembly optimization. To take advantage of PC comparisons when shifting to comparing a verified single-instruction execution stream to a VLIW simulator, we take steps to ensure that the streams maintain equivalence. VLIW instructions are represented sequentially as a group of instructions in the verified simulator. First, we enforce operations within a single VLIW instruction respect antidependencies, meaning that execution of each individual instructions in-order in the pack will still allow for correct execution. Second, we compare the PC of each individual operation in the VLIW pack from first to last, instead of using only the PC of the entire long instruction word. We do not compare PCs of *nops* in both the VLIW pack and target simulator due to control flow changes causing discrepancies in retired instruction counts.

When shifting to a new processor description, instruction ordering and PC representation can change. Depending on the approach, we have found that variants of coarse-grained tracking are often effective.

3.5 Technique Limitations and Uses

When pinpointing errors, the technique of divergence helps detect where the location of the error is. The nature and type of error typically requires further analysis. Ensuring identical workspaces can be particularly important when dealing with simulators that use specific resources.

In particular, system calls can cause a divergence in loaded values or control flow if the verified simulated program and the target simulated program are in different directories and the exact file path is of a different length, in which case the function call `strlen()` may return a different value or iterate a different number of times. Similarly, values such as file descriptors may return different values depending on the simulator's proxy kernel implementation, leading to something like an `fopen()` call to write a different file descriptor. When implementing these techniques, attention should be paid to the types of system call differences that may arise, and the environment should ensure the proxy kernels behave the same between simulators.

Chapter 4

Synchronized Lane Architecture(SLA)

4.1 Introduction

Statically scheduled processors require the compiler to be responsible for exploiting instruction-level parallelism, allowing for significantly simpler frontends. However, despite their simplicity, they are often left to niche application areas, such as regular scientific code and digital signal processing. Due in part to these use cases, advancements in statically scheduled processors started to stagnate with the sunsetting of Intel and Hewlett-Packard's Itanium architectures.

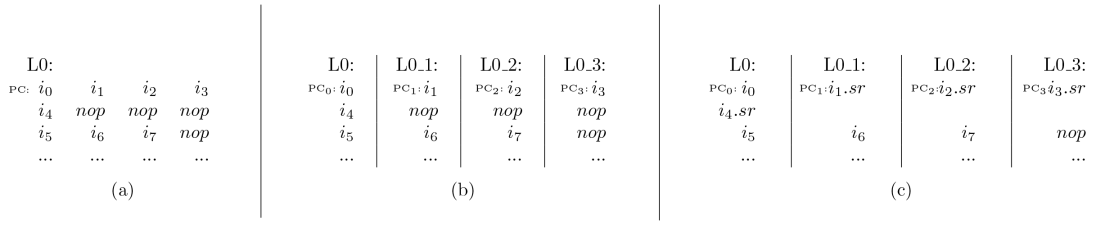


Figure 4.1: Realizing SLA Operation through Single and Multiple Program Counters. (a): VLIW Code, (b): Multi-lane code, with PCS for each stream, (c): Multi-lane code with lane start/resume

Most statically scheduled architectures are variations of very long instruction word (VLIW) processors, which group independent operations into long instruction words (or bundles of operations). Hence, long instruction words are groups of operations that look much like what is portrayed in Figure 4.1(a), which shows the code for a 4-wide VLIW processor. In this representation, each long instruction is arranged in contiguous memory and are referenced using a single PC. The execution of this code is fundamentally the same as that of a simple pipelined processor, except each instruction can encode more than one operation. If there aren't enough independent operations to be packed into the instruction, no operations (*nops*) are substituted instead. This code execution would proceed in a VLIW processor as follows: In the first cycle, the instruction containing i_0, i_1, i_2, and i_3 is issued. In the next cycle, the instruction containing i_4 and three *nops* is issued. Finally, in the third cycle, the third instruction containing i_5, i_6, and i_7, along with another *nop* is issued. While there have been variations, the outlined execution is typical of VLIW processors where a single program counter (PC) points to an instruction which contains multiple operations. Having multiple operations per PC allows for a simple way to to sequence the parallel

execution of operations.

This **horizontal** design for representing parallelism is powerful, but it has certain limitations. While operations can remain consistent between two VLIW processors, code generated for a VLIW of a given width cannot run on a processor of a larger width, limiting backwards-compatibility without additional mechanisms such as the stop bits and bundle templates used in the Itanium architecture[15, 35]. Furthermore, the instruction stream is filled with large numbers of *nops* that reduce instruction cache performance and lower much of the energy savings of the simplified frontend. We believe there is potential in reevaluating the fundamental representation used by VLIW processors. In this respect, many of the weaknesses of past statically scheduled processors can be targeted by reevaluating the approach we use to represent parallelism in the instruction stream.

In this chapter, we introduce a new statically scheduled processor architecture which we call the Synchronized Lane Architecture (SLA). The SLA is a multi-threaded processor architecture in which thread synchronization is accomplished through simple instruction encodings under compiler control. As such, it can emulate the full behavior of statically scheduled processors and enables a **vertical** design as shown in Figure 4.1(b), as opposed to the horizontal design shown in Figure 4.1(a). In this approach, independent operations forming the VLIW instruction are distributed onto separate streams which are processed in a lock-step fashion. Each cycle, all streams

use their own PC to simultaneously fetch the next operation encoded in their stream, in essence dynamically forming the VLIW instruction through the parallel operation of the lanes. We refer to each stream and their accompanying execution unit as an execution *lane*. Although each lane maintains its own PC, they read from and write to a shared register file, just like a VLIW processor.

Now, consider the code sequence shown in Fig. 4.1 (b). When executed on an SLA processor, in the first cycle, lane 0 issues i_0, lane 1 issues i_1, lane 2 issues i_2 and lane 3 issues i_3. The PC for each lane is incremented by the length of an operation, then the process is repeated. Lane 0 issues i_4 and the other three lanes issue *nops*. Finally, in the last cycle, lane 0 issues i_5, lane 1 issues i_6 and lane 2 issues i_7 while lane 3 issues another *nop*, yielding the same exact schedule as the VLIW processor.

While benefits of this approach may not be initially obvious, it is easy to see that the mechanism becomes quite powerful when we allow lanes to dynamically suspend and resume their operation. Encoding when a lane should be suspended or resumed is easily accomplished by reserving a single bit of the instruction representation, which we refer to as the suspend-resume *(sr)* bit. Suspend-resume bits are the primary means of synchronization in SLA. Since a suspended lane cannot resume operation on its own, we keep one lane, namely lane 0, always active and use that lane's suspend-resume information to resume other lanes.

In this chapter, we utilize this simple synchronization mechanism provided by the

multi-threaded design of SLA to control independent streams of instructions scheduled by the compiler in a lock-step fashion to emulate VLIW architectures. Clearly, this approach is more powerful than just simply emulating a VLIW processor architecture. We would like to develop SLA into a multi-threaded, compiler controlled architecture to open up new research in statically controlled processor architectures. This chapter, being the first step in that direction, simply aims to demonstrate that even when emulating simple VLIW behavior, the SLA approach is efficient and flexible enough to provide non-inferior performance to that of a traditional VLIW, while providing significant power savings.

Revisiting our example, any instruction in lanes 1 through 3 with its *(sr)* bit set will still result in the lane's PC being incremented after that instruction is issued, but the lane ceases making progress after the current operation is issued. On the other hand, an instruction with the *sr* bit in lane 0 indicates that other suspended lanes should resume execution in the next cycle.

We can see how the *sr* bit works in Figure 4.1(c), where operations with the *.sr* mnemonic have the suspend-resume bit set. In this version, each stream of instructions is contiguous in memory instead of packs of operations being contiguous. In the first cycle, lane 0 issues i_0, lane 1 issues i_1, lane 2 issues i_2, and lane 3 issues i_3. Since lanes 1, 2, and 3 each issued an instruction with the *sr* bit, they increment their PCs to *i6*, *i7*, and the *nop* instructions, respectively, but enter a suspended state. The

next cycle, only lane 0 issues *i4*, and other lanes remain suspended. Since the lane 0's *sr* bit is set, other lanes are instructed to resume execution in the next cycle. Finally, just like in the last example, the final group of operations can issue with each lane issuing it's own operation.

As it can be seen, having the ability for lanes to dynamically start and stop allows us to eliminate the majority of *nops* in the program. Furthermore, when a lane is not active, lanes can enter into a low-power state without the need to fetch and decode *nops* or access that lane's instruction cache. Since lanes that never execute can remain in a suspended state, we can easily execute code compiled for a narrower machine on wider machines without any code modifications.

The SLA approach allows for significant power reductions, but introduces special challenges. For example, before a stream can be utilized, the PC responsible for that lane must be initialized with the beginning address of that stream, such as *L0_1, L0_2*, etc., shown in Figure 4.1(c). Such beginning addresses will change upon encountering any branch instruction, as well as function calls, since all lanes need to change direction to their respective target addresses to maintain the lock-step flow of the instruction streams. Use of multiple branch instructions is a possibility, but it will not be efficient as these additional branch instructions will take up valuable issue slots. Furthermore, such a solution would require all lanes to be active during the execution of those branch instructions. Due to separate compilation of functions,

the number of active streams necessary for a given function will not be known in advance, so mechanisms are needed for the processor to be in a predictable state upon entering the function. Similarly, the callee's use of the streams may be different from the caller's needs, so the architecture must provide tools to restore the caller's streams as well.

4.2 Overview of the SLA Approach

A SLA processor can be designed to allow up to n operations to simultaneously issue, one per each execution unit. We refer to each unit as a lane, which we label as 0, 1, ..., n-1. We generate code so that the instructions for each lane within a function are contiguous in memory. Having each lane appear as though they are a single instruction stream facilitates fetching the next sequential instruction for a lane when there is not a transfer of control.

We refer to the group of instructions that are synchronously executed together as a *pack* of instructions. A *pack* of instructions can be seen as equivalent to a wide-instruction in a VLIW processor in function, but operations in SLA packs are not in contiguous memory, but instead are fetched from separate instruction streams. Since operations are not contiguous, *nop* instructions are not required for alignment. Since alignment is not an issue, synchronization becomes particularly important. To

maintain synchronization, we provide several mechanisms to ensure that lanes are synchronized.

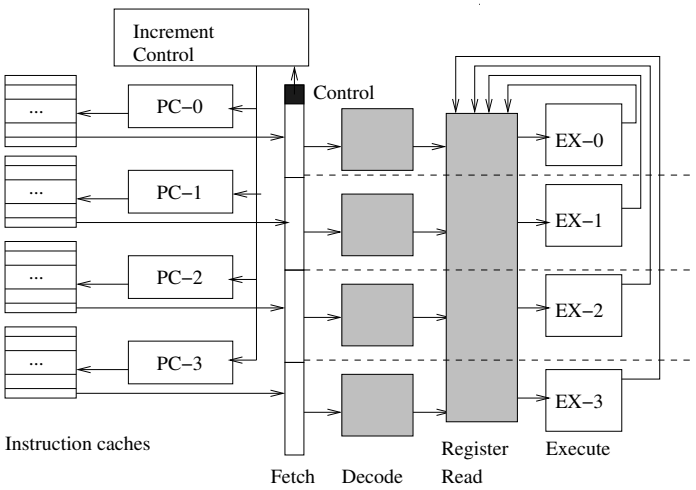


Figure 4.2: An Overview of the SLA Processor

In an SLA architecture, lanes can be in one of three states, all of which are under complete compiler control:

† An *inactive (or halted)* state indicates that the lane has not started execution for that function or has been explicitly disabled. This is the initial state when the processor is powered on for all lanes except lane 0.

† A *suspended* state indicates that the lane's execution had started, but is currently suspended.

† An *active* state indicates that the lane's execution for that function has started and is currently active.

To prevent deadlocks, lane 0 is always active. However, the other lanes can be active, suspended, or inactive at various points during the execution of a program. When a lane is not active, the lane is left in a low-power state, since it does not need to fetch or process any instructions. If a lane does not need to provide an instruction in a given cycle, access to that lane's cache is also suppressed.

Since the architecture is synchronous (i.e., lock-step) a miss in any instruction cache associated with an active lane stalls all lanes. If multiple caches miss, each one pipelines their request to the level 2 cache so that progress can be made on multiple caches at once.

An overview of the processor is shown in Figure 4.2. Each of the lanes maintain their own program counter, have their own dedicated instruction cache, and have a dedicated decoder. Delineated by the dotted lines, each lane looks and functions similar to that of a simple pipelined processor in the frontend, but they share a single register file. In fact, an SLA processor can directly execute a single RISC style instruction stream in lane 0, since lane 0 is active when the processor is initialized. In this respect, code generated for a simple pipelined processor can be considered to be a special case of SLA code where the width of the processor is always one operation wide. The rest of the execution pipeline is nearly identical to a typical VLIW processor, including its forwarding logic, as well as the execution pipelines.

A significant difference between an SLA processor and a VLIW processor is the increment control unit, which is responsible for maintaining lane status for each lane. These responsibilities include suspending, resuming, or, halting a particular lane. Since these actions need to take immediate effect, *sr* bits always occupy the same position in an instruction. Having a dedicated bit permits immediate use of this information by the control unit as soon as an instruction is fetched and before it is decoded.

The SLA paradigm is orthogonal to many VLIW pipelined processor designs. In other words, any VLIW processor design can be converted to an SLA architecture by replacing the frontend with one which utilizes streams in the described manner. Therefore, we do not further discuss the details of the execution pipelines under the paradigm, except the necessary modifications, as needed. In our evaluations however, we faithfully simulate a 5-stage pipelined VLIW processor in a cycle-accurate manner. This design is utilized as the baseline processor, as well as forming the basis for the SLA architecture by replacing its frontend as previously discussed.

4.3 The SLA ISA and Code Generation

Traditional VLIW processors employ an instruction encoding where each instruction has a number of operands and multiple opcodes describing each of the multiple operations to be performed by a single wide instruction. Since the SLA architecture fetches individual operations using multiple program counters, each of the streams can fetch and execute RISC style instructions that describe the particular operation to be performed. In order to evaluate the SLA paradigm, we rely on an entirely new instruction set architecture. We refer to this new ISA as the Statically Controlled Asynchronous Lane Execution (SCALE) ISA. The SCALE ISA introduces a suspend-resume bit, which is a dedicated opcode bit, so the SCALE ISA has fewer available opcodes than other 32-bit ISAs. Furthermore, we introduce several instructions dedicated to supporting multiple-lane control-flow.

The single bit *sr* field occupies the most significant bit in each instruction. In this manner, lane control actions can be completed when the instruction is fetched without decoding the instruction.

A dedicated *sr* bit also allows us to have two versions of the same instruction, one which sets the *sr* bit and another which has a zero *sr* bit. The former utilizes the assembly mnemonics of the instruction followed by `.sr` and the latter does not have

the suffix. For example, our SLA add instruction would either be `add $2,$3,$4`, or, `add.sr $2,$3,$4`. As previously mentioned, when the *sr* bit is set for an instruction in any lane other than lane 0, then the bit indicates that this lane is suspended after the current instruction executes. Any subsequent instructions are not to be dispatched until lane 0 encounters an instruction with an *sr* bit set. In all lanes, except lane 0, the *sr* bit affects (suspends) only the lane in which it is executed and in lane 0, it affects (resumes) all other suspended lanes. We later discuss how suspends and resumes are accomplished in the instruction pipeline. Since *sr* bits are handled by their own lanes, multiple instructions can suspend the execution of their lanes in the same synchronized pack.

4.3.1 Handling Function Calls

The SLA paradigm relies on all control-flow being primarily controlled by lane 0, and only lane 0 is allowed to have branches, special call and call-register (callr), and return instructions. Since function calls are much more involved than simply changing the program counter to the target address, we extend the base instruction set with the addition of special call and return instructions. These instructions ensure the program will be in a predictable state upon a function invocation. Two instructions we introduced, namely call/callr instructions, are analogous to *jal* or *jalr* in MIPS, but they cause all lanes other than lane 0 to be marked as inactive while the lane 0

PC is set to the call target.

When a *call* or *callr* instruction is encountered, the processor saves the processor state into a set of registers, which is equivalent to the return address register in conventional architectures. The special registers are called *RT* registers, and each lane is provided one to store the PC state at the point of the function call. Similarly, a previous lane status *(PLS)* register is provided to store the state of each lane in the processor.

For any non-leaf function, the compiler is responsible for scheduling instructions to store each of the *RT* registers in the function prologue and instructions to restore each of the *RT* registers in the function epilogue. Since these are not general-purpose registers, the ISA has special store (*swrt*) and load (*lwrt*) instructions to save and restore *RT* registers, respectively. An additional store-active-status (*sas*) and load-active-status (*las*) instruction is also provided for the *PLS* register.

Lanes that are never enabled in a function never need to load or store an RT register. Since restoring the RT register to the previous state is handled by callee functions, if a callee modifies the RT register of a lane that is inactive in the caller function, the register will be restored to the proper state once a call occurs. The RT register restoration being handled by the callee means that caller functions do not need to know the total width of the processor, and do not need to store or load values for lanes the function does not activate.

