DSP Extensions to the TRIPS ISA

Kevin Beckwith Bush Doug Burger

Abstract

In this paper, we propose a set of DSP extensions to the TRIPS ISA and evaluate their performance. By extending the TRIPS ISA with specialized DSP instructions, we offer an explorative look at the interaction conventional specialization techniques (such as SIMD instructions) have with EDGE ISAs. We discuss the implementation and its feasibility and provide non-intrusive compiler support through hand-written library functions. Finally, we evaluate the performance benefits of our extensions with custom library-emphasizing benchmarks and compare our results with those of the industry standard TI c6416 digital signal processor.

1 Introduction

While diminishing performance gains in conventional architectures are fueling novel designs which more effectively extract parallelism, the demand for high performance real-time applications continues to grow. In conjunction, these two pressures are driving the need for next-generation general-purpose architectures to perform well in the DSP domain. TRIPS, the first implementation of an EDGE ISA, is an example of such next-generation architectures. With up to 1024 instructions in flight and 16 redundant execution units, one expects good performance potential in the DSP domain where regularity and concurrency are common [12, 5]. We take an explorative look at the impact conventional DSP specialization techniques - such as SIMD instructions - have on EDGE ISAs by proposing and evaluating a set of DSP extensions to the TRIPS ISA.

To facilitate reliable performance testing we augmented an already developed TRIPS toolchain to support the ISA extensions [14]. A high-performance library of common DSP functions was hand-coded to leverage the new instructions while a scheduler, linker, and assembler were extended to provide seamless vertical support of the extensions. Both an architectural simulator (functional) and a processor simulator (cycle-accurate) were also enhanced to provide correctness verification and performance modeling. Additionally, to remove hand-optimization as a performance variable, we provide a duplicate library with similar control-flow for each function without leveraging the new instructions.

By writing simple wrapper benchmarks, we are able to isolate the library functions and measure their performance directly. In this way, we can directly compare the performance advantage our DSP extensions provide in all the functions our library supports. We then make a comparison of our results with those of industry standard DSP processors.

The rest of this paper is organized as follows. Section 2 provides a brief background summary. Section 3 provides an architectural summary of the TRIPS and TIc64x architectures. Section 4 describes the extensions proposed. Sections 5 and 6 discuss our implementation methodology and benchmark suite. Section 7 provides an analysis of our performance results. Section 8 has some concluding remarks and a description of future work can be found in Section 9.

2 Background

To facilitate discussion, below is a brief introduction to the TRIPS architecture, Digital Signal Processing, and the nature of ISA extensions.

2.1 TRIPS

The TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System) project is a multidisciplinary effort to develop a high performance next generation architecture at the University of Texas at Austin. The principle investigators of the TRIPS project are Dr. Doug Burger, Dr. Steve Keckler, and Dr. Kathryn McKinley.

The TRIPS architecture is the first of a new class of architectures called EDGE (Explicit Data Graph Execution) architectures. EDGE architectures vary from conventional modern microprocessors on multiple accounts, most notably for their block atomic execution model that exposes parallelism by aggregating groups of instructions into atomic units [13]. the TRIPS architecture is a general-purpose architecture which consists mostly simple RISC-like (Reduced Instruction Set Computation) instructions. While these instructions provide the functionality necessary for any application, they do not provide the same performance potential as specialized instructions in the DSP domain.

2.2 Digital Signal Processing

Digital signal processors are specialized microprocessors designed specifically for DSP (Digital Signal Processing) applications. Digital signal processors characteristically have separate program and data memories and the ability to act as direct memory access devices in host environments.

DSP applications usually measure, filter, or emulate real-world analog signals. The DSP domain can be best characterized as possessing regular data streams or values and often is under real-time constraints [4]. The core operations in these applications commonly have high degrees of inherent data-level parallelism.

2.3 ISA Extensions

We can expect to improve the performance of the TRIPS architecture in the DSP domain with specialization. By augmenting the TRIPS ISA to support high-performance DSP instructions we effectively tune the architecture for workloads characteristic of the DSP domain.

This sort of specialization of general purpose architectures has seen significant success in the past. High-profile examples include the *3DNow!* and *MMX* extension for the x86 ISA, and the *AltiVec* extension to the Power ISA [2, 3, 9].

3 Architectural Summary

3.1 TIc6416

The the TMS320C64x series processor is part of Texas Instruments' TMS320C6000 DSP platform. It is a 8-wide VLIW (Very Long Instruction Word) architecture designed for real-time DSP applications[1]. The VLIW execution model is often chosen in DSP architectures for its ability to achieve relatively high degrees of ILP while maintaining power efficiency. This power efficiency is achieved by exposing complexity to the compiler allowing for the removal of power hungry features such as a re-order

Feature	TI64x	TI67x	TRIPS
Execution Model	8-wide VLIW		Explicit Data-Graph Execution
Target Application	Real-time DSP		General Purpose
Data dependence	Conditional instructions		Compound predication support
Max theoretical throughput	8 IPC		16 IPC
Execution resources	2 Multipliers & 6 ALUs		16 ALUs
Registers	64 32-bits	32 32-bits	128 64-bits
Integer data support	8/16/32/40/64-bit	16/32/40-bit	64-bit only
FP data support	None	32-bit & 64-bit	64-bit only
Data bandwidth	Four 32-bit ports	Two 32-bit ports	Four 64-bit banks
Instruction bandwidth	One 256-bit port		512-bits per cycle

Table 1: Summary of ISAs

buffer [4]. Even though the TIc67x architecture has floating point support analogous to the TRIPS architecture, We chose the TIc64x architecture because of it's comparable data bandwidth and more complex suite of DSP-tuned instructions.

Table 1 shows an overview of the TIc6416's architecture as compared to TRIPS. The TIc64x series ISA features many specialized DSP instructions despite an absence for hardware floating-point support, making it an ideal candidate for comparison of for our extensions. Application-specific domains include cryptography, error correction, and advanced mathematics. Additionally, SIMD support is available in many instructions, both application-specific and general purpose. A list description of many of the supported DSP instructions can be found in Table 2.

Instruction	Description	SIMD	Application
ADD	Compute sum	2x16/4x8-bit	General Purpose
MUL	Compute product	2x16/4x8-bit	General Purpose
ANDN	Bitwise logical-and, and invert	-	Cryptography
BITR	Bitwise reverse	-	Cryptography
DEAL	De-interleave and pack	-	Cryptography
SHFL	Logical shuffle bits (reverse DEAL)	-	Cryptography
ROTL	Logical bit rotation left	-	Cryptography
BITC4	Packed Bit count	-	Error Correction
GMPY4	Packed Galios-field multiply	-	Error Correction
NORM	Count repeating bits	-	Error Correction
AVG	Arithmetic mean	2x16/4x8-bit	Advanced Math
DOTP	Dot product	2x16/4x8-bit	Advanced Math
MAX/MIN	Maximum/Minimum value retrieval	2x16/4x8-bit	Advanced Math

Table 2: TIc64x DSP instructions

3.2 TRIPS

The TRIPS architecture is the first implementation of an EDGE (Explicit Data Graph Execution) ISA [**?**]. EDGE ISAs leverage a limited data-flow execution model to expose high degrees of ILP without

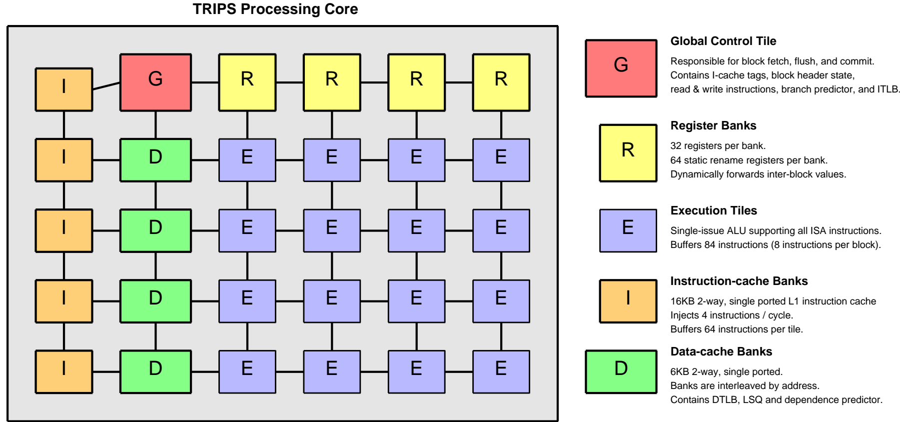

Figure 1: Block Diagram of TRIPS Processing Core

requiring fundamental changes to the programming paradigm. In this model, instructions are aggregated into instruction blocks which are in turn fetched, executed, and committed atomically. Within a block, instruction communication occurs directly, thus bypassing the global register file and supporting distributed out-of-order execution [12].