The inverse of the *call* instructions is the (*return*) instruction, which is typically the last instruction executed in each function. In the SLA ISA, the *return* instruction indicates for all lanes to restore their PC and their active status using the *RT* and *PLS* registers.

Since some library functions execute jump instructions to other functions and analysis of some of the library code is difficult, we also provide a *callm* instruction, short for call-muted, which behaves identically to a normal *call* instruction, but does not update *RT* or *PLS* registers.

4.3.2 Creating and Halting Instruction Streams

Since call instructions mark nonzero lanes as *inactive*, functions are scheduled with *colane* instructions to activate new instruction streams. There can be as many streams as there are lanes supported by the processor, or fewer if the function does not have enough parallelism to fully utilize the processor's resources. To minimize required changes in the BTB to support *colane* instructions, we enforce that these instructions can only be scheduled in lane 0, similar to branches. Each *colane* instruction specifies the instruction stream's starting offset and the lane assigned to handle the instruction stream. Note that we allow a *colane* instruction to activate a lane that can be immediately suspended with a *nop.sr* instruction.

We also introduce a *halt* instruction. This instruction marks the lane in which it resides as *inactive*, disabling it for the rest of the function execution. This instruction always has the ".sr" bit set, but also halts the lane, so the lane is not resumed on all further lane 0 ".sr" bits. Note that lane 0 cannot use this instruction and treats a *halt* as a *nop* in case of mispredicted control flow. Setting a lane to inactive frees up instruction space and reduces the number of times a lane is woken up, allowing for power savings.

4.3.3 Scheduling Branches and Jumps

A conditional branch or unconditional jump only contains enough information to affect the PC of the lane in which the transfer of control resides. Since branches and jumps can only be issued in lane 0, they behave identically in that lane compared to standard processors. For simplicity, we have each lane follow the same control flow, meaning that nonzero lanes do not need to perform the redundant truth operation for conditional branches. Instead, we provide an instruction that allows for nonzero lanes to update their PC when a branch or jump is taken in lane 0. The prepare-branch (*pb*) instruction takes a PC-offset and no other arguments. Before a branch, each lane other than lane 0 must execute a pb instruction to prepare the target destination of the lane prior to the lane 0 branch. When it is fetched, it sets a register with the destination called the PB register. Upon lane 0 taking a branch, whether conditional

or unconditional, each lane that is active or suspended sets the lane PC to be equal to the destination stored in the PB register.

```
Lane 0   Lane 1   Lane 2   Lane 3
i0.1     i1.1     pb LN_2  pb.sr LN_3
i0.2     pb LN_1  i2.1.sr  -
j LN_0   i1.2     -        -
...                 ...
```

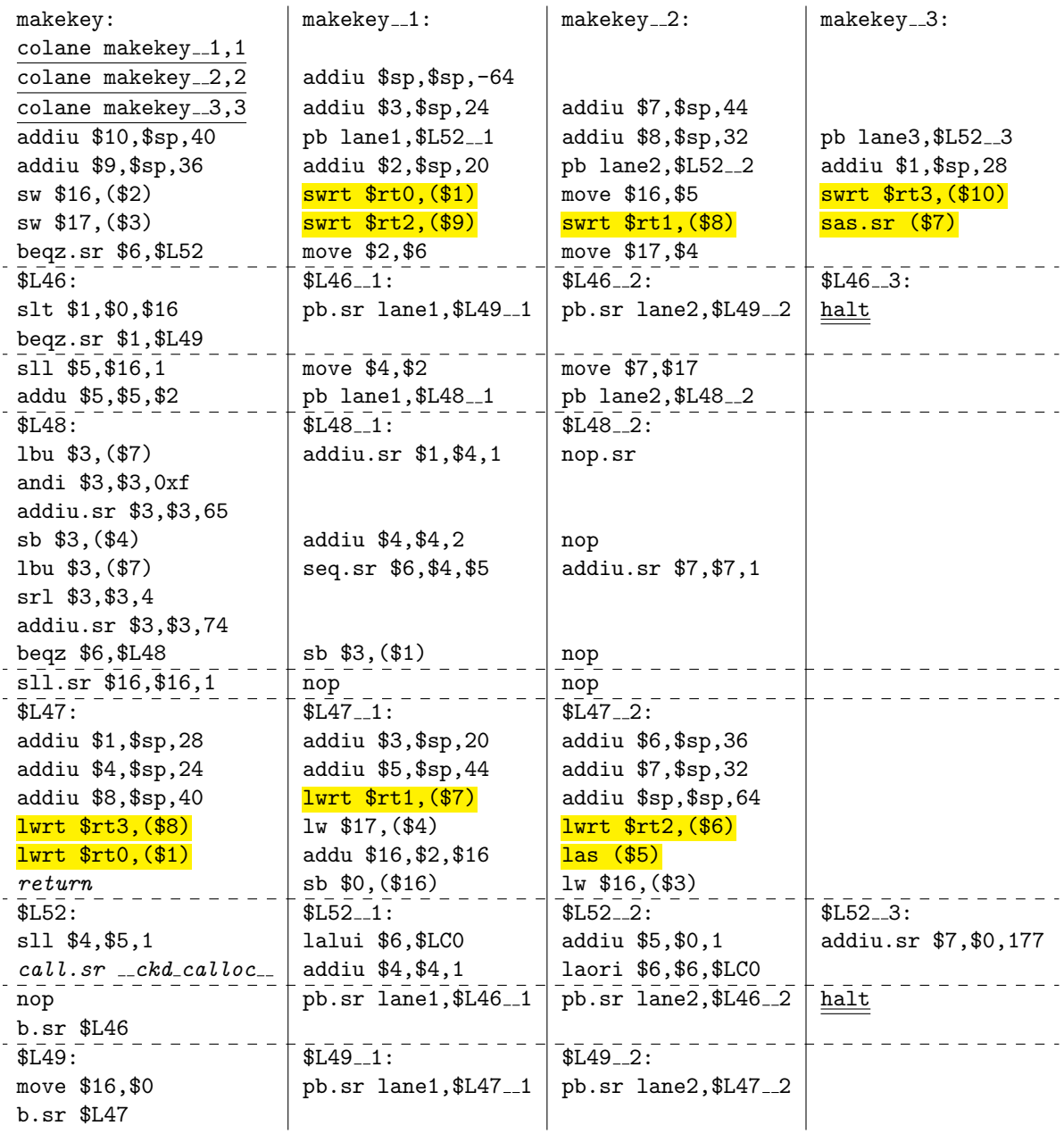
Figure 4.3: An example of *pb* instruction Scheduling

Since *pb* instructions are independent of other instructions, they can be scheduled any time prior to the branch in lane 0, as shown in Figure 4.3, Allowing for *pb* instructions to be scheduled prior to a branch also lifts the requirement for all lanes to be active at the time as the branch or duplicating branches for each lane.

4.3.4 Code Generation Strategy for an SLA Processor

Traditional MIPS processors support a *displacement addressing* mode for loads and stores. In order to initiate a long latency memory operation early in a pipelined data path, modern dynamic processors dynamically break a load (or store) with the displacement into two machine instructions[16]. The first instruction computes the effective memory address and the second performs the data memory access.

The SLA processor also exploits this feature by (1) supporting a *register deferred addressing* mode only for loads or stores, and by (2) requiring the code generator



```
makekey:                makekey__1:             makekey__2:             makekey__3:
colane makekey__1,1
colane makekey__2,2     addiu $sp,$sp,-64
colane makekey__3,3     addiu $3,$sp,24         addiu $7,$sp,44
addiu $10,$sp,40        pb lane1,$L52__1        addiu $8,$sp,32         pb lane3,$L52__3
addiu $9,$sp,36         addiu $2,$sp,20         pb lane2,$L52__2        addiu $1,$sp,28
sw $16,($2)             swrt $rt0,($1)          move $16,$5             swrt $rt3,($10)
sw $17,($3)             swrt $rt2,($9)          swrt $rt1,($8)          sas.sr ($7)
beqz.sr $6,$L52         move $2,$6              move $17,$4
-------------------------------------------------------------------------------------
$L46:                   $L46__1:                $L46__2:                $L46__3:
slt $1,$0,$16           pb.sr lane1,$L49__1     pb.sr lane2,$L49__2     halt
beqz.sr $1,$L49
-------------------------------------------------------------------------------------
sll $5,$16,1            move $4,$2              move $7,$17
addu $5,$5,$2           pb lane1,$L48__1        pb lane2,$L48__2
-------------------------------------------------------------------------------------
$L48:                   $L48__1:                $L48__2:
lbu $3,($7)             addiu.sr $1,$4,1        nop.sr
andi $3,$3,0xf
addiu.sr $3,$3,65
sb $3,($4)              addiu $4,$4,2           nop
lbu $3,($7)             seq.sr $6,$4,$5         addiu.sr $7,$7,1
srl $3,$3,4
addiu.sr $3,$3,74
beqz $6,$L48            sb $3,($1)              nop
-------------------------------------------------------------------------------------
sll.sr $16,$16,1        nop                     nop
-------------------------------------------------------------------------------------
$L47:                   $L47__1:                $L47__2:
addiu $1,$sp,28         addiu $3,$sp,20         addiu $6,$sp,36
addiu $4,$sp,24         addiu $5,$sp,44         addiu $7,$sp,32
addiu $8,$sp,40         lwrt $rt1,($7)          addiu $sp,$sp,64
lwrt $rt3,($8)          lw $17,($4)             lwrt $rt2,($6)
lwrt $rt0,($1)          addu $16,$2,$16         las ($5)
return                  sb $0,($16)             lw $16,($3)
-------------------------------------------------------------------------------------
$L52:                   $L52__1:                $L52__2:                $L52__3:
sll $4,$5,1             lalui $6,$LC0           addiu $5,$0,1           addiu.sr $7,$0,177
call.sr __ckd_calloc__  addiu $4,$4,1           laori $6,$6,$LC0
-------------------------------------------------------------------------------------
nop                     pb.sr lane1,$L46__1     pb.sr lane2,$L46__2     halt
b.sr $L46
-------------------------------------------------------------------------------------
$L49:                   $L49__1:                $L49__2:
move $16,$0             pb.sr lane1,$L47__1     pb.sr lane2,$L47__2
b.sr $L47
```

Figure 4.4: Example of SLA generated code

to statically split every memory instruction to use a register deferred memory mode instead. Deferred addressing allows for memory instructions to access the data cache at the same time the rest of a bundle performs computations in execution units. We

developed a post-assembly optimizer that includes various optimizations including VLIW scheduling, which is required for high quality SLA code generation.

We first generate 32 bit MIPS assembly code using gcc with the -03 flags. We then perform VLIW scheduling to generate packs of VLIW operations for our target-width SLA processor. For each function, we then perform transformations on the assembly code as described in the rest of this subsection to generate our final SLA code.

As a case study for SLA code generation, consider the general layout of a 4-lane function in Figure 4.4, taken from the function `makekey` in the 482.sphinx3 benchmark from spec2006. In our study, we allow the code to schedule floating-point and memory operations anywhere, but similar to a VLIW processor, SLA processors can allow for asymmetric lanes where certain capabilities are limited to specific lanes, such as memory ports only being accessible to instructions in lanes 0 and 1. The actual number of lanes is determined by the largest width of the VLIW pack without *nops* in the scheduled code. For `makekey`, the maximum width is 4. Vertical lines denote each lane, and instructions are aligned to show when they will execute in a pack together, so blank spots indicate when a lane is either suspended or inactive.

We also take steps to always generate conditional branches to compare a single register to zero (e.g. *beqz* or *bnez*) due to fewer available instruction opcodes. We then change all function calls and returns to *call* and *return* instructions. The *call* and *return* instructions are *italicized* in Figure 4.4.

In order to create a separate instruction stream per lane, each nonzero lane is assigned unique labels by duplicating labels in the lane 0. Each lane's target labels are a copy of the label, with an extension of __#, where # is the lane number to provide a unique destination in each instruction stream.

Next, we insert lane specific *pb* instructions prior to jumps or branches in lane 0, with each PC reflecting the lane-duplicated label in tandem with the destination of lane 0. Duplicated labels and inserted *pb* instructions are marked in **bold**. Each *pb* instruction can be scheduled anywhere within the basic block of the same lane, as long as it is prior to the branch instruction in lane 0. As a minor loop transformation, each *pb* instruction in a single basic block loop is moved out to its loop preheader as being shown in the loop at label $\$L48$ in Figure 4.4.

Any time prior to the first call of the function, there must be an *swrt* instruction scheduled for each lane to store the return PC for that lane. However, only a single *sas* instruction is performed for the entire processor, since active information requires only two bits per lane, allowing for all lanes to store the state in a single word (up to 16 lanes in a 32-bit word). For each function with a call, the epilogue prior to a return must contain *lwrt* instructions executed to restore the state of each lane. Similarly, there must be a single *las* instruction to restore the lane active status for each lane. Since `makekey` has a *call*, we insert stores (*swrt/sas*) to save the current values of the *RT* and *PLS* registers to its stack frame at the function prologue, and

loads (*lwrt/las*) to restore the values at each function epilogue (return block). The inserted instructions for saving and restoring *RT* and *PLS* registers are highlighted in Figure 4.4.

After making changes described above, we reapply VLIW-style scheduling using the maximum detected width of the function. Since all lane instructions simultaneously execute, even during control flow changes, the *sb* and *lw* instructions scheduled in lanes 1 and 2 alongside the *return* instruction in Figure 4.4 will execute prior to any instructions at the return point in the lane 0. Similarly, the instructions scheduled alongside the *return* will execute before any instructions in the callee.

We next insert *colane* instructions at the function entry to provide the starting address of each lane within the function. With the exception of lane 0 (the master lane), we need to insert a *colane* instruction for each lane in which one or more instructions are scheduled within the function. If we detect limited parallelism, we can limit the number of *colane* instructions. Figure 4.4 shows colane instructions as underlined.

We then introduce proper synchronization when lanes need to be suspended or resumed. We insert *sr* bits in lane 0 when there are more instructions in the next pack, and can contract processor width using *sr* bits in those packs. When a lane is started, but does not have a ready instruction to execute, we insert a *nop* instruction to maintain lane alignment. The *nop* can also have an *sr* bit set. In a worst case, the packs with nops behave as the same as a single full-width VLIW pack.

We follow this step by detecting where to insert *halt* instructions. We perform a simple analysis to detect when a lane will no longer perform any instructions outside of those required for synchronization. During this analysis, we omit the function epilogue, since epilogue instructions can be re-scheduled into other lanes. If no further useful instructions are detected, we place a *halt* instruction (marked with double-underlined instructions in Figure 4.4) in the lane to disable unnecessary *sr* bit reactivations. We eliminate instructions in those lanes, and shift any remaining computations that are assigned to that lane in the function epilogue into other lanes.

An overview of all introduced synchronization instructions are in Table 4.1, with a definition of of each architectural effect.

4.4 Architectural Design of an SLA processor

4.4.1 The SLA Frontend

The SLA processor is designed to be similar to several simple 32-bit pipelined processors working in conjunction. As is shown in Figure 4.5, each frontend is similar to a simple pipelined processor. Any new registers are defined in Table 4.2. Lane 0 behaves nearly identically to a typical pipelined processor, but sends a message to other lanes when calls, returns, and branches are encountered, as well as when

Instruction	Arguments	Description
call/callr	target_address or address_register	Sets L0 PC to the target_address, halts all other lanes, and saves the PC and status of each lane. Each lane PC is stored in a separate RT register. Each lane active status in a PLS register.
callm	target_address	A call instruction that does not save lane states.
return		Restore the PC from each corresponding RT register and restores status from the PLS register.
swrt	rt_reg, store_address_reg	Stores the value of rt_reg using store_address_reg.
lwrt	rt_reg, load_address_reg	Loads the value using load_address_reg into rt_reg.
sas	store_address_reg	Stores the value in the PLS register at the address in store_address_reg.
las	load_address_reg	Loads the value at the address in load_address_reg into the PLS register.
colane/fork	lane_id, target_address	Sets lane of lane_id to active, and starts it's PC at target_address.
halt		Halts the lane executing the instruction, does nothing in lane 0.
pb	branch_offset	Sets the PB register for a nonzero lane. When lane 0 takes a branch, each lane sets PC to the value in the PB register.

Table 4.1
SLA Synchronization Instructions

instructions with *.sr* bits are fetched from the instruction cache. Nonzero lanes are also similar, but do not need to access the BTB. Instead, nonzero lanes simply access a PB register which is updated via *pb* instructions, and takes directional information from Lane 0.