The TRIPS architecture maintains power efficiency by distributing general computing resources and removing power hungry structures such as centralized register files. Figure 1 shows a block diagram of a TRIPS processing core. It is a distributed tile architecture arranged in a 2-D mesh topology. Operands are passed between tiles via an Operand Network (OPN) allowing for direct communication within a block. Communication across blocks are carried out through the banked data-cache and register file. The architecture maintains up to 1024 instructions in flight with support for concurrent execution of 8 blocks (7 speculative) of at most 128 instructions each. The execution grid consists primarily of 16 execution tiles, each contain 1 ALU and the necessary instruction buffers, input ports, and operand buffers.

Blocks are formed and scheduled by the compiler onto the microarchitecture. These statically scheduled blocks resemble the basic execution unit of VLIW architectures, however, the key difference is that instructions within a block are not required to be independent and are dynamically issued. Scheduling in this context, refers to spatial location and not execution order. Additional block constraints are required by the architecture, most significantly a maximum of 32 register reads and writes and at most 32 memory accesses are allowed per block.

Several simplifying assumptions were made in order to produce the TRIPS hardware prototype. These include an absence of hardware support for floating point division, square root, and 32-bit floating point operations. The solution used by the prototype is to emulate these operations in software and convert all 32-bit floating point variables to 64-bit. The cost of these emulations and conversions can have a substantial impact on overall performance. Since these are prototyping artifacts and would not be present in a full production model, our extensions will focus on other aspects of the architecture and the benchmarks will deliberately not expose them.

4 Extensions Summary

In total, we selected 5 instructions common in specialized digital signal processors for our evaluation. A summary of each extension can be found below. Included is a motivational discussion

outlining our selection of instructions. We have organized by category: SIMD instructions in Subsection 4.1, and Bitwise operators in Subsection 4.2. A complete definition, format description, and placement of each extension in the TRIPS instruction space can be found in the appendix. We will employ the same symbolic notation and formatting conventions as the TRIPS architectural reference manual [10]. Of particular relevance, Chapter 8 of the reference manual includes an instruction set summary and a thorough description of notation.

4.1 SIMD

The use of SIMD (Single Instruction, Multiple Data) instructions is a well established technique to achieve high degrees of data-level parallelism. It has seen widespread use in conventional general-purpose architectures through ISA extensions such as *MMX* and *AltiVec* for the x86 and Power architectures [3, 9]. We observe that even specialized DSP architectures, such as the TIc6416, exploit data-level parallelism with this technique. A list of all the SIMD instructions available in the TIc6416 is available in Section 3.

SIMD instructions are particularly attractive on the TRIPS architecture because of the 64-bit data-path. TRIPS supports 64-bit payloads for all operand traffic, has 64-bit operand reservation stations, and all of the 128 registers are 64 bits wide. Additionally, TRIPS supports accesses to memory in double-word blocks (64 bits). In this way, TRIPS does not require data-path augmentations or specialized register files to support SIMD instructions with data-types of 32-bits or fewer. All of the SIMD instructions proposed below require no more than full 64-bit width of the TRIPS data-path.

By collapsing instructions, 64-bit SIMD instructions effectively more densely pack operands improving overall OPN bandwidth utilization. Previous studies have demonstrated that contention along the OPN can have a significant impact on overall processor performance [7, 6].

4.1.1 32-bit Data

We chose to augment the TRIPS ISA with packed 32-bit data, SIMD instructions - `ADD2` and `MUL2`. 32-bit data operators are natural candidates for SIMD extension because of their frequent use. Single-precision integers are common primitive data types employed in modern programming [4].

`ADD2` (appendix A) performs two packed 32-bit additions in a 64-bit space. Independent single-precision addition is commonplace in vector operations, graphics processing, and matrix calculations. By combining two operations into one, this packed addition instruction can reduce the dynamic instruction count by up to a factor of two.

`MUL2` (appendix C) performs similarly to the `ADD2` instruction by computing two packed 32-bit multiplications in a 64-bit space. The result is two 32-bit products corresponding to the upper 32-bits, and lower 32-bits of OP0 and OP1 respectively. `MUL2` complements `ADD2` in our SIMD study because it allows us to precisely analyses the performance impact of instruction latency in SIMD instructions on the TRIPS processor.

4.1.2 8-bit Data

Extending on our observation that TRIPS maintains a 64-bit data-path, we also chose to augment the TRIPS ISA with packed 8-bit data, SIMD instructions - `ADD8` and `MUL8`. 8-bit data, employed mostly in digital audio stream processing, is commonly selected for use in SIMD instruction sets because of the vast data-level independence expressed in most audio processing applications. Digital signal processors, such as the TIc6416, commonly support SIMD operation on this data size.

ADD8 (appendix B) performs eight packed 8-bit additions in a 64-bit space. We chose against an instruction-encoded immediate for functional generality (such as in the TIc64x), however, such an instruction could easily be included in a production-level extension suite.

MUL8 (appendix D) performs eight packed 8-bit multiplications in a 64-bit space. MUL8 complements ADD8 in our SIMD study in an analogous fashion to our 32-bit SIMD extensions.

4.2 Bitwise Rotate

Bitwise rotation (appendix E) is a common operation supported by specialized ISAs to improve performance in cryptographic applications. Cryptographic algorithms such as AES encryption, require rotation of blocks of bits at their core, and as such are very performance sensitive to the latency of this operation. We observe that the TIc64x architecture has support to reduce the number of instructions required to perform this operation by half via a rotate-left instruction [1]. We chose to include an analogous 64-bit version in our ISA extension, ROTL (appendix E).

Functionally equivalent to two complementary bit-wise shift operations, ROTL will provide us with a metric by which to evaluation the effect instruction reduction has on the TRIPS architecture in the absence of data-level independence.

5 Implementation

5.1 Library

To provide an accurate performance model of the extensions without forcing undo complexity onto the compiler, we hand-coded a suite of DSP functions which benefit from the ISA extensions and bundled them into a library which can be linked by the compiler. We employed the Scale compiler to provide the code foundation for our hand optimizations [11]. To provide a fair baseline for comparison, we also hand-coded a second library with the same functional support which does not exercise the ISA extensions but maintains the same control flow. Additionally, we augmented the source code with directives unique to the TIc64x compiler to enable the most aggressive optimization on that machine.

5.2 Compiler and Simulation

By leveraging a library infrastructure, we minimized the impact our extensions had on the compiler. Still, simple modifications were necessary to handle the new instructions in the spatial scheduler, assembler, and linker.

We employed an already developed TRIPS scheduler which statically assigns instructions to execution units [8]. The instructions wait at these execution units for their operands to arrive and execute as soon as possible. Scheduling up to 128 instructions is a sophisticated problem and carries a significant portion of the responsibility for the overall performance of execution. We observe that the latency and pipeline-ability of packed SIMD instructions is the same as their non-SIMD counterparts and were able to duplicate the current scheduling logic already in place, resulting in good-quality, easy-to-implement schedules of our instruction extensions.

The TRIPS assembler and linker both required only trivial enhancements to support the new instructions such as instruction definition and opcode space assignment.

A previously developed cycle-accurate simulator, *tsim_proc*, was chosen for our performance modelling which has been verified against a hardware prototype design [14]. Modifications were limited to our instruction definition and timing.

5.3 Feasibility

The TRIPS architecture allows for the placement of the instruction extensions within its instruction space without significant modification. All five instructions can share the primary opcode of like-formatted *G:2* instructions by occupying the last 5 spaces of the extended-opcode space. Thus allowing for shared decoding resources and preventing unnecessary artifacts as a result of additional architectural complexity.