The processor also expands the BTB to support *colane* instructions. A colane instruction is a transfer of control flow for another lane, and to prevent waiting for a lane to decode the instruction, colane instructions update the BTB, storing the target lane

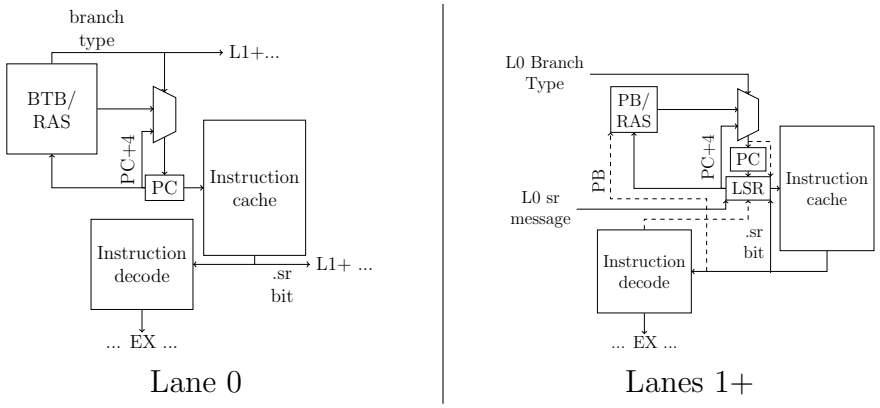


Figure 4.5: The SLA Frontend Design for Lane 0, and Lanes 1+

Register/Use	**Size**	**Location**
PB Prepare Branch Address	PC Size	Each Nonzero Lane
LS Lane Status	2 Bits	Each Lane
RT Return Address	PC Size	Register File (Per Lane)
PLS Previous Lane Status	2 Bits per lane	Register File

Table 4.2
New SLA Registers

and target address when relevant. To prevent the requirement of larger BTB entries or a dedicated structure, we limit a single colane instruction to the entire pack.

Note that unlike a traditional VLIW, in which any operation in the wide instruction can potentially contain branches or jumps, all transfers of control occur in lane 0. This simplifies BTB accesses since only lane 0 in the SLA processor needs to access the BTB. Aside from the normal BTB structures, the BTB needs to have two bits to indicate if the instruction in the BTB is a colane and the target lane number.

The return address stack (RAS) in each lane also behaves similar to a normal processor, but needs to also store previous lane state along with the destination PC.

Bit Configuration	**LSR State**
0 0	Inactive. Suppress lane 0 wakeup requests.
1 0	Suspended. Wake up on lane 0 sr requests.
1 1	Active. Execute Instructions.

Table 4.3
LSR States

To maintain the state of each lane, we introduce a lane status register *(LSR)*. The LSR stores two bits. Table 4.3 shows the valid states. The most significant bit is responsible for suppressing requests to resume execution, while the least significant bit is responsible for lane active status. When the LSR active bit is 0, the entire lane is suspended.

When nonzero lanes receive a branch type from lane 0, they may also modify the LSR in their lane. When a *call* instruction is encountered, the value of the LSR is set to inactive. When a *return* instruction is encountered, the value of the LSR is updated by the values stored in the lane's RAS. The LSR is also updated to active and enabled when a colane instruction is sent from the BTB. If a *halt* instruction is decoded, the LSR is set to 00 in that lane. Since the *halt* instruction always has the *sr* bit set, the lane first sets the LSR to suspended when the halt is fetched, then the lane is set entirely to inactive once the instruction is decoded.

The instruction caches of SLA processors are separated, so the SLA processor has

separate tags associated with each operation in a pack. Since the instruction caches of the SLA processor are at an operation granularity, it has similar overhead to a more traditional processor. The VLIW, however, only has one tag associated with an entire wide instruction. Since the VLIW processor tracks fewer overall PCs due to the use of wide instructions, it uses fewer tags in its instruction cache than other processor designs.

4.4.2 Supporting Single-Cycle PB Updates

When a branch instruction immediately follows a PB instruction, the PB instruction would be decoded at the same point that the BTB is being accessed, meaning the PB register is stale by the point of a control flow change from a jump or taken branch.

To ensure the PB register is correct by the end of the fetch stage, we propose predecoding *pb* instructions at the point of insertion into the instruction cache. When a line is filled in the L1 instruction cache for a nonzero lane, we examine the opcodes of the instructions loaded in for any *pb* instructions. When one is detected, we can add the PC of the instruction to the offset field, leaving the instruction predecoded in the instruction cache.

To mark predecoded instructions, we add a bit to the instruction cache called the PB-target *(PBT)* bit. Since PB instructions always target the same address, when a

PB instruction is loaded into the instruction cache, we add the PC of the instruction to the offset, and store it in the 31 lower bits to preserve the *sr* bit of the instruction. The most significant bit can be reconstructed using the PC when the instruction is fetched. When a PB instruction is fetched, the *sr* bit is handled the exact same way as all other instructions, but the remaining fetched data is simply moved into the PB register for that lane.

Lane 0 does not need to implement the PBT modification, and does not need to have the same pre-decode step when performing an L1 cache line fill, since PB instructions are only placed in nonzero lanes.

4.4.3 The SLA Processor Backend

Unique to the architecture are two other required registers that do not reside in the frontend, and are defined in Table 4.2. These two registers are the (PLS) register and the (RT) registers.

These registers do not need to reside in the front end. Since they are not speculatively updated, they can be updated similar to the return register for a jump-and-link instruction. The PLS register stores the concatenation of all previous lane status values upon a call instruction and is designed to maintain data for the processor to use upon a return if the RAS is insufficient. Similarly, the RT register is responsible

for maintaining previous lane PCs. Since the LSR works in conjunction with the PC, the processor must restore both the PC and LSR upon rollback. The PLS and RT registers inform rollbacks when a return is mispredicted.

Much of the remaining pipeline design closely resembles a standard VLIW processor, since instructions proceed lock-step through the rest of the pipeline.

4.4.4 Supporting Speculation

Speculation can be supported in a similar fashion to a normal processor, but special considerations must be made to support speculation in the face of speculative lane active status updates. Upon a rollback, each lane must restore both the lane PCs and states, since lanes could have been enabled or disabled on a wrong path.

LS registers can be repaired using the state of the processor at EX using similar mechanisms to PC, but lane PCs need to be calculated with the repaired LS states. If a load encounters an exception, a similar step needs to be taken to ensure the pack properly replays. Similar to a normal processor, program counters need to be rolled back to the state that the prediction is made, including verifying suspended and halted lanes did not do so speculatively.

However, PB registers do not need to be restored. Since PB instructions are always

scheduled prior to the next branch or jump instruction, the PB registers will be correct by the time they are read and used, even when incorrect PB targets may have speculatively written to those registers.

Similarly, the *RT* and *PLS* registers are not speculatively written to, so they do not need additional hardware support for speculation.

4.4.5 Variable-Width Behavior in SLA

By providing explicitly started instruction streams and supporting colane design, the SLA processor is capable of behaving like a narrower-width processor. If a wider SLA processor is provided with code compiled for a narrower SLA processor, the program will function as though the wider SLA processor were the same width as the processor targeted by the compiler.

Explicit activation of lanes enable the SLA processor to naturally support backwards compatibility with lower-width processors, which requires additional engineering for standard VLIW processors, since PC increments in VLIW processors vary based on the width of the wide instruction.

While the SLA processor is underutilized during these parts, the SLA processor natively supports narrower width code with no modifications. VLIW processors also

underutilize their hardware when executing narrower code, but also need ways to modify PC increments, as well as structures to align packs when widths do not match.

4.4.6 Expanding BTB Capabilites

While not evaluated in this work, our processor does not need to necessarily impose the lane 0 restriction on branches. Simply, we propose this to maintain simplicity in our baseline design. At the cost of a slightly more complex BTB, we can allow nonzero lanes to contain branches.

The primary motivation for allowing nonzero lanes to execute instructions is to enable functionality of partially committing packs. Instructions in lanes prior to the branch can commit, while instructions in lanes after the branch can be filled with instructions in the not-taken path, and are discarded if the branch is computed taken in the execution step of the pipeline.

While it is possible for VLIW processors, including our baseline VLIW, to contain multiple branches per pack, many state-of-the-art VLIW processors still only allow for one branch to be predicted in the BTB per instruction pack, such as the currently in-production TigerSHARC TS-101S[2, ch.8 p.44]. This restriction allows the BTB entry to be accessed using an aligned instruction address and removes the need for multiple BTB accesses per pack.

Similarly, we can access the BTB using only the PC of lane 0. Since the BTB stores both a lane identifier for `colane` instructions as well as a mechanism to move the stored address to that lane, the primary modification is an addition of a *PB* register to lane 0 and mechanisms to squash instructions in lanes that follow lane 0.

Upon a BTB hit that predicts taken, the lane that corresponds with the BTB entry's identifier adopts the value of the target in the BTB. The targets of each other lane are computed using the PB registers for that lane. At the cost of some complexity and energy, allowing instruction packs to partially commit offers more flexibility for compiler scheduling, potentially improving performance.

4.5 Methodology

Fundamentally, the objective of this design is to propose a new processor. To measure performance, we are aiming to categorize characteristics versus a baseline VLIW processor. The team at Florida State University developed an assembly optimizer to perform transformations at the assembly level for this project. Use of an assembly optimizer allows us to leverage all the optimizations performed by a mature compiler like *gcc* while being able to perform low-level transformations in a much simpler infrastructure. We first generate MIPS assembly code using *gcc*. The assembly optimizer next expands pseudo assembly instructions that comprise of more than one

machine instruction to equivalent assembly instructions so that there is a one-to-one mapping between assembly and machine instructions. At this time, the assembly optimizer also translates instructions to the new SCALE instruction set. We also obtain information from *gcc* indicating which registers are passed as arguments at the point of each call and which register(s) are used as a return value at the point of each return. This information allows the assembly optimizer to exactly know which registers are live at each instruction, then schedules the assembly instructions as described in Section 4.3.4.

We use the Fast/ADL simulator [28] to simulate both a conventional SCALE VLIW processor as the baseline and a modified processor supporting the SLA approach. These processors use variants of the SLA ISA and are implemented as faithful cycle-accurate simulators. In ADL, the instructions are loaded from an instruction cache, data values are loaded from the data cache, values are obtained through an explicitly simulated pipeline, branch target addresses from the branch target buffer are actually used, etc. In other words, these simulators are not simple emulation and timing implementations.

Table 4.4 shows other details regarding the processor configuration that we utilize in our simulations. We support a separate L1 instruction cache for each lane to allow simultaneous instruction fetches. The total L1 instruction cache space for all lanes is 32KiB, but the SLA instruction cache space is split among dedicated instruction

Branch Predictor	4096 entry direct-mapped BTB GShare predictor w/ 17-bit branch history.
Page Size	4KB
L1 ICs	32KiB total, 64B line size, 4-way, VLIW: 11 cycle miss pen. SLA: 12 cycle miss pen.
L1 DC	32KiB, 64B line size, 4-way
DTLB	32 entries, fully associative
L2 Unified Cache	1MiB, 64B line size, 16-way, 80 cycle miss pen.
5 stage integer pipeline	IF, ID, RF, EX, WB

Table 4.4
Processor Configuration

caches for each lane. In our simulations, the L2 cache does not have dedicated ports for each L1 instruction cache, so L1 icache requests are pipelined and other caches must wait if another cache is occupying the L2 cache ports. These delays can be reduced by adding more ports into the cache at the cost of higher energy usage, but this was not modeled in our experiments.

We support a 5 stage integer pipeline, where the memory reference for a load instruction is executed in the EX stage due to no displacement with loads. While more modern processors use deeper pipelines, both processors are affected in the same way. In both the baseline VLIW and the SLA processors, misprediction penalties are increased by each stage that occurs prior to the misprediction detection, and latencies increase with each additional stage added to the processor as a whole.

We test two 4-wide SLA processors and two 8-wide SLA processors. The processors are identical, aside from the lanes and their instruction cache configurations. We test

Description	L0	L1	L2	L3	L4	L5	L6	L7
SLA-4x8k	8	8	8	8	-	-	-	-
SLA-16-8-4-4	16	8	4	4	-	-	-	-
Wide-SLA-8x4k	4	4	4	4	4	4	4	4
Wide-SLA-Asym	8	4	4	4	4	4	2*	2*

Table 4.5
KiBs allocated per SLA Lane Instruction Cache
*2KiB caches have 32 Byte lines

both a symmetric and asymmetric design, and caches are set to store a total of 32KiB of instruction cache space. The processor configurations are shown in Table 4.5. The 2KiB caches of the Wide-SLA-Asym configuration are set to be 32-byte lines instead of the 64-byte lines of other caches since these caches are quite small and lanes are often much more frequently disabled.

4.5.1 Ensuring Equivalent Workloads

We ran the simulator on a variety of SPEC 2006 benchmarks on reference inputs. Simulating benchmarks with their full ref inputs takes unacceptably long, so another mechanism to ensure both simulations execute the same specific window in the program is required. Similarly, since both simulators offer fundamentally different instruction spaces, IPC is insufficient to compare performance, since VLIW and SLA both contain many instructions that do not directly contribute to a program's execution, but are needed for correctness.

Instead, to ensure equivalent workloads, we classified each operation of the benchmark

as either *meaningful* or *meta* instructions. In this context, an instruction is a single operation.

We classify *meta* instructions as operations that are not part of a program's core control flow or calculations. These include loads and stores as part of nonzero lanes for RT and PLS registers, and any accompanying add immediate instructions used to calculate their addresses. The load and store for RT registers for lane 0, however, are equivalent, to a load and store of \$31 for the VLIW processor, and thus are not marked as *meta* instructions. We also categorize *pb* instructions as *meta* instructions, since they serve to keep lanes synchronized. Furthermore, we classify any *nop* instructions as *meta* instructions, as VLIW uses these for alignment, and the SLA may occasionally insert these after an sr is broadcast from lane 0. Finally, the lane control instructions of *colane* and *halt* are meta instructions that are unique to the SLA processor.

Instructions that are not meta instructions contribute directly to the completion of the program, and a program can be tracked through the progress of these instructions. We refer to these instructions as meaningful instructions. During the lifetime of a program and function, both the VLIW and SLA processor execute the same number of meaningful instructions. In both cases, actual rate of progress through a program is measured using Meaningful Instructions per cycle (MIPC), which is a representation of $MIPC = Instructions_{meaningful}/Cycles_{total}$. Since we count all cycles but omit meta instructions, if a section of code consists of only meta instructions, the MIPC

for that section would be 0. To verify that this is correct, we tracked the number of function calls for several benchmarks in both the SLA and VLIW processor to ensure they performed the same number of function calls in a given window of meaningful instructions.

In our simulations, we allow each benchmark to warm up for 2 billion meaningful instructions, followed by gathering statistics for 10 billion meaningful instructions. Each benchmark was compiled using *gcc* and passed through our assembly optimizer to generate both VLIW and SLA code with identical optimizations. Expansion from VLIW to SLA was completely mechanical using our assembly optimizer, and no additional code optimizations were performed once the SLA conversion occurred.

When performing power calculations, we use CACTI 7 [4] with 32 nm technology and 1 read port for the L1 caches, with the port being 32 bits for each SLA lane, and 128/256 bits for the baseline 4-wide/8-wide VLIW. Similarly, the L2 cache is simulated with a read/write port instead, but it is otherwise identical between the two processors.

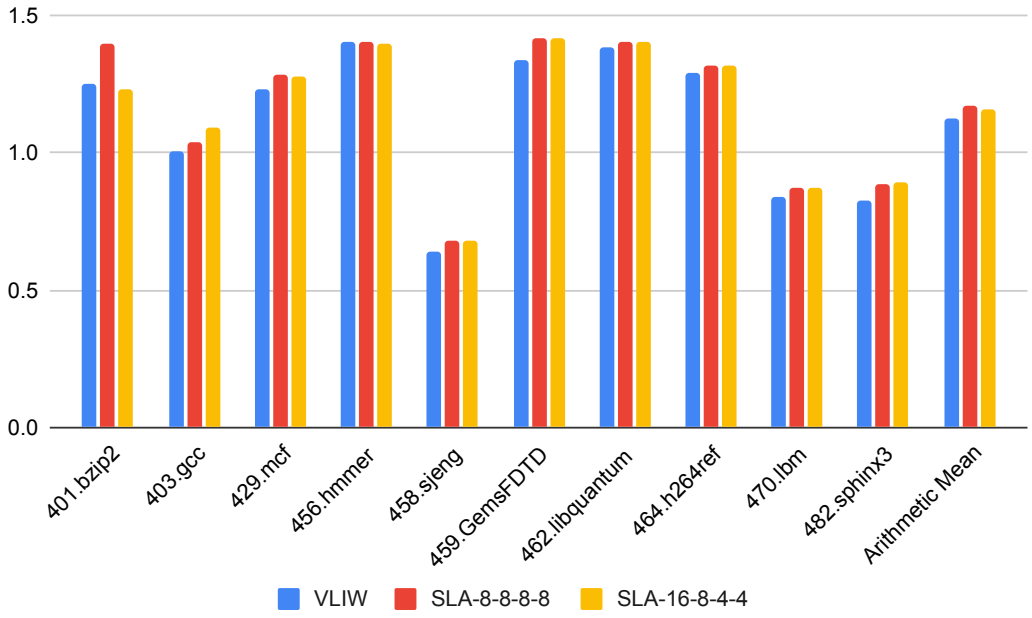


Figure 4.6: Meaningful IPC (Higher is Better)

4.6 Results

4.6.1 Performance Analysis

As shown in Figure 4.6, we find that meaningful IPC is similar to that of a baseline VLIW processor in nearly every benchmark. While the SLA processor introduces extra instructions to the stream, which lowers MIPC, the L1 instruction cache for each lane functions significantly more efficiently. The primary reason performance can be maintained despite an increase in instructions is due to improved L1 instruction cache performance.