The packed 64-bit SIMD instructions proposed can be reasonably expected to be non-critical on the processors cycle timing. This is primarily because they require at most as much logical depth as their already implemented non-SIMD counter-parts. We observe that the TRIPS *ADD* instruction is a 64-bit adder and thus has more depth than two 32-bit or eight 8-bit adders since the source of serialization is the carry-over bits. The fewer bits in the effective adder, the shorter the depth. Similarly, *ROTL* can be expected to be non-critical on the processors cycle timing because it is a commonly implemented bit-wise function.

6 Benchmarking

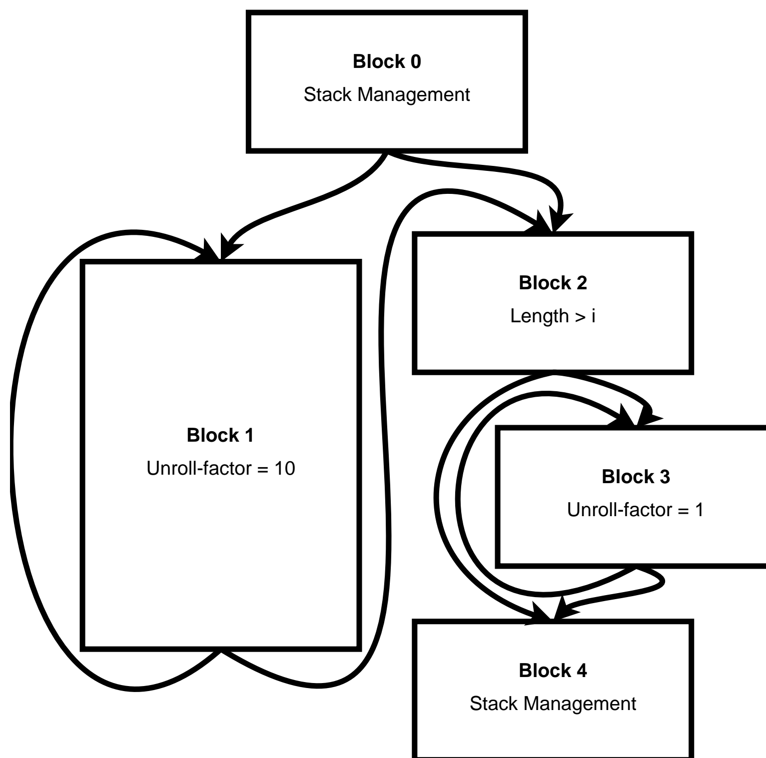

Figure 2: Control Flow for Vector Add and Vector Multiply in Optimized TRIPS Library.

To examine the peak performance potential of our extensions without contributing to the complexity of an optimizing compiler, hand optimization of the benchmark kernels was necessary.

A generic loop structure was employed as a model for hyper-block construction within the

TRIPS assembly code for all vector-based benchmarks. A simple loop was duplicated, unrolled, and unpredicated to optimize for long vectors and reduce the number of comparisons. A slower predicated loop was then necessary to guarantee functional correctness along vector boundaries. Reference figure 2 for a diagram of this generic control flow.

6.1 Vector Addition

Commonly used in a variety of DSP applications, vector-based addition was a natural selection for benchmarking our SIMD addition extensions. Our benchmark computes the element-wise sum of two arrays and writes the result to a third array. This operation exhibits element-wise independence making it an excellent benchmark for our packed `ADD2` and `ADD8` instructions.

To insulate our performance results from initialization overhead, we chose to run vector add on a large input size (3 arrays of 1-million elements in length).

6.2 Vector Multiplication

To complement our vector addition, we chose an analogous vector-based multiplication to benchmark our SIMD multiplication extensions. The element-wise product of two arrays is computed and written to a third array. Like its vector-add counter-part, it exhibits tremendous data-level parallelism and is thus an excellent choice for benchmarking our packed `MUL2` and `MUL8` instructions. Further, the similarities between vector-multiply and vector-add will allow us to study the impact of static instruction latency on overall performance.

Like vector-add, we chose to run this benchmark on a large dataset size - arrays of 1-million elements in length. This strategy will amortize the cost of initialization and expose the asymptotic performance characterization of the benchmark.

6.3 Block Rotation

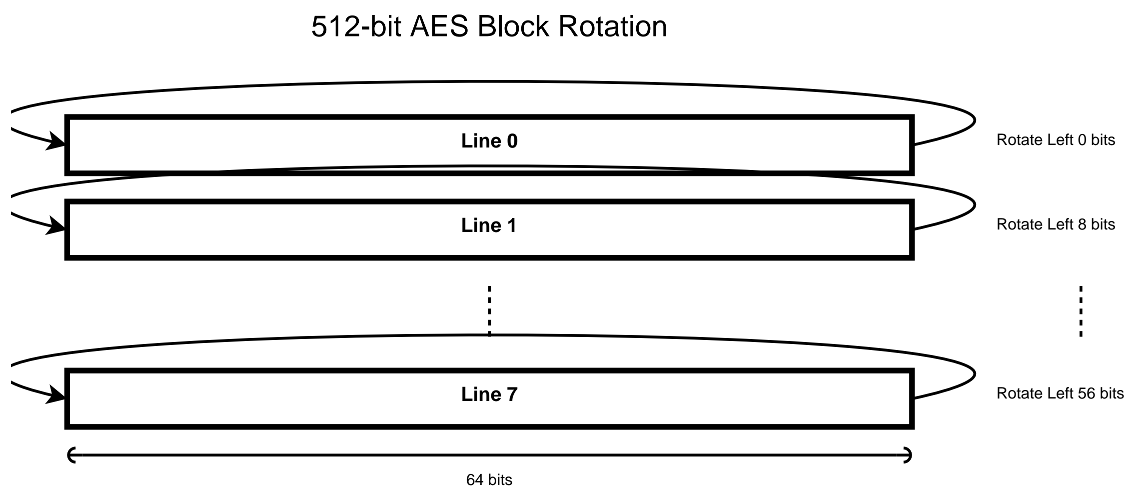

Figure 3: Overview of Block Rotation Benchmark.

Cryptographic applications such as AES encryption require bit-wise rotations of a lines of data in a block. We chose to benchmark our bitwise rotation extension by implementing a micro-kernel

of the AES encryption scheme. The micro-kernel performs 8 rotations of varying length on the lines of a 512-bit 2D matrix. We repeat this operation in a loop to amortize any initialization overhead. A high-level diagram of the algorithm can be seen in Figure 3.

7 Evaluation

7.1 Vector-add

ADD8 **(8-bit Elements)**			
	TIc64x	**TRIPS**	**TRIPS w/ Exts.**
Instructions	1,519,954	6,300,296	788,566
Blocks	-	100,035	12,605
Cycles	1,770,014	2,281,186	309,475
IPC	0.85	2.76	2.55
ADD2 **(32-bit Elements)**			
	TIc64x	**TRIPS**	**TRIPS w/ Exts.**
Instructions	6,031,886	6,300,296	3,150,406
Blocks	-	100,035	50,045
Cycles	8,467,848	2,563,735	1,318,081
IPC	0.71	2.46	2.39

Table 3: Vector Add

We first observe an expected result and find a strong correlation between the SIMD degree and the dynamic instruction count in vector addition benchmarks. The usage of the *ADD8* and *ADD2* extensions resulted in dynamic instruction reductions of a factor of 8 within 0.1% variation and 2 within 0.01% variation, respectively. A similar correlation is observed in dynamic block count as well.

Correlated, but not as strongly, proportional cycle reductions are observed in vector addition. With fewer cycles to buffer, the performance susceptibility to network contention on the TRIPS architecture increases. The small variation that is observed here can be accounted for through imperfections in the critical blocks' scheduling.

With support for a packed, 8-bit, *ADD4* instruction, the TIc64x achieves a reasonable instruction count - twice that of the TRIPS architecture with the *ADD8* extension. However, the TIc64x has support for only one half of the memory bandwidth as that of the TRIPS architecture and thus experiences a 2x slow-down in addition to the 2x instruction count. A net 4x speedup for the TRIPS architecture with the *ADD8* extension is reasoned and observed. The TRIPS architecture with the *ADD2* extension experiences a 2x speedup over the TRIPS baseline and a 4x speedup over the TIc64x on the 32-bit data size.

7.2 Vector-multiply

Similar to vector addition, the vector multiply benchmarks observe a tight correlation between SIMD degree and the dynamic instruction, block, and cycle count performance on the TRIPS architecture.