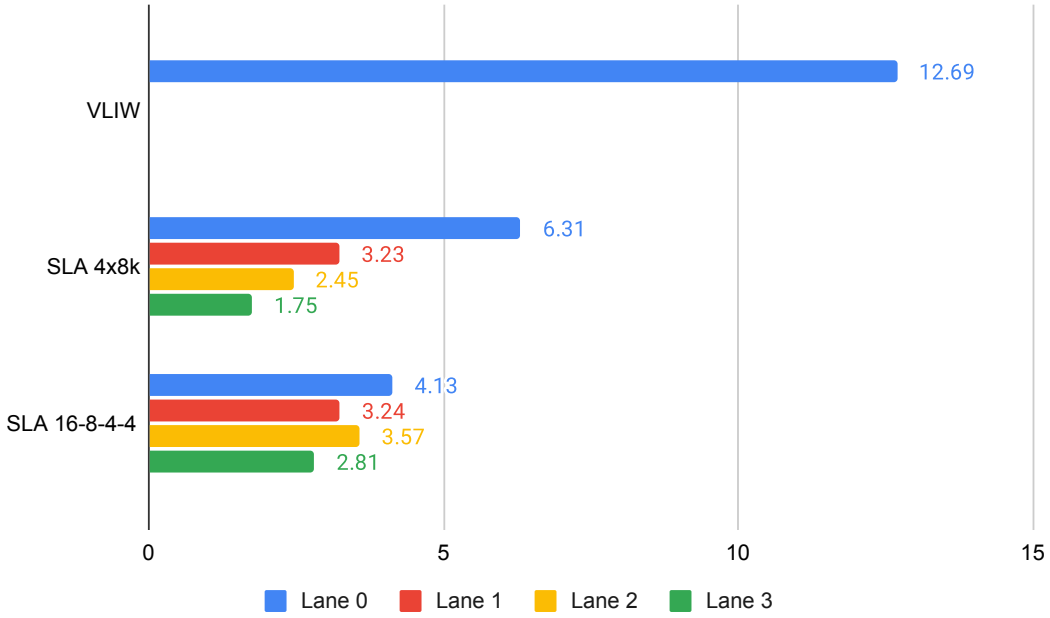


Figure 4.7: Arithmetic Mean Instruction Cache Misses Per 1000 Meaningful Instructions over Spec 06 (Lower is better)

We can see instruction cache performance from Figure 4.7. The L1 instruction cache misses less often for SLA processors than the VLIW. Performance is highly sensitive to instruction cache hit rates, meaning that even marginal improvements can make significant differences in terms of performance. SLA instruction caches do not need to store many *nop* instructions, hence they can hold a larger number of meaningful operations and fewer *meta* instructions. While the total number of misses in SLA processors are marginally higher, across all four instruction caches (13.74 for SLA 4x8k, and 13.75 for SLA 16-8-4-4, many misses occur immediately after a taken branch. During these taken branches, all instruction caches miss, allowing for opportunities to pipeline requests to the L2 cache, reducing the overall instruction cache miss penalty. Also, instructions that trigger a cache miss, but suspend the

lane can begin handling the cache miss without interrupting execution of other lanes. Furthermore, cache lines in lanes that are often suspended can functionally service more packs of operations before a miss occurs, as lanes do not change their program counters when inactive.

Due to improved performance for instruction caches, we find that on average, meaningful IPC is within 0.04% for both configurations of the SLA processor and the VLIW baseline, despite the increase of around 1.2% in total packs executed.

4.6.2 Instruction-Cache Behavior

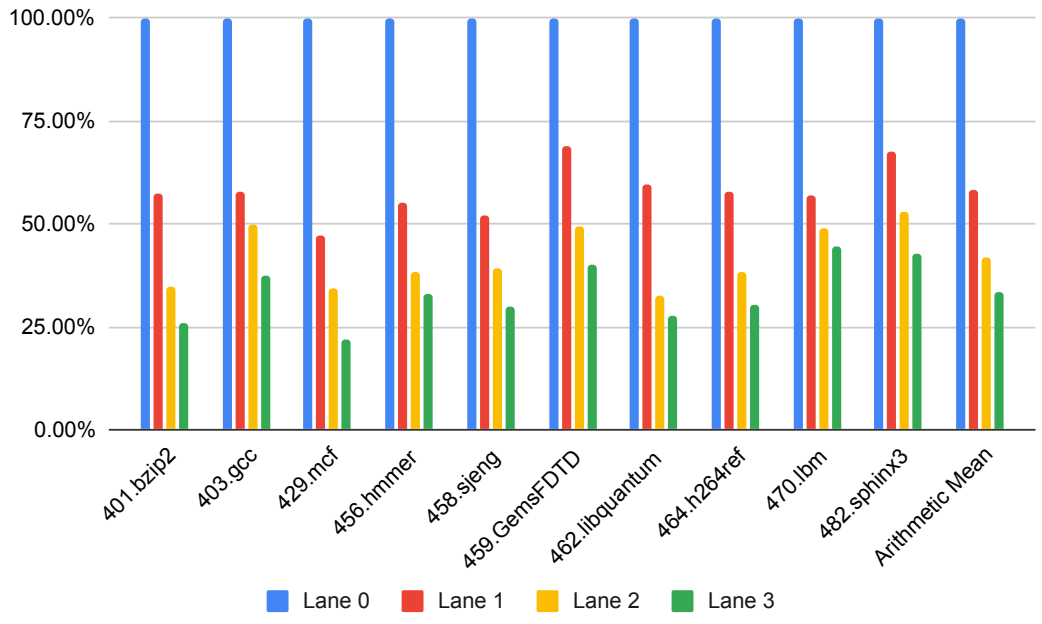


Figure 4.8: SLA 4x8k caches, number of accesses with respect to lane 0

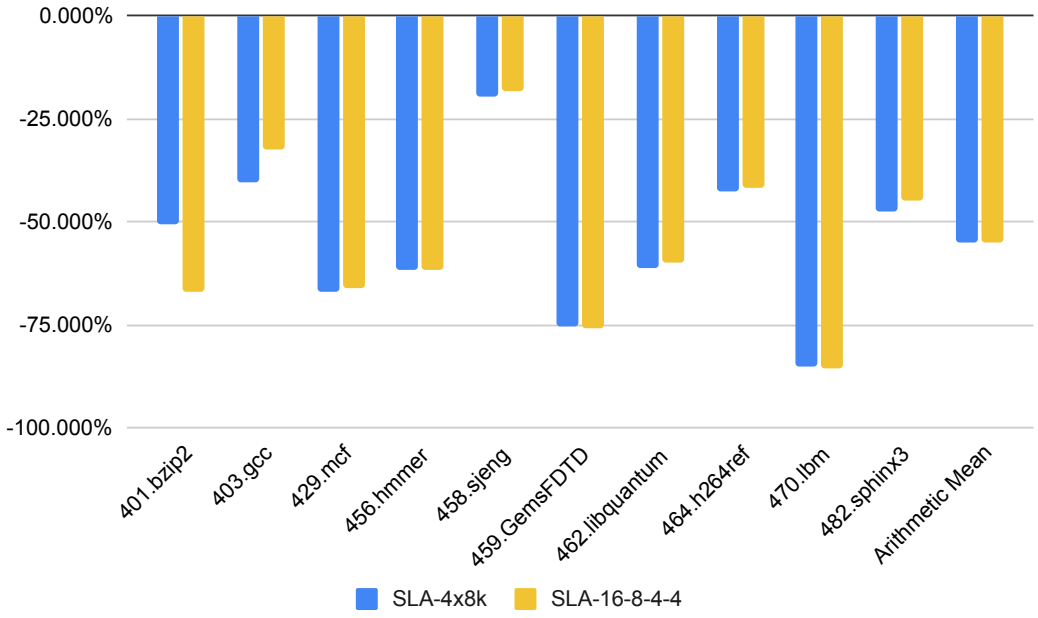


Figure 4.9: Instruction Cache Power Normalized to VLIW (Lower is Better)

Figure 4.8 shows how frequently that each lane accesses the instruction cache for an SLA processor with the symmetric 4-8K cache configuration over the entire benchmark suite. We do not show the SLA 16-8-4-4 configuration since it behaves nearly identically (within 1% for each lane). When a lane is suspended or halted, that lane does not need to issue a cache request. Suspending lanes means that during periods of lower IPC, lane instruction caches can save significant energy on average. What we see is that, on average, lane 3 accesses it's instruction cache 35.97% as often as lane 0. Furthermore, each instruction cache is significantly smaller than the VLIW instruction cache, lowering power requirements for each cache.

Since instruction caches are more lightweight in the SLA processor, we find that the

majority of the time, only one or two of the instruction caches are accessed. As shown in Figure 4.9, this allows for the SLA processor to save significant energy with instruction caches in most cases. 458.sjeng performs most poorly due to bad instruction cache coverage coupled with many small, function calls. When a function is encountered for the first time, the SLA processor needs to perform an L2 instruction cache access for each lane instead of the single access that the VLIW requires. The instruction cache hit rate for 458.sjeng is lowest in the benchmarks, and due to how poorly it caches, the SLA processor needs to more frequently access the L2 cache compared to the VLIW when a new function is encountered. While the SLA processor still saves energy with the L1 caches, the power requirements for additional L2 cache accesses offsets some of the power savings of the L1 caches. However, this is the exception, and not the rule, and the power savings of these instruction caches are otherwise significant, with an average mean reduction in power being 55.17% for SLA-4x8K and 55.29% for SLA-16-8-4-4. The number of bits being read for the cache per access is also significantly larger for the VLIW, so when an instruction cache is off, caches benefit from both searching smaller caches, as well as an effective reduction in the total number of bits being read from the instruction caches. SLA-16-8-4-4 performs marginally better despite on average using slightly more power in lane 0 since there are fewer calls to the L2 instruction cache from lane 0, which typically has the largest lane code footprint.

4.6.3 Lane Power Analysis

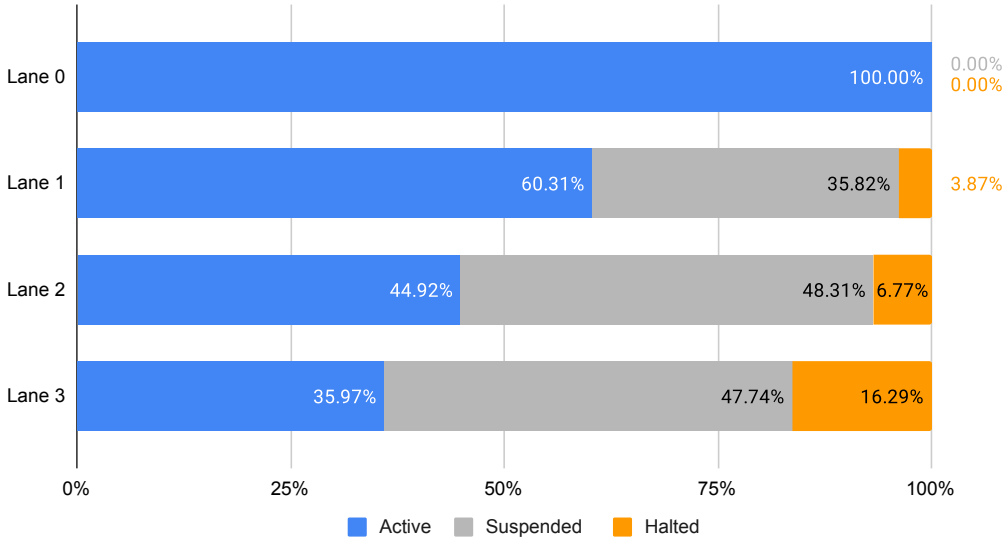


Figure 4.10: SLA 4x8k - Distribution of Lane Status across Spec06

While lane power statistics are more difficult to calculate, we can perform analysis on the state of each lane and time spent in lower power modes. When suspended or inactive, lanes do not need to fetch or decode any operations, providing energy savings beyond the IC benefits. Figure 4.10 shows that lanes 2 and 3 are spent in a low-power state the majority of clock cycles. Similarly, lane 1 only executes, on average, 60.31% of cycles, while suspended or halted the rest of the time. Lanes 2 and 3 execute even less, needing to be active in under half of the total cycles. We use slightly more power when all lanes are active, but find our energy usage is much better when any lanes are turned off. Overall, the lane utilization of the entire processor occurs under 35.97% of the time, meaning that the lane configuration is capable of

significant power savings. Almost 40% of cycles involve only lane 1 actively working with full power and all other lanes maintain a lower power state.

4.6.4 SLA Scalability

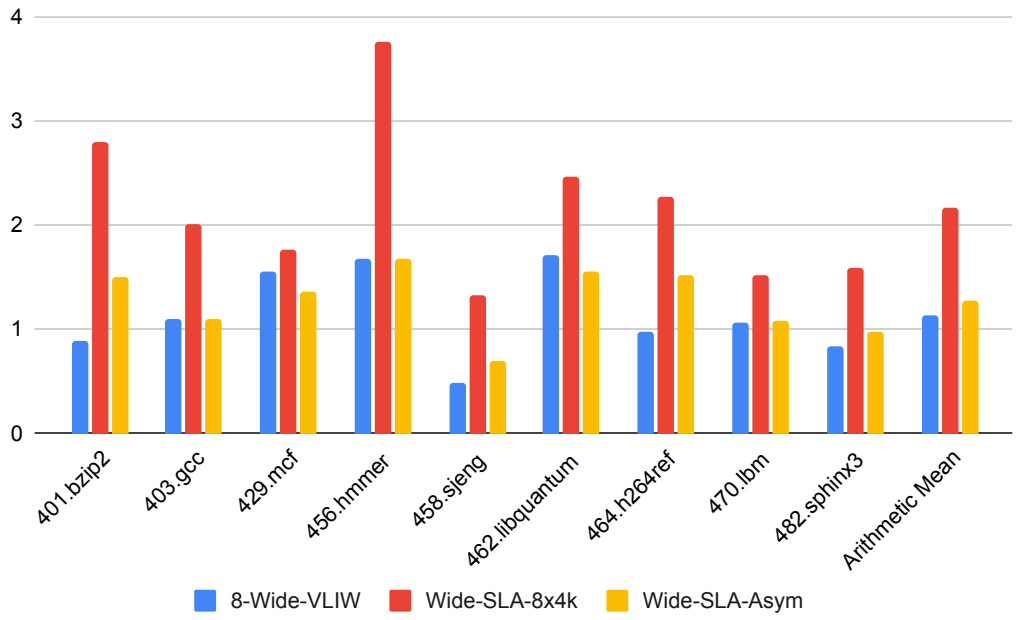


Figure 4.11: 8-Wide Meaningful IPC

When the width of the processor is expanded, the SLA processor scales better than same-width VLIW processors. A comparison of MIPC is shown in Figure 4.11. We find that as the width of the processor grows, the SLA processor performs better with respect to MIPC than the VLIW. Since the discrepancy of lane sizes is slightly higher for the wider-width processor, we also see that the performance of the asymmetric configuration is better than the symmetric configurations, since some performance is

lost by lane 0 instruction cache misses. When comparing IPC to the 4-lane configuration, 8-wide processors perform slightly worse due to the nature of general-purpose benchmarks.

During points of limited IPC, the VLIW processor performs worse at higher widths since instruction cache hit rate dramatically decreases, and increased parallelism does not compensate for the increase in *nop* instructions. For instance, in 401.bzip, the 4-wide VLIW has a 98.72% instruction cache hit rate, but hit rate drops in the 8-wide VLIW to 91.58%, which is a 7.14% drop in hit rate. However, the hit rate of the asymmetric SLA processor is impacted significantly less than a VLIW. In SLA-16-8-4-4, the lane 0 hit rate of SLA-16-8-4-4 in 401.bzip2 was 99.96%, while the lane 0 hit rate of Wide-SLA-Asym only decreased to 98.28%, a 1.68% drop in hit rate. All other caches for the asymmetric cache configurations degraded less than lane 0, meaning the SLA processor is able to maintain performance even during periods of low IPC.

Since *nops* are not needed to be stored in caches, we find that instruction cache performance scales significantly better than VLIW processors. Figure 4.12 demonstrates the instruction cache power usage of an 8-wide VLIW vs SLA configurations. All SLA configurations, including the 8-wide configurations outperform 4-wide VLIW on average (shown as 0% in the figure), though the symmetric 8-wide configuration does not perform quite as well as the asymmetric version due to more L2 cache accesses

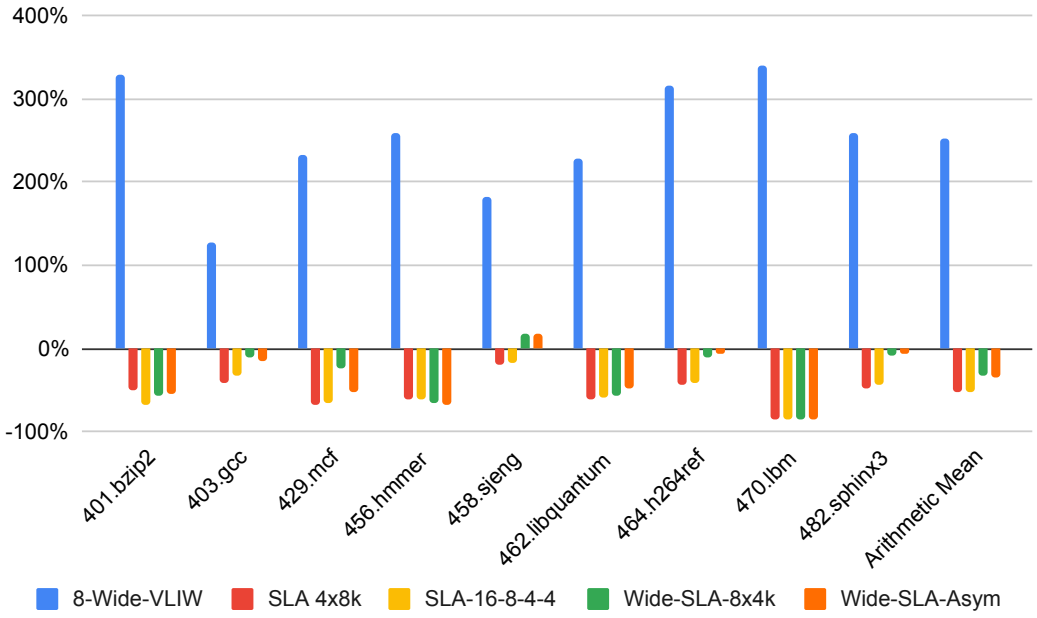


Figure 4.12: 8-Wide Cache Power Statistics (Normalized to 4-wide VLIW)

from lane 0. While Wide-SLA 8x4 requires 20.35% less instruction cache energy than the baseline 4-wide VLIW, Wide-SLA Asym is able to perform better by using 42.95% less instruction cache energy than the baseline 4-wide VLIW, almost as well as the 4-wide SLA configurations despite having more lanes.

4.6.5 Binary Encoding Size

Another notable byproduct of eliminating nops compared to VLIW processors are smaller binary sizes. In smaller systems where storage is at a premium, smaller code size can impact cost. For instance, embedded systems often have the binary on ROM

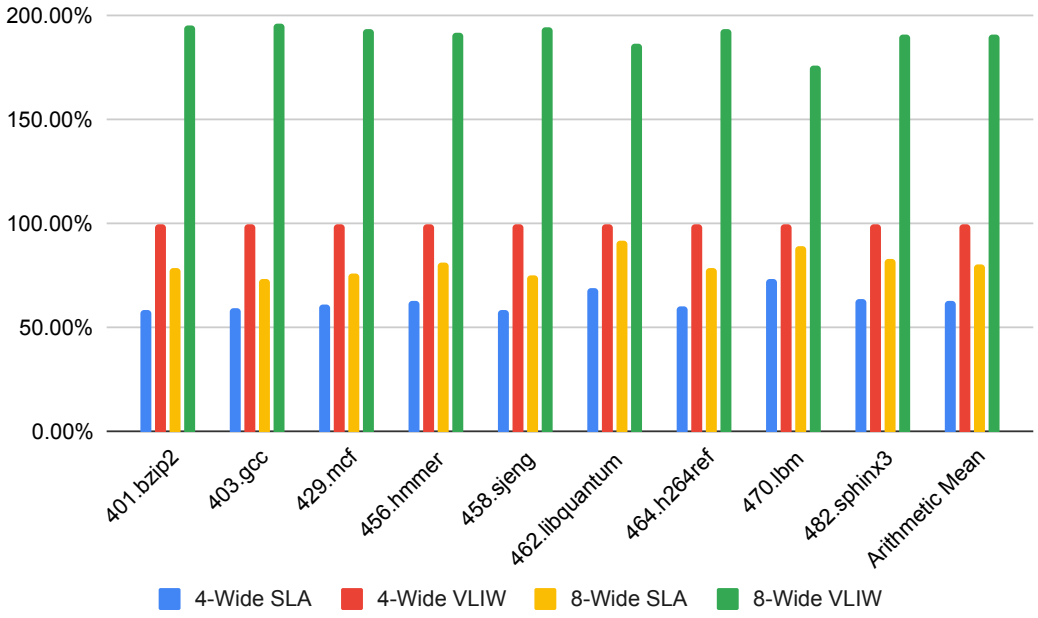


Figure 4.13: Number of Encoded Operations (Normalized to 4-Wide VLIW)

chips and decreasing code size could decrease the size of ROM chips needed, which can reduce cost. Figure 4.13 shows the number of operations required to encode the binaries for each of the benchmarks, excluding libraries. We find that the 4-wide SLA processor only needs to encode 63.50% of the number of operations compared to the same binary for a 4-wide VLIW.

Furthermore, when the size is expanded to 8-wide processor configurations, we find the SLA processor maintains a significantly smaller binary. Larger pack widths introduce more no-operations. However, the 8-wide SLA configuration requires a smaller binary than even the 4-wide VLIW configuration, encoding just 80.74% of the operations compared to the 4-wide baseline. In contrast, the 8-wide VLIW requires 190.85% of

the encoded operations compared to the 4-wide VLIW. This shows that SLA code generation results in binaries that have little code size growth as the number of lanes increases.

Chapter 5

Variable-Width SLA Processor Design

5.1 Introduction

Meta instructions, or instructions that do not directly contribute to the completion of a program and are instead inserted for processor maintainence, must be inserted to begin and maintain each instruction stream in a Synchronized Lane Architecture (SLA) processor. The number of required meta-instructions is directly correlated with the width of the processor. In general, the higher the width of the lanes, the more meta-instructions are needed to maintain synchronization.

Each instruction stream, aside from the initial lane, must be started with a `colane` instruction, which instructs a lane to begin execution, and at which PC. Similarly, at the point of a function call, each lane must save it's lane PC for when the function returns.

Lane 0	**Lane 1**	**Lane 2**	**Lane 3**
colane 1, ...			
colane 2, ...	...		
colane 3, ...	...	...	
...	...	...	...

Figure 5.1: Traditional Function Initialization in an SLA Processor

While both SLA and VLIW representations are able to represent the same program, an SLA processor introduces some meta-instructions in exchange for better instruction cache power efficiency. Since meta instructions are often scheduled in function prologues and epilogues, they can add to the critical path of the program and can offer a significant performance penalty in cases where the code cannot fit in the instruction cache. Figure 5.1 shows the typical initialization for a 4-wide function. If the function is not cached, the highlighted instructions each trigger a cache miss for their respective lane. This means that encountering a new 4-wide function begins with a penalty of four full instruction cache misses. While the SLA processor will take longer to encounter the next cache miss compared to an equivalent VLIW processor, the SLA processor cannot make use of this in functions that are short and infrequent. When smaller functions are frequently encountered that do not already reside in the

instruction cache, the performance of the SLA processor degrades. While packs residing in the L1 instruction cache for each lane tend to perform better than their VLIW counterpart, smaller functions often do not benefit from this. Furthermore, since these small functions have a higher warm-up penalty, the overhead of the SLA approach becomes less negligible.

Lane 0	**Lane 1**	**Lane 2**	**Lane 3**
...	...	...	...
`addiu $1,$sp...`	`addiu $2,$sp,...`	`addiu $3,$sp,...`	`addiu $4,$sp,...`
`swrt $rt0,$1`	`swrt $rt1, $2`	`swrt $rt2, $3`	`addiu $5,$sp,...`
`swrt $rt3,$4`	`sas $5`	...	...
...	...	...	...
`call foo`	...	...	...
...	...	...	...
`addiu $1,$sp...`	`addiu $2,$sp,...`	`addiu $3,$sp,...`	`addiu $4,$sp,...`
`lwrt $rt0,$1`	`lwrt $rt1, $2`	`lwrt $rt2, $3`	`addiu $5,$sp,...`
`lwrt $rt3,$4`	`las $5`	...	...
`return`	...	...	...

Figure 5.2: Function Saving and Restoring in a Traditional SLA Processor, assuming 4 active lanes.

Furthermore, as shown in Figure 5.2, meta overhead can also increase memory pressure in the data cache. Prior to a function call, the return registers must be saved and restored through the use of `swrt` and `lwrt` instructions. An additional store and load must also be performed to store all lane states, which is executed by the `las` and `sas` instructions. In the given case, we find that this can cause unwanted growth, particularly in asymmetric configurations with limited memory ports.

When analyzing short functions, we also find that many smaller functions also have limited opportunity to exploit parallelism, and a small width VLIW processor would

be sufficient to fully handle the processor design. Furthermore, multiple-stream processing is orthogonal to multiple-operation processing.

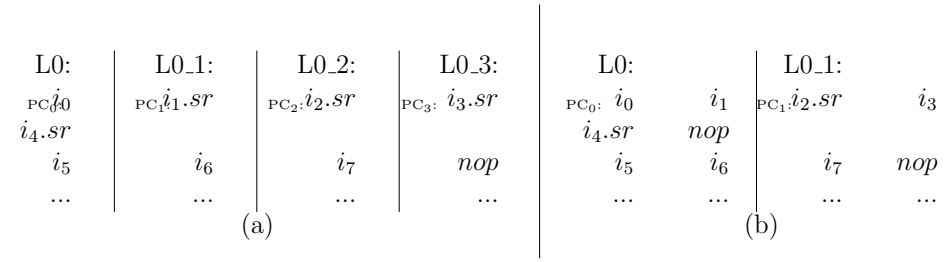


Figure 5.3: Traditional SLA and Hybrid instruction streams.

Compared to the traditional SLA processor instruction stream shown in Figure 5.3(a), we observe that another valid configuration is to allow for SLA processors that consist of VLIW packs of varying sizes, such as what is seen in Figure 5.3(b). In this example, each lane of the SLA processor behaves as a 2-wide VLIW processor. Instead of four lanes, we have two lanes, each fetching two operations per cycle. Unlike the traditional SLA processor, this processor fetches bundles of operations, such that each lane processes two instructions at a time. Lane 0 fetches operations starting at label `L0:`, and lane 1 fetches operations starting at label `L0_1:`. Each lane fetches and processes two operations at a time. The first pack of instructions contains the operations i_0, i_1, $i_2.sr$, and i_3. Lane 0 is responsible for i_0 and i_1, and lane 1 is responsible for $i_2.sr$ and i_3. Since $i_2.sr$ has the sr bit set, the entire lane suspends. Unlike the SLA processor, since i_1 is part of lane 0's execution, it does not have an sr bit. Similarly, i_3 does not need an sr bit, since sr is handled by $i_2.sr$. The next pack contains $i_4.sr$, and since the lane needs to maintain alignment, a nop is scheduled

alongside it. Since the sr bit is set, each other lane, which in this example is just lane 1, is instructed to resume execution. Finally, the last pack of the i_5, i_6, i_7 and a nop are handled similar to the previous examples.

By enabling the SLA processor lanes to each behave as a smaller VLIW processor, we allow for the processor to maintain many of the power efficiency advantages of the SLA while lowering meta-instruction and SLA hardware overhead that comes with increasing processor width, as well as reclaiming fewer tags per operation in the instruction cache similar to traditional VLIW processors. We refer to this architecture as the SLA Variable-Width Design (SLA-VW).

In the SLA-VW approach, lanes can process multiple operations at the same time. Each lane is given a width of N *ways*, where each *way* can handle a single operation. A group of all operations in a single lane are referred to as a *bundle*. In the SLA-VW organization, a *pack* consists of all *bundles* across all lanes.

5.2 SLA-VW Lane Design Considerations

Functionally, an SLA processor is an SLA-VW processor with the added restriction of each lane maintaining a bundle size of 1. We refer to each SLA configuration using the shorthand of L# for each lane, where # is the lane width. So a processor with

a lane 0 that is size 1, and two lanes of size 4 would be an SLA processor with an L1-L4-L4 configuration.

Just as in SLA processors, each lane in an SLA-VW processor is either active, inactive (halted) or suspended. Active lanes fetch instructions and execute the fetched instructions, as per usual. Suspended lanes have a tracked PC, but do not execute instructions until lane 0 encounters an instruction with an sr bit set. Finally, inactive/halted lanes are fully disabled, and suppress any sr requests.

Lane 0	**Lane 1**	**Lane 2**	L0 State	L1 State	L2 State
i1 i2	i3.sr i4	- -	active	active	halted
i5.sr i6	- -	- -	active	suspended	halted
i7 i8	i9 i10	- -	active	active	halted

Figure 5.4: sr behavior in SLA-VW processor with an L2-L2-L2 configuration

An example of sr usage is in Figure 5.4. The most significant instruction is responsible for the sr bit representation in our example. On the left is the instruction stream, and the right indicates the state of the lane in the corresponding pack. At the start, we assume lanes 0 and 1 are active, and lane 2 is inactive. The most significant bit of lane 1 is set, so the lane is marked as suspended in the next cycle. Similarly, in the second pack, lane 0 encounters a pack with an sr bit, so each suspended lane is marked as active for the next pack. Since L2 is halted, any sr requests are suppressed.

Each lane has an associated width which informs the number of instructions fetched

by the lane and the number of resources available in that lane. Each lane width is defined by the architecture, based on the number of execution units allocated to the lane. Since PC increments are independent, lane widths do not need to be tied together, meaning the same processor can provide a lane with width 1, and another lane of width 3.

Streams are started the same way as SLA processors. Colane (or fork) instructions provide a lane ID and a target address offset. In the next pack, the new lane begins execution at the target PC. Halt can be placed into any nonzero lane and halt the entire lane, including all ways of the lane. Packs with a halt, similar to a traditional SLA, imply the pack's sr bit is set.

Since lanes can be of multiple sizes, we can expand the size of the processor with fewer colane instructions than a traditional SLA processor.

5.2.1 Control Flow and Lane Width Considerations

Control flow is handled the same as a traditional SLA processor. Prior to a branch, each nonzero lane must execute a single prepare-branch (PB) instruction. The instruction can be inserted anywhere in a lane, and when lane 0 encounters a taken branch, each lane jumps to the address of the last executed PB instruction.

Encoding of the PB operation can either be using PC offsets relative to the size of the bundle, or to the size of individual operations. Displacement calculations that use operation granularity allow for lanes of arbitrary size, but limit the distance the lane can branch with a single PB operation. Since powers of two can be quickly represented using bit-shifts, operation granularity and allows for simpler calculations when inserting the bundle into the instruction cache. Conversely, displacement calculations that use bundle granularity can be nontrivial when bundle sizes are not a power of two, since offsets must then be represented with a multiplication operation.

5.3 SLA-VW Architectural Design

The SLA-VW processor is similar to the traditional SLA processor, but has key differences in the front end of the processor. Wider lanes require a VLIW-style fetch unit, meaning multiple operations are fetched and decoded by those lanes each cycle.

Figure 5.5 shows the basic layout of an SLA-VW processor with an L1-L2-L4 configuration. Dotted lines separate lanes, with lane 0 being on top. Depending on lane-width decisions, the overall design may be fundamentally different. While the depth of pipeline stages can be expanded, the basic layout remains the same.

At maximum width, the L1-L2-L4 processor in Figure 5.5 can behave as a 7-wide

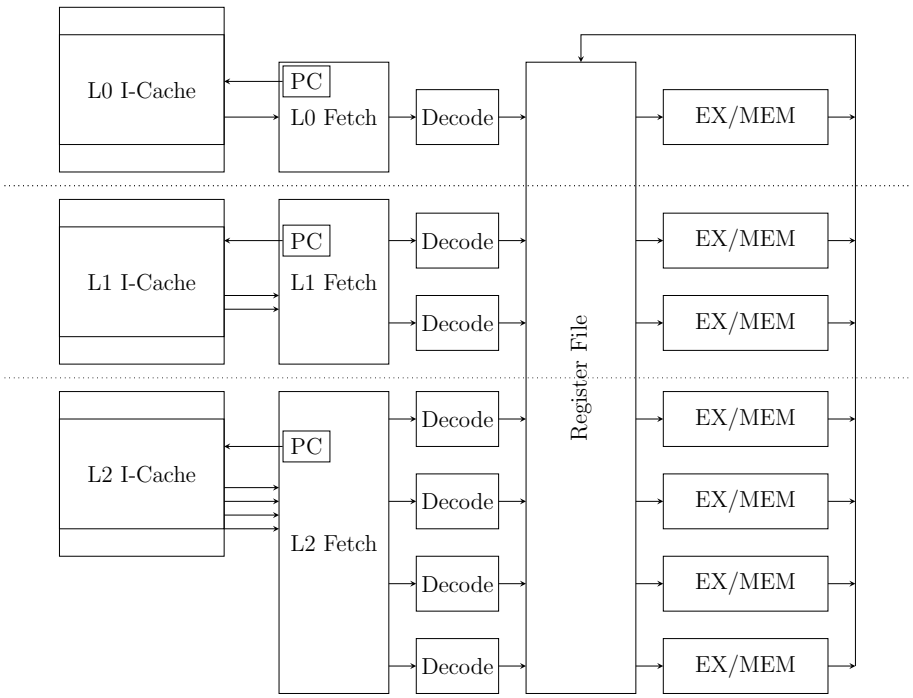


Figure 5.5: Overview of an SLA-VW processor with an L1-L2-L4 configuration

VLIW processor, using only 3 instruction streams. Note that each lane behaves similarly to a small pipelined processor or VLIW processor, but with a shared register file for all lanes.