MUL8 **(8-bit Elements)**			
	TIc64x	**TRIPS**	**TRIPS w/ Exts.**
Instructions	2,002,455	6,300,311	788,581
Blocks	-	100,037	12,607
Cycles	2,113,151	2,256,872	309,993
IPC	0.94	2.79	2.54
MUL2 **(32-bit Elements)**			
	TIc64x	**TRIPS**	**TRIPS w/ Exts.**
Instructions	6,031,992	6,300,311	3,150,421
Blocks	-	100,037	50,047
Cycles	7,001,032	2,575,250	1,319,428
IPC	0.86	2.45	2.39

Table 4: Vector Multiply

the TIc64x suffers from resource contention in addition to memory bandwidth saturation in 8-bit vector multiplication. *MPY4*, as available on the TIc64x architecture is pipelined but only available on one half of the execution units within each data path. This accounts for the 8x performance speedup of the augmented TRIPS architecture relative to the TIc64x. Limited only by memory bandwidth and instruction count, we observe a 4x performance improvement on the TRIPS architecture relative to the TIc64x.

7.3 Block-rotation

ROTL **(512-bit Blocks)**			
	TIc64x	**TRIPS**	**TRIPS w/ Exts.**
Instructions	7,600,272	7,600,260	4,300,260
Blocks	-	300,023	300,023
Cycles	4,000,330	2,452,385	2,402,387
IPC	1.90	3.10	2.00

Table 5: Block Rotation

Unique to block rotation in our study, a very tight instruction reduction factor of 2x is not mirrored in dynamic block count. This is because the block constraints were already reached without the extension. While we reduced the necessary instructions to perform the task, we could not reduce the number of necessary blocks. Accordingly, we observe a small performance increase of 2.0%, despite a halving of the dynamic instruction count. The performance ratio across architectures is a reflection of the available memory bandwidth.

8 Conclusions

In this paper we have outlined a set of DSP extensions to the TRIPS ISA and evaluated their performance. We have reasoned about the feasibility of these extensions and modeled how they can fit in

the TRIPS instruction space. As an explorative study, these extensions have demonstrated that the performance benefits of conventional DSP specialization techniques can be observed in an EDGE ISA.

We evaluated the performance of the augmented TRIPS ISA by comparing our results to those of an industry standard DSP architecture, the TIc64x. While both architectures demonstrated an insensitivity to static instruction latency, we discovered bandwidth bottlenecks and observed performance ratios accordingly.

By employing hand optimization and custom written microbenchmarks, we were able to observe degree order performance gains from SIMD extensions to the TRIPS ISA. In addition, we observed a correlation between dynamic block count and performance in the TRIPS ISA. We believe that this demonstrates the flexibility of the TRIPS architecture to absorb excess dynamic instructions by leveraging redundant execution units. Accordingly, extensions which simply reduce dynamic instruction count (such as *ROTL*), are unable to realize any significant performance gains and should be avoided. Further, we propose that future architectural augmentations focus on reducing the demand on the memory system.

9 Future Work

A way of reducing the stress on the memory system is to reduce the utilization rate of the OPN. Future work should investigate two alternative methods for doing this.

First, a new addressing mode could be defined. Such an addressing mode would push memory addresses to small registers embedded in, and replicated across, the data-cache tiles. Fewer bits would then be required to specify a memory access. The saved bits could then be re-employed in a number of ways, such as to name additional operand buffer targets.

Alternatively, by redefining a store instruction to release two independent packets (target address and value), 40 bits of the OPN could be reclaimed. A complementary, but trivial, modification to load instructions would be necessary. The reclaimed OPN bandwidth could then be turned around and augmented with little cost to router complexity or chip area. OPN links might then be able to carry two packets per cycle, doubling the effective bandwidth.

References

[1] Texas Instruments, TMS320C6416 Digital Signal Processor. December 2001.

[2] AMD. *3DNow! Technical Manual*, 2000.

[3] AMD. *AMD Extensions to the 3DNow! and MMX Instruction Sets*, 2000.

[4] J. D. Broesch. *Digital Signal Processing Demystified*. Newnes, March 1997.

[5] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, pages 44–55, July 2004.