Each lane is similar, but the number of operations processed in each lane are defined by the width of the lane. In the L1-L2-L4 configuration, lane 0 is given a single operation per pack, meaning it decodes and executes a single operation. Similarly, lane 1 fetches 2 operations per bundle, meaning it decodes and executes two operations. Lane 2 is 4-wide, meaning it fetches 4 operations, decodes them, and executes them per bundle.

The PC increment is determined by the width of the lane, since the bundles in each

lane may be of different sizes, but beyond the fetch unit, the design is similar to the original SLA design.

Since instruction packs only require a single SR bit, a single bit is analyzed for each lane, but lane semantics remain the same as an SLA processor, with lane 0 broadcasting to other lanes when an sr bit is encountered, and other lanes stopping execution when an instruction with an sr bit is set.

5.3.1 Instruction Cache Design

Similar to an SLA processor, each lane has a dedicated instruction cache. However, unlike an SLA processor, each cache may be of a different width depending on the size of the packs being stored. A side effect of bundling operations means that instruction caches only need one tag per bundle, as opposed to one tag per operation. This saves instruction cache space in a similar form to traditional VLIW processors, where only one tag is needed per wide-instruction.

Similar to a normal SLA processor, *prepare branch* (PB) instructions are pre-decoded and inserted into the instruction cache to allow for single-cycle updates to a register in the corresponding lane front end, called the *PB register*. When a pre-decoded PB instruction is fetched, the PB register is set with the value of the PB's target address. When lane 0 encounters a branch or jump, each lane sets their PC to the value in the

PB register for the next cycle. For nonzero lanes, each PB instruction must execute prior to a branch or jump in lane 0. To allow for these instructions to be scheduled the cycle prior to a jump, we pre-decode PB instructions as they are inserted into the instruction cache. When a pre-decoded PB instruction is fetched, the value in the cache, aside from the SR bit, is loaded directly into the PB register.

In a traditional SLA processor instruction cache, there is a single bit reserved to indicate an operation is a PB instruction. In one-wide lanes, the same PB target (PBT) bit that the SLA instruction caches use are sufficient. In lanes wider than one-wide, there are three primary ways to encode PB instructions.

PBT Representation	**Bits Required Per N-wide Bundle**
Assign PBT bits for each operation	N
Assign PBT bits to indicate which way contains a PB instruction	$\lceil log_2(N) \rceil$
Enforce PB instructions are only scheduled in way 0.	1

Table 5.1
Predecoded PBT Options

Table 5.1 discusses three possible PBT configurations. The simplest solution is to have a PBT bit for each operation in a bundle. This is equivalent to an SLA processor's design, since each bundle of size one. However, since there is ever only one PB instruction in a bundle, we can instead use a set of PBT bits to indicate either which way contains the PB instruction.

Alternatively, PB instructions are given a particular way they must be assigned to, reducing the overhead to predecoded PB instructions into a single bit per bundle. However, if ways have specific asymmetric computational capabilities, such as a memory ports or floating point units being available in only specific lane ways, this solution may limit scheduling flexibility.

5.4 Methodology

The objective of this new processor is to lower SLA overhead while minimizing the cost of added nops.

We modified our SLA assembly optimizer to perform transformations for a L2-L2 SLA-VW design. Initial assembly is formed and optimized by *gcc*, before feeding the assembly into our assembly optimizer. Next, we expand the pseudo instructions into individual machine instructions and schedule the machine instructions using the optimizer in conjunction with register information for calls and returns from *gcc* to maintain register liveness information. All optimizations were the same for all processors. We first generate VLIW code, with optimizations using global scheduling, then use mechanical transformations to generate both SLA and SLA-VW L2-L2 code.

We use the Fast/ADL simulator [28] to simulate a 4-wide VLIW processor, a traditional 4-wide SLA processor, and a SLA-VW L2-L2 processor configuration. The processors use variants of the SLA ISA and are implemented as faithful cycle-accurate processors.

We ran the simulator on a variety of SPEC 2006 benchmarks using reference inputs. For power statistics, we use CACTI 7 with 32nm Technology. Each instruction cache is provided 1 read port, with the width being the size of the lane's bundle. So for the SLA processor, the read port is 32 bits, the SLA-VW L2-L2 processor is 64 bits, and the 4-wide VLIW processor is 128 bits. The L2 cache is simulated using a single read port for the instruction caches, meaning that if two or more instruction caches are contending for the L2 cache, the requests must be handled in a pipelined fashion.

Branch Predictor	4096 entry direct-mapped BTB GShare predictor w/ 17-bit branch history.
Page Size	4KB
L1 Partitioned IC	32KiB total, 64B line size, 4-way, VLIW: 11 cycle miss pen. SLA and SLA-VW: 12 cycle miss pen.
L1 DC	32KiB, 64B line size, 4-way
DTLB	32 entries, fully associative
L2 Unified Cache	1MiB, 64B line size, 16-way, 80 cycle miss pen.
5 stage integer pipeline	IF, ID, RF, EX, WB

Table 5.2
Processor Configuration

Table 5.2 shows our processor configurations for our experiments. The VLIW processor uses a single instruction cache configuration of 32KiB, the SLA-VW L2-L2

processor uses a 16KiB cache for each lane, and the SLA processor uses a 4KiB cache for each lane. To ensure fairness, we measure performance using *meaningful instructions*.

For our benchmarks, we allowed each simulator to warm up for 2 billion meaningful instructions, followed by gathering statistics for 10 billion meaningful instructions.

5.5 Results

5.5.1 Performance Analysis

Figure 5.6 shows MIPC for each benchmark. MIPC indicates that the L2-L2 configuration of the SLA processor performs much closer to the VLIW than the SLA processor on it's own. Notably, in benchmarks where VLIW performs significantly better, we find that SLA-VW performs closer to VLIW than SLA. 401.bzip2 performs well for SLA-VW, since when IPC is between 1 and 2 instructions, lanes do not need to be activated, allowing for SLA-VW to take advantage of better cache performance without the need for many additional meta instructions. Benchmarks with regular and highly parallelizable code tend to perform better on the VLIW processor, and what we find is that the SLA-VW processor can achieve similar parallelism faster and with fewer instructions, so in these benchmarks the overhead of meta instructions are

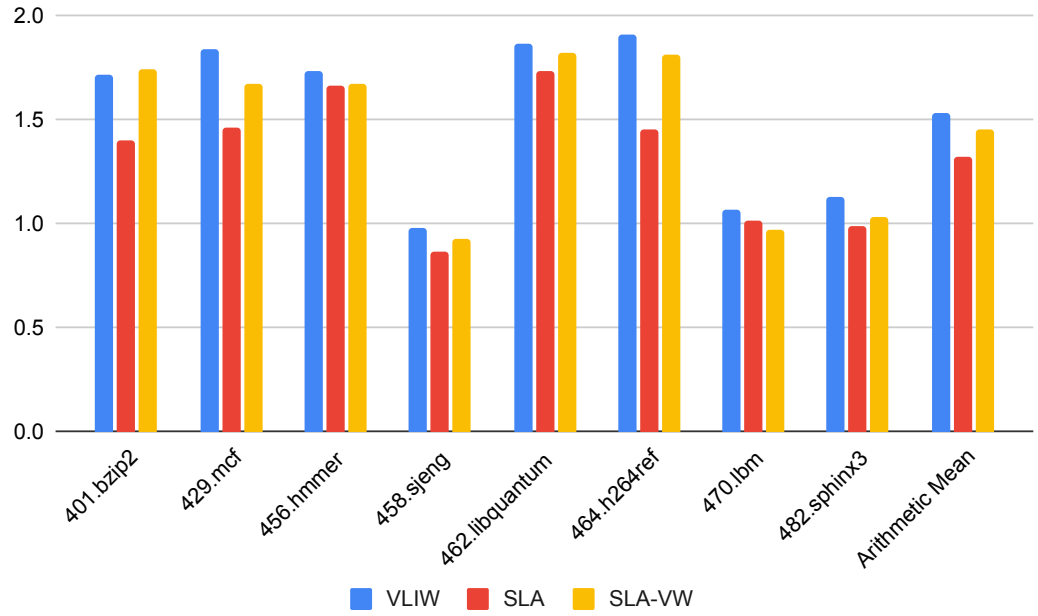


Figure 5.6: Meaningful IPC (Higher is Better)

significantly reduced.

5.5.2 Energy Analysis

Figure 5.7 shows what the state of lane 1 is throughout the lifespan of the benchmarks run. We find that lane 1 only needs to execute around 38.3% of all instruction packs. During other times, it is either suspended or halted, allowing the lane to remain in a low-power state. This is similar to the benefits provided by an SLA processor, and disabling fetching and instruction cache access allows for opportunities for energy savings.

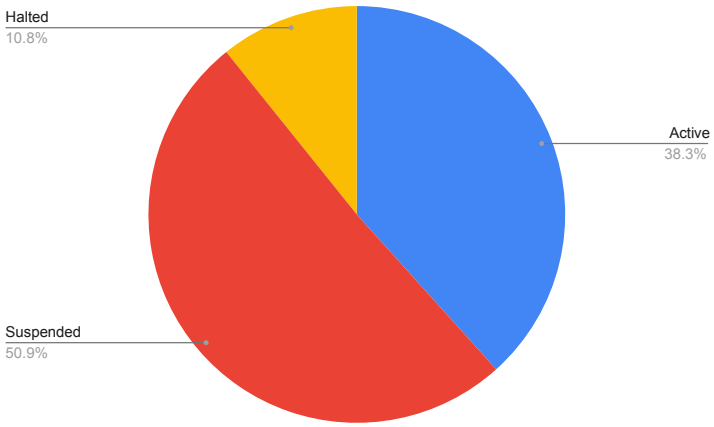


Figure 5.7: SLA-VW L2-L2 Lane Active States for Lane 1

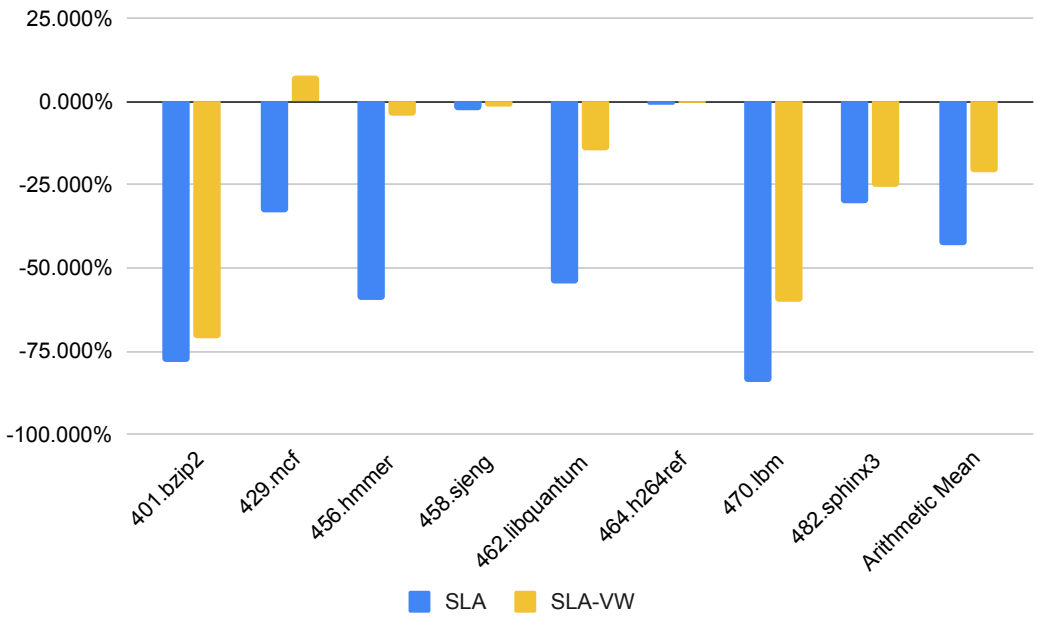


Figure 5.8: Instruction Cache Power Normalized to VLIW (Lower is Better)

As shown by Figure 5.8, SLA-VW loses some power savings in the instruction cache since the lane active granularity is lower than the SLA processor. However, it still outperforms the VLIW processor, with around a 21.3% reduction in instruction cache

power. The most notable exception to power reduction is 429.mcf, which had poor instruction cache performance, meaning more accesses to the unified level 2 cache, driving up power usage. However, in all other cases, the SLA-L2-L2 simulator regularly performed between the instruction cache power of the VLIW and the SLA processors.

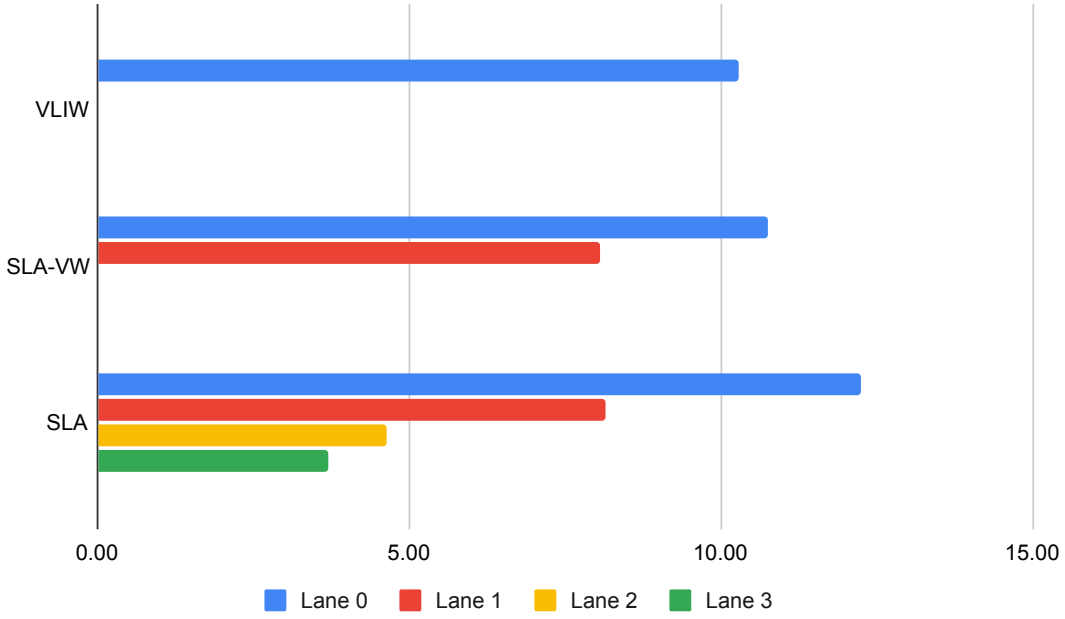


Figure 5.9: Instruction Cache misses per 1k meaningful instructions (Lower is Better)

Figure 5.9 shows that, in our benchmarks, we find instruction cache misses increase for the SLA processor due to a significant number of introduced meta instructions in lane 0. However, we find that the performance of the SLA-VW processor is more resiliant to high numbers of meta instructions, as the potential to move them off of the critical path is increased. Both processors have a marginally better cache hit rate per

access, but due to an increase in meta instructions, perform worse in the meaningful-instructions metric. However, the decrease in performance is mitigated by the the ability to mask instruction cache latency by interleaving cache miss requests when a lane misses, since caches can send requests when suspended, and when multiple caches miss at the same time, requests can be pipelined.

5.5.3 Critical Path Expansion Analysis

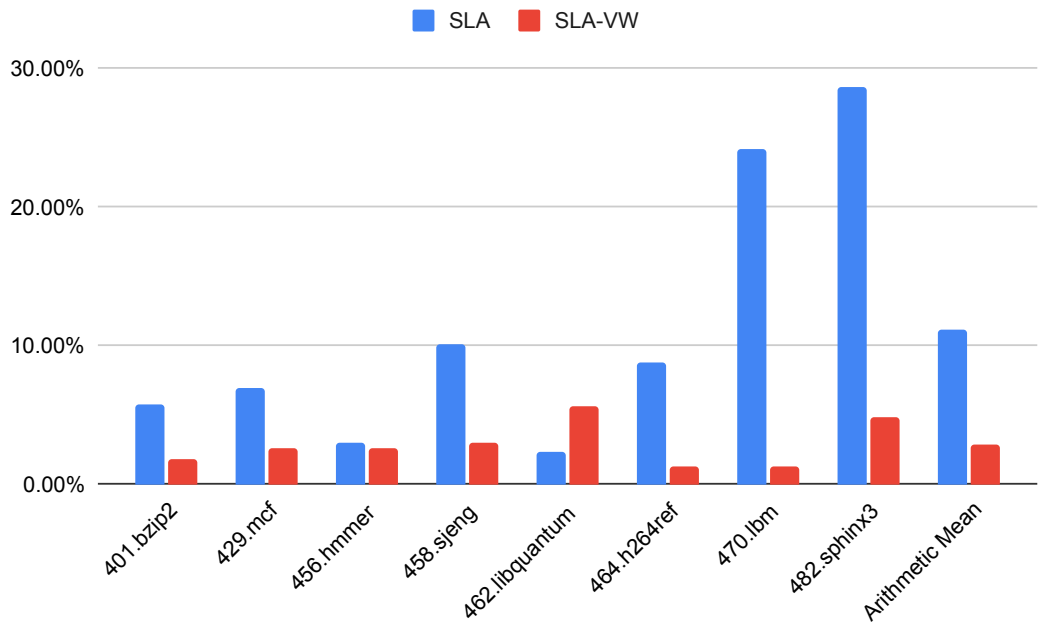


Figure 5.10: Total Instruction Packs Retired Normalized to VLIW (Lower is Better)

We analyze the total number of packs retired during the benchmark execution of 10 billion meaningful instructions to determine the length of the critical path of

the processor. Figure 5.10 shows that SLA-VW expands the length of the critical path significantly less than the SLA processor, since many functions that previously introduced meta instructions only have modest parallelism, and functions that do not expand the width of the processor do not introduce as many meta instructions. The SLA-VW processor only expands the path by 3% throughout spec06. The main reason 470.lbm and 482.sphinx3 significantly expand the packs retired involves tight loops, where a pack must be dedicated for pb instructions to finish executing. Many of the tight loops needing an additional pack for pb instruction are eliminated in the SLA-VW L2-L2 configuration, since a way in lane 1 can be dedicated for a pb operation while a meaningful operation can be scheduled in the other way. Increasing pb flexibility scheduling reduces the chances of the pb operation expanding the critical path, especially in tight loops that contain only a few operations. In 462.libquantum, SLA shrinks the critical path compared to SLA-VW, this is primarily due to functions that do not execute a colane in SLA-VW, where SLA-VW expands to 3 lanes. This is due to the scheduler for the SLA-VW finding that remaining 2-wide was preferential to 4-wide in a large number of functions, and that the meta instruction overhead would be greater than the performance gained by an additional two active lanes.

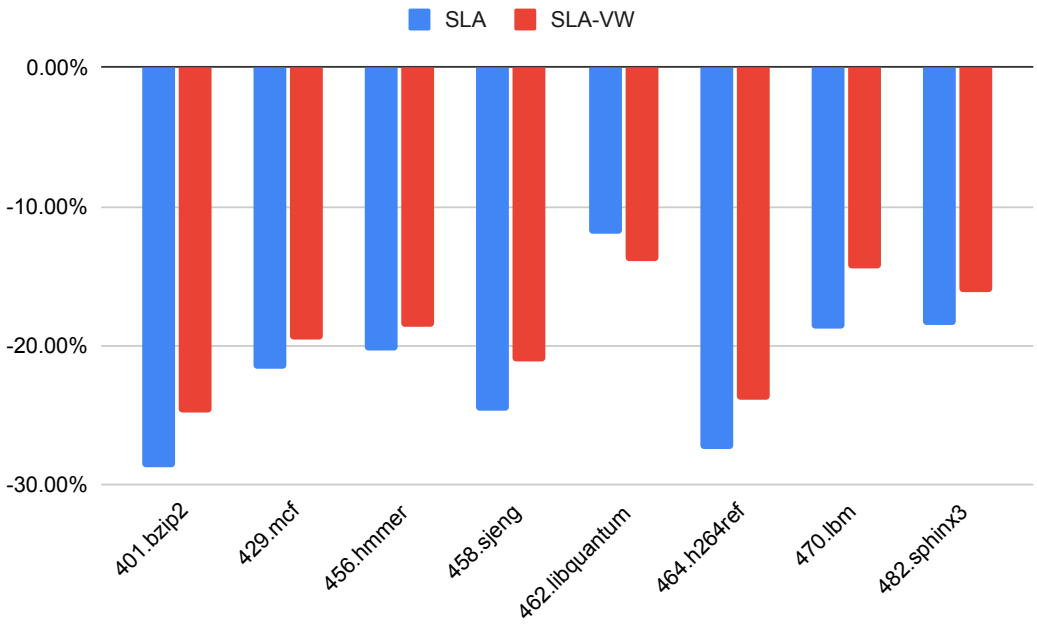


Figure 5.11: Total Operations Encoded (Normalized to VLIW)

5.5.4 Binary Encoding Size

When analyzing the size of the binary encodings, we find that SLA-VW processors tend to perform much closer to SLA processors than VLIW processors, as shown by Figure 5.11. Similar to before, we omit the libraries and count the total number of encoded operations for each benchmark. In most lanes, we found that the operations encoded were close to that of the SLA-VW processor. Notably, we found that 462.libquantum encoded fewer operations than the SLA-VW, primarily due to the number of functions that remained 2-wide instead of 3-wide in the SLA binary, meaning that there were fewer meta instructions introduced.

5.6 Related Work

5.6.1 Multi-PC and Distributed VLIW Designs

The approach of increasing granularity was addressed by research into clustered VLIWs. To mitigate wire delays, from centralized resources, Zhong et al. proposed a variation of the VLIW processor design, called the distributed control path VLIW *(DVLIW)* [42]. The DVLIW processor design similarly distributed fetch, decode, and distribution logic, and each cluster was provided it's own PC. There have been other clustered VLIW approaches as well, which cluster and partition resources. The MultiVLIW processor proposed by Sánchez and González introduced partitioned VLIW architecture design, and designed partitions to maximize locality and minimize wire delays, along with distributing cache memory and instruction fetch. Both of these designs focused on mitigating wire delay and improving VLIW scalability through clustering resources, not the ability to synchronize lanes through suspending and resuming control flow.

5.6.2 Static Code Compression

There have been many techniques to reduce the code size of VLIW architectures by the use of compression. Some architectures compress *nop* operations in the VLIW by setting special bits within the VLIW, such as the Philips Trimedia and TigerSHARC processors [1, 2], which is also referred to as variable length VLIW encoding [36, 40]. Some of these architectures will compress the VLIWs in memory and have the *nop* operations explicitly represented in the instruction cache or have hardware dedicated to realigning packs of instructions that cross cache boundaries and variable PC increments informed by the size of previous packs [2, 5, 19]. A scheme proposed by Li. et al. considers vertical instruction stream compression, but focuses on increasing parallel decoding of Huffman codes [24] instead of entirely decoupling streams. Other proposals to allow for on-the-fly decoding of compressed binaries using dedicated hardware have also been used[23, 34]. The Itanium processor uses template bits to represent which types of operations are placed in a bundle to indicate that the operations are independent and can be executed in parallel [35]. Other solutions include allowing support for compact operations, which take up less space than normal operations, allowing for some packs to have more operations than the normal instruction width would allow [18, 36]. All of these techniques introduce additional complexity to reduce code size. We believe the SLA processor offers many of the same benefits as these compression algorithms without the need for dedicated mechanisms

to maintain alignment that are required in VLIW processors.

5.6.3 Instruction Cache Power

There are many cache improvements that have been targeted for traditional processor designs, and many are not mutually exclusive to the SLA processor. A notable technique is like drowsy instruction caches [22], which reduce power via reducing cache leakage, which can be informed by lane state instead of predictions. Similarly, there are many techniques involving tools to predict either the cache way or subdividing instruction caches to only access parts of the cache that likely contain the data [8, 21, 41]. The instruction cache design in the SLA processor gains efficiency using similar principles by having each lane have it's own instruction cache that is responsible for that specific lane's data.

Chapter 6

Conclusion and Future Work

This dissertation provides a new architectural design paradigm, where single threads can be represented by parallel instruction streams. We explored novel techniques to schedule and execute these streams, and analyzed the consequences of SLA processors versus existing VLIW design through the SCALE ISA.

In this work, our primary contributions were:

1. We formed a novel program representation that decouples independent instructions within a pack into separate instruction streams within a single thread.

2. We designed a new ISA to support this new representation, called the Statically Controlled Synchronous Lane Architecture ISA, which provided tools for

effective lane management.

3. We implemented simulators of a new architecture called the Synchronized Lane Architecture, which encompasses both the hardware and accompanying instruction stream design.

4. We showed an extension of the SLA design, called SLA-VW, that supports a hybrid VLIW-like design for each lane, allowing for wider lanes, and showed how we can minimize SLA overhead while widening the processor.

We believe that the representation of single-threads as a set of instruction streams offers a significant new design space that has previously been unexplored. While this work does not resolve certain issues, such as variable-length instruction latencies, it removes the need for instruction packs to be represented together and lays the groundwork for future solutions to these weaknesses. The SLA processor is a fundamentally new architecture, and with maturity, we believe that this new design can be a powerful, new tool for representation of parallelism in binary code. We demonstrate that the SLA processor's ability to dynamically start and stop streams of instructions without the need for *nops* allows it to remain competitive with current solutions in Chapter 4.

While not as mature as many other processors, the design is orthogonal to many

other approaches, meaning the SLA processor can still benefit from many of the optimizations and opportunities that exist in in-order superscalar and VLIW machines, and can even become a hybrid processor capable of handling combinations of other instruction styles. We also demonstrate an SLA-unique optimization that allows for hybrid alternatives, giving behaviors similar to a combination of the VLIW and SLA processors in Chapter 5.

We aim to continue exploring avenues that this design can offer, including widening the width of the machine to allow for more streams, decoupling the frontend and backend widths, and exploring how fetch windows can be expanded with superscalar machines capable of using multiple streams. We also believe there is promise in looking for ways to remove the need for an explicit synchronization bit, and instead using register dependencies to track lane active requirements to relax the lock-step requirements to which lanes have so far been limited.

In conclusion, we have designed, implemented, and evaluated the new SLA architecture and accompanying ISA, and developed techniques to compare performance with traditional processor design. The approach was comparable to other representations, and shows promise as a new avenue of code representation and execution.

References

[1] "Philips trimedia processor home page," 2002. [Online]. Available: www.semiconductors.philips.com/trimedia

[2] Analog Devices, "Adsp-ts101 tigersharc processor programming reference," 2005. [Online]. Available: https://www.analog.com/media/en/dsp-documentation/processor-manuals/34605284816196ts201_pgr.pdf

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu, "Integrated predicated and speculative execution in the impact epic architecture," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 227–237, 1998.

[4] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[5] A. Brandon, J. Hoozemans, J. van Straten, A. Lorenzon, A. Sartor, A. C. Schneider Beck, and S. Wong, "A sparse vliw instruction encoding scheme compatible with generic binaries," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015, pp. 1–7.

[6] L. N. Chakrapani, J. Gyllenhaal, W.-m. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran: An infrastructure for research in instruction-level parallelism," in *Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers 17*. Springer, 2005, pp. 32–41.

[7] J. Cheng, L. Josipovic, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 288–298.

[8] E.-Y. Chung, C. H. Kim, and S. W. Chung, "An accurate and energy-efficient way determination technique for instruction caches by early tab matching," in *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. IEEE, 2008, pp. 190–195.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of*

the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989, pp. 25–35.

[10] S. Ding, J. Earnest, and S. Önder, "Single assignment compiler, single assignment architecture: Future gated single assignment form*; static single assignment with congruence classes," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 196–207.

[11] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE transactions on computers*, vol. 30, no. 07, pp. 478–490, 1981.

[12] C.-y. Fu and T. M. Conte, "Value speculation mechanisms for epic architectures," *Techni-cal Report, Dept. of Electrical and Computer Engineering, North Carolina State Uni-versity*, 1998.

[13] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value speculation scheduling for high performance processors," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 262–271, 1998.

[14] J. Grossman, "Compiler and architectural techniques for improving the effectiveness of vliw compilation," in *International Conference on Computer Design*, 2000.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2011.

[16] Y.-S. Hwang and J.-J. Li, "On reducing load/store latencies of cache accesses," *Journal of Systems Architecture*, vol. 56, no. 1, pp. 1–15, 2010.

[17] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab *et al.*, "The superblock: An effective technique for vliw and superscalar compilation," in *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*. Springer, 2011, pp. 229–248.

[18] T. Instruments and D. TMS320C66x, "Cpu and instruction set reference guide," *Texas Instruments, Literature Number SPRUGH7*, 2010.

[19] T. Jin, M. Ahn, D. Yoo, D. Suh, Y. Choi, D.-H. Kim, and S. Lee, "Nop compression scheme for high speed dsps based on vliw architecture," in *2014 IEEE International Conference on Consumer Electronics (ICCE)*, 2014, pp. 304–305.

[20] M. Johnson, *Superscalar multiprocessor design*. Prentice-Hall, Inc., 1991.

[21] C. H. Kim, S. W. Chung, and C. S. Jhon, "Pp-cache: A partitioned power-aware instruction cache architecture," *Microprocessors and Microsystems*, vol. 30, no. 5, pp. 268–279, 2006.

[22] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Drowsy instruction caches. leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 219–230.

[23] H. Lekatsas, J. Henkel, and V. Jakkula, "Design of an one-cycle decompression hardware for performance increase in embedded systems," in *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 34–39.

[24] G. Li, Y. Hou, and J. Zhu, "An efficient and fast vliw compression scheme for stream processor," *IEEE Access*, vol. 8, pp. 224 817–224 824, 2020.

[25] A. Mortensen, S. Pomerville, D. Whalley, S. Onder, and G.-R. Uh, "Facilitating the bootstrapping of a new isa," in *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2023, pp. 2–12.

[26] R. Nagarajan, S. Kushwaha, D. Burger, K. McKinley, C. Lin, and S. Keckler, "Static placement, dynamic issue (spdi) scheduling for edge architectures," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004, pp. 74–84.

[27] S. Önder, "Adl++: Object-oriented specification of complicated instruction sets and microarchitectures," in *Processor description languages*. Elsevier, 2008, pp. 247–273.

[28] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *IEEE International Conference on Computer Languages*, Chicago, May 1998, pp. 80–89.

[29] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 257–271.

[30] S. Pomerville and S. Önder, "Putting the cart before the horse: Speculative cache support through delayed store queues," 2022, research Qualifying Examination.

[31] B. R. Rau, "Dynamically scheduled vliw processors," in *Proceedings of the 26th Annual international Symposium on Microarchitecture*. IEEE, 1993, pp. 80–92.

[32] J. Sánchez and A. González, "Modulo scheduling for a fully-distributed clustered vliw architecture," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 124–133.

[33] M. S. Schlansker and B. R. Rau, "Epic: Explicitly parallel instruction computing," *Computer*, vol. 33, no. 2, pp. 37–45, 2000.

[34] S.-W. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 251–254.

[35] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, 2000.

[36] Z. Shen, H. He, X. Yang, D. Jia, and Y. Sun, "Architecture design of a variable length instruction set VLIW DSP," *Tsinghua Science and Technology*, vol. 14, no. 5, pp. 561–569, 2009.

[37] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 344–354.

[38] trimaran, *Trimaran: A Compiler and Simulator for Research on Embedded and EPIC Architectures*, 4th ed., 2007.

[39] A. Unger, T. Ungerer, and E. Zehendner, "Static speculation, dynamic resolution," in *Proc. 7th Workshop Compilers for Parallel Computers*. Citeseer, 1998, pp. 243–253.

[40] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for vliw processors using variable-to-fixed coding," in *International Symposium on System Synthesis*, Oct. 2002, pp. 138–143.

[41] C.-C. Yu, Y. H. Hu, Y.-C. Lu, and C. C.-P. Chen, "Power reduction of a set-associative instruction cache using a dynamic early tag lookup," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1799–1802.

[42] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker, "A distributed control path

architecture for vliw processors," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, 2005, pp. 197–206.

Appendix A

Scale ISA Instructions

The SCALE ISA is a MIPS-like ISA that uses the same major instruction categories of R-type, J-type, and I-type operations, though some instructions concatonate fields. Furthermore, since the opcode fields are more contracted, we add an additional type to handle many floating point operations, called FP-type. Below is the ISA and bit layouts for the SCALE ISA, followed by tables of introduced dedicated synchronization instructions and their definitions and then tables of all other SCALE instructions.

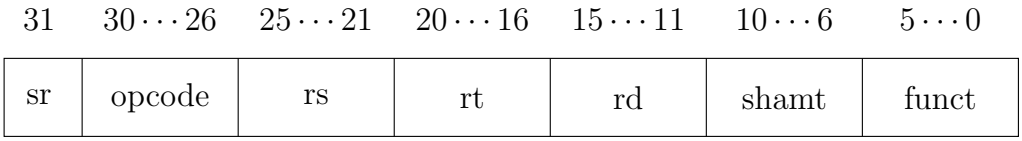


Figure A.1: R-type instruction layout

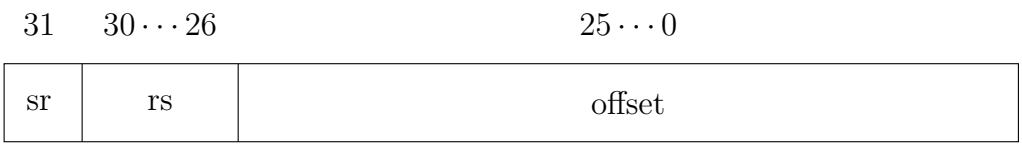


Figure A.2: J-type instruction layout

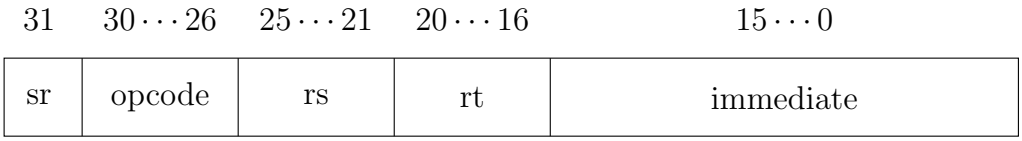


Figure A.3: I-type instruction layout

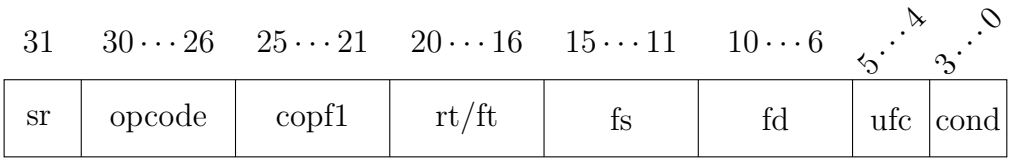


Figure A.4: FP-type instruction layout

Instruction	**opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Definition**
Fork/Colane	00000	rs	rt	rd	targetlane	110000	$PC[targetlane] \leftarrow concat(rs, rt, rd)$
Halt	00000	0	0	0	0	110111	$LAS[current\_lane] \leftarrow inactive$
Return	00000	0	0	0	0	010101	for each lane i=0..n-1: $LAS \leftarrow PLS$ $PC[i] \leftarrow RT[i]$
LAS	00000	rs	0	0	1	010011	$PLS \leftarrow mem[rs]$
SAS	00000	rs	0	0	1	010011	$mem[rs] \leftarrow PLS$
LWRT	00000	rs	0	0	0	010011	$RT[rt] \leftarrow mem[rs]$
SWRT	00000	rs	rt	0	1	010011	$mem[rs] \leftarrow RT[rt]$

Table A.1
SCALE R-type instructions

Instruction	**opcode**	**Offset**	**Definition**
Call	00111	offset	for each lane i=0..n-1: if $LAS[i] = active$: $RT[i] \leftarrow PC$[i] + 4, else $RT[i] \leftarrow PC[i]$ $PLS[i] = LAS[i]$ if($i > 0$): $LAS[i] \leftarrow inactive$ $PC[0] \leftarrow MostSignificantBits_4(PC[0])|(offset << 2)$
Callm	10010	offset	for each lane i=0..n-1: if $(i > 0)$: $LAS[i] \leftarrow inactive$ $PC[0] \leftarrow MostSignificantBits_4(PC[0])|(offset << 2)$
PB	01101	offset	PB[current_lane]=PC[current_lane]+offset

Table A.2
SCALE J-type instructions

Name	**opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Definition**
jr	00000	rs	0	0	0	001000	$PC[0] \leftarrow gpr[rs]$ *only used in baseline
jalr	00000	rs	0	rd	0	001001	$PC[0] \leftarrow gpr[rs]$ $gpr[rd] \leftarrow PC[0] + 4$ *only used in baseline
sll	00000	0	rt	rd	shamt	000000	$gpr[rd] \leftarrow gpr[rt] << shamt$
srl	00000	0	rt	rd	shamt	000010	$gpr[rd] \leftarrow gpr[rt] >>_{logical} shamt$
sra	00000	0	rt	rd	shamt	000011	$gpr[rd] \leftarrow gpr[rt] >>_{arithmetic} shamt$
sllv	00000	rs	rt	rd	0	000100	$gpr[rd] \leftarrow gpr[rt] <<$ $LeastSignificantBits_5[gpr[rs]]$
srlv	00000	rs	rt	rd	0	000110	$gpr[rd] \leftarrow gpr[rt] >>_{logical}$ $LeastSignificantBits_5[gpr[rs]]$

Name	**opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Definition**
srav	00000	rs	rt	rd	0	000111	$gpr[rd] \leftarrow gpr[rt] >>_{arithmetic} LeastSignificantBits_5[gpr[rs]]$
add	00000	rs	rt	rd	0	111000	$gpr[rd] \leftarrow gpr[rt] + gpr[rs]$
sub	00000	rs	rt	rd	0	100010	$gpr[rd] \leftarrow gpr[rt] - gpr[rs]$
addu	00000	rs	rt	rd	0	100001	$gpr[rd] \leftarrow gpr[rt] +_{unsigned} gpr[rs]$
subu	00000	rs	rt	rd	0	100011	$gpr[rd] \leftarrow gpr[rt] -_{unsigned} gpr[rs]$
rdhwr	00000	rs	0	rd	0	000001	$gpr[rd] \leftarrow kernel\_register[rs]$
slt	00000	rs	rt	rd	0	101010	$gpr[rd] \leftarrow gpr[rs] < gpr[rt]$
sltu	00000	rs	rt	rd	0	101011	$gpr[rd] \leftarrow gpr[rs] <_{unsigned} gpr[rt]$
syscall	00000	0	0	0	0	001100	perform syscall
and	00000	rs	rt	rd	0	100100	$gpr[rd] \leftarrow gpr[rs]\&gpr[rt]$
or	00000	rs	rt	rd	0	100101	$gpr[rd] \leftarrow gpr[rs]\|gpr[rt]$
xor	00000	rs	rt	rd	0	100110	$gpr[rd] \leftarrow gpr[rs] \oplus gpr[rt]$

Name	**opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Definition**
nor	00000	rs	rt	rd	0	100111	$gpr[rd] \leftarrow \neg(gpr[rs] \vert gpr[rt])$
mult	00000	rs	rt	0	0	011000	$hi\_register \leftarrow (gpr[rs] * gpr[rt])_{hi}$ $lo\_register \leftarrow (gpr[rs] * gpr[rt])_{lo}$
multu	00000	rs	rt	0	0	011001	$hi\_register \leftarrow (gpr[rs]$ $*_{unsigned} gpr[rt])_{hi}$ $lo\_register \leftarrow (gpr[rs] *$ $_{unsigned} gpr[rt])_{lo}$
div	00000	rs	rt	0	0	011010	$hi\_register \leftarrow (gpr[rs]/gpr[rt])_{hi}$ $lo\_register \leftarrow (gpr[rs]/gpr[rt])_{lo}$
divu	00000	rs	rt	0	0	011011	$hi\_register \leftarrow (gpr[rs]/$ $_{unsigned} gpr[rt])_{hi}$ $lo\_register \leftarrow (gpr[rs]/$ $_{unsigned} gpr[rt])_{lo}$

Name	**opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Definition**
mfhi	00000	0	0	rd	0	010000	$gpr[rd] = hi\_register$
mflo	00000	0	0	rd	0	010010	$gpr[rd] = lo\_register$
mthi	00000	rs	0	0	0	010001	$hi\_register = gpr[rs]$
mtlo	00000	rs	0	0	0	010011	$lo\_register = gpr[rs]$
ctc1	01011	rs	fs	00110	0	0	$fp\_control[fs] \leftarrow gpr[rs]$
cfc1	01011	rs	fs	00010	0	0	$gpr[rs] \leftarrow fp\_control[fs]$
mtc1	01011	rs	fs	00100	0	0	$fpr[fs] \leftarrow gpr[rs]$
mfc1	01011	rs	fs	00000	0	0	$gpr[rs] \leftarrow fpr[fs]$

Table A.3
SCALE R-type instructions

Name	**opcode**	**Offset**	**Definition**
j	01100	j_offset	$PC[0] \leftarrow MostSignificantBits_4(PC[0])\|(offset << 2)$
jal	01101	j_offset	$gpr[31] = PC[0] + 4$ $PC[0] \leftarrow MostSignificantBits_4(PC[0])\|(offset << 2)$ *only used in baseline

Table A.4
SCALE J-type instructions

Instruction	**opcode**	**rs**	**rt**	**immediate**	**Definition**
addi	00001	rs	rt	immediate	$gpr[rt] \leftarrow gpr[rs] + sign\_extend(immediate)$
addiu	00010	rs	rt	immediate	$gpr[rt] \leftarrow gpr[rs] +_{unsigned}$ $sign\_extend(immediate)$
andi	00011	rs	rt	immediate	$gpr[rt] \leftarrow gpr[rs] \& zero\_extend(immediate)$
xori	00100	rs	rt	immediate	$gpr[rt] \leftarrow gpr[rs] \oplus zero\_extend(immediate)$
ori	00101	rs	rt	immediate	$gpr[rt] \leftarrow gpr[rs] \mid zero\_extend(immediate)$
beqz	00110	rs	0	immediate	if $rs = 0$: $PC[0] \leftarrow PC[0] + immediate$ for each lane i=1..n-1: $PC[i] \leftarrow PB[i]$
bnez	00110	rs	1	immediate	if $rs \neq 0$: $PC[0] \leftarrow PC[0] + immediate$ for each lane i=1..n-1: $PC[i] \leftarrow PB[i]$
lb	01110	rs	rt	0	$gpr[rt] \leftarrow sign\_extend(mem_{byte}[gpr[rs]])$

Instruction	**opcode**	**rs**	**rt**	**immediate**	**Definition**
lbu	01111	rs	rt	0	$gpr[rt] \leftarrow zero\_extend(mem_{byte}[gpr[rs]])$
lh	10000	rs	rt	0	$gpr[rt] \leftarrow sign\_extend(mem_{halfword}[gpr[rs]])$
lhu	10001	rs	rt	0	$gpr[rt] \leftarrow zero\_extend(mem_{halfword}[gpr[rs]])$
lw	10011	rs	rt	0	$gpr[rt] \leftarrow mem[gpr[rs]]$
lwc1	10100	rs	ft	0	$fpr[ft] \leftarrow mem[gpr[rs]]$
lwl	10101	rs	rt	0	$gpr[rt] \leftarrow LeastSignificantBits_{16}(gpr[rt])$ $\|(mem_{halfword}[gpr[rs]] << 16)$
lwr	10110	rs	rt	0	$gpr[rt] \leftarrow MostSignificantBits_{16}(gpr[rt])\|$ $mem_{halfword}[gpr[rs]]$

Instruction	**opcode**	**rs**	**rt**	**immediate**	**Definition**
sb	10111	rs	rt	0	$mem_{byte}[gpr[rs]] \leftarrow (gpr[rt]\&0xff)$
sh	11000	rs	rt	0	$mem_{halfword}[gpr[rs]] \leftarrow (gpr[rt]\&0xffff)$
sw	11011	rs	rt	0	$mem[gpr[rs]] \leftarrow gpr[rt]$
swl	11100	rs	rt	0	$mem_{halfword}[gpr[rs]] \leftarrow MostSignificantBits_{16}(gpr[rt])$
swr	11101	rs	rt	0	$mem_{halfword}[gpr[rs]] \leftarrow LeastSignificantBits_{16}(gpr[rt])$

Table A.5
SCALE I-type instructions

Instruction	**opcode**	**copf1**	**rt/ft**	**fs**	**fd**	**ufc**	**cond**	**Definition**
bc1f	01011	01000	00000	off_0	off_1	off_2	off_3	if $cpc\_r = 0$: $PC[0] \leftarrow PC[0]+$ $concat(off_1, off_2, off_3)$ for each lane i=1..n-1: $PC[i] \leftarrow PB[i]$
bc1t	01011	01000	00001	off_0	off_1	off_2	off_3	if $cpc\_r \neq 0$: $PC[0] \leftarrow PC[0]+$ $concat(off_1, off_2, off_3)$ for each lane i=1..n-1: $PC[i] \leftarrow PB[i]$
c.cond.s	01011	10000	ft	fs	0	11	cond	cpc_r = $compare_{cond}(fpr[fs], fpr[ft])$

Instruction	**opcode**	**copf1**	**rt/ft**	**fs**	**fd**	**ufc**	**cond**	**Definition**
c.cond.d	01011	10001	ft	fs	0	11	cond	$cpc\_r = compare\_double_{cond}(fpr[fs], fpr[fs+1], fpr[ft], fpr[ft+1])$
cvt.d.w	01011	10010	0	fs	fd	10	0001	$fpr[fd], fpr[fd+1] \leftarrow convert_{fixed\_to\_double}(fpr[fs])$
cvt.d.s	01011	10000	0	fs	fs	10	0001	$fpr[fd], fpr[fd+1] \leftarrow convert_{float\_to\_double}(fpr[fs])$
cvt.s.w	01011	10000	0	fs	fd	10	0000	$fpr[fd] \leftarrow convert_{single\_to\_fixed}(fpr[fs])$
cvt.s.d	01011	10001	0	fs	fd	10	0000	$fpr[fd] \leftarrow convert_{double\_to\_float}(fpr[fs], fpr[fs+1])$

Instruction	**opcode**	**copf1**	**rt/ft**	**fs**	**fd**	**ufc**	**cond**	**Definition**
add.s	01011	10000	ft	fs	fd	00	1000	$fpr[fd] \leftarrow fpr[fs] +_{single} fpr[ft]$
add.d	01011	10001	ft	fs	fd	00	1000	$fpr[fd], fpr[fd + 1] \leftarrow (fpr[fs], fpr[fs + 1]) +_{double} (fpr[ft], fpr[ft + 1])$
sub.s	01011	10000	ft	fs	fd	00	0001	$fpr[fd] \leftarrow fpr[fs] -_{single} fpr[ft]$
sub.d	01011	10001	ft	fs	fd	00	0001	$fpr[fd], fpr[fd + 1] \leftarrow (fpr[fs], fpr[fs + 1]) -_{double} (fpr[ft], fpr[ft + 1])$
div.s	01011	10000	ft	fs	fd	00	1010	$fpr[fd] \leftarrow fpr[fs]/_{single} fpr[ft]$

Instruction	**opcode**	**copf1**	**rt/ft**	**fs**	**fd**	**ufc**	**cond**	**Definition**
div.d	01011	10001	ft	fs	fd	00	1010	$fpr[fd], fpr[fd + 1] \leftarrow (fpr[fs], fpr[fs + 1])/_{double}(fpr[ft], fpr[ft + 1])$
mul.s	01011	10000	ft	fs	fd	00	0010	$fpr[fd] \leftarrow fpr[fs] *_{single} fpr[ft]$
mul.d	01011	10001	ft	fs	fd	00	0010	$fpr[fd], fpr[fd + 1] \leftarrow (fpr[fs], fpr[fs + 1]) *_{double} (fpr[ft], fpr[ft + 1])$
neg.s	01011	10000	0	fs	fd	00	0111	$fpr[fd] \leftarrow negate(fpr[fs])$
neg.d	01011	10001	0	fs	fd	00	0111	$fpr[fd], fpr[fd + 1] \leftarrow negate(fpr[fs], fpr[fs + 1])$
abs.s	01011	10000	0	fs	fd	00	0101	$fpr[fd] \leftarrow abs(fpr[fs])$

Instruction	**opcode**	**copf1**	**rt/ft**	**fs**	**fd**	**ufc**	**cond**	**Definition**
abs.d	01011	10001	0	fs	fd	00	0101	$fpr[fd], fpr[fd + 1] \leftarrow abs(fpr[fs], fpr[fs + 1])$
mov.s	01011	10000	0	fs	fd	00	0110	$fpr[fd] \leftarrow fpr[fs]$
mov.d	01011	10001	0	fs	fd	00	0110	$fpr[fd], fpr[fd + 1] \leftarrow fpr[fs], fpr[fs + 1]$
trunc.w.s	01011	10000	0	fs	fd	00	1101	$fpr[fd] \leftarrow truncto_{fixed}(fpr[fs])$
trunc.w.d	01011	10000	0	fs	fd	00	1101	$fpr[fd] \leftarrow truncto_{fixed}(fpr[fs], fpr[fs+1])$

Table A.6
SCALE FP-type instructions