[6] K. B. Bush, M. Gebhart, D. Burger, and S. W. Keckler. A Characterization of High Performance DSP Kernels on the TRIPS Architecture. Technical Report TR-06-62, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, November 2006.

[7] K. B. Bush, M. Gebhart, E. Wei, N. Yudin, B. Maher, N. Nethercote, D. Burger, and S. W. Keckler. Evaluation and Optimization of Signal Processing Kernels on the TRIPS Architecture. In *Proceedings of the Annual Workshop on Optimizations for DSP and Embedded Systems (ODES)*, March 2006.

[8] K. Coons, X. Chen, S. Kushwaha, K. S. McKinley, and D. Burger. A Spacial Path Scheduling Algorithm for EDGE Architectures. In *The Twelth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[9] IBM. *Power ISA, Version 2.04*, 2007.

[10] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS Processor Reference Manual. Technical Report TR-06-62, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, November 2006.

[11] K. S. McKinley, J. Burrill, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The Scale Compiler. Technical report, Department of Computer Sciences, University of Massachusetts, University of Texas, 2005.

[12] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, May 2003.

[13] TRIPS. Department of Computer Sciences, University of Texas. http://www.cs.utexas.edu/users/trips, 2006.

[14] B. Yoder, J. Burrill, R. McDonald, K. B. Bush, K. Coons, M. Gebhart, S. Govindan, B. Maher, R. Nagarajan, B. R. K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software Infrastructure and Tools for the TRIPS Prototype. In *Third Annual Workshop on Modeling, Benchmarking, and Simulation*, June 2007.

A ADD2 - Packed Add 2

Format: G:2
Instruction Type: General
Number of Targets: 2
Primary Opcode: 23
Extended Opcode: 26

Figure 4: ADD2 Format

Description:
The OP0 value is added to the OP1 value via a 64-bit adder with carry-overs on 32-bit boundaries. The result is two 32-bit sums corresponding to the 32-bit addition of bits 0-31 of OP0 and OP1, and of bits 32-63 of OP0 and OP1. Arithmetic overflows are ignored on 32-bit boundaries.

Operation:
T0, T1 $\leftarrow$ *OP0* $+_{2x32-bit}$ *OP1*

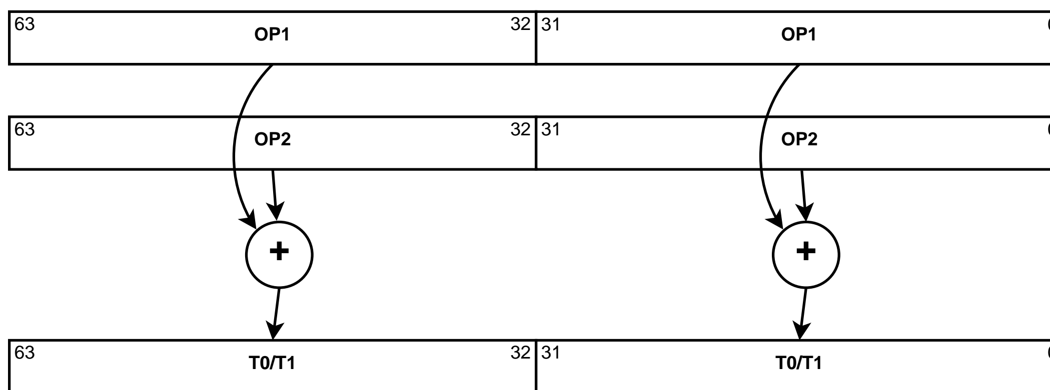

Figure 5: ADD2 Operation

Exceptions:
None.
Notes:
None.

B ADD8 - Packed Add 8

Format: G:2
Instruction Type: General
Number of Targets: 2
Primary Opcode: 23
Extended Opcode: 27

Figure 6: ADD8 Format

Description:
The OP0 value is added to the OP1 value via a 64-bit adder with carry-overs on 8-bit boundaries. The result is eight 8-bit sums corresponding to the 8-bit addition of bits 0-7, 8-15, 16-23, 24-31, 32-39, 40-47, 48-55, and 56-63 of OP0 and OP1. Arithmetic overflows are ignored on 8-bit boundaries.

Operation:
T0, T1 $\leftarrow$ *OP0* $+_{8x8-bit}$ *OP1*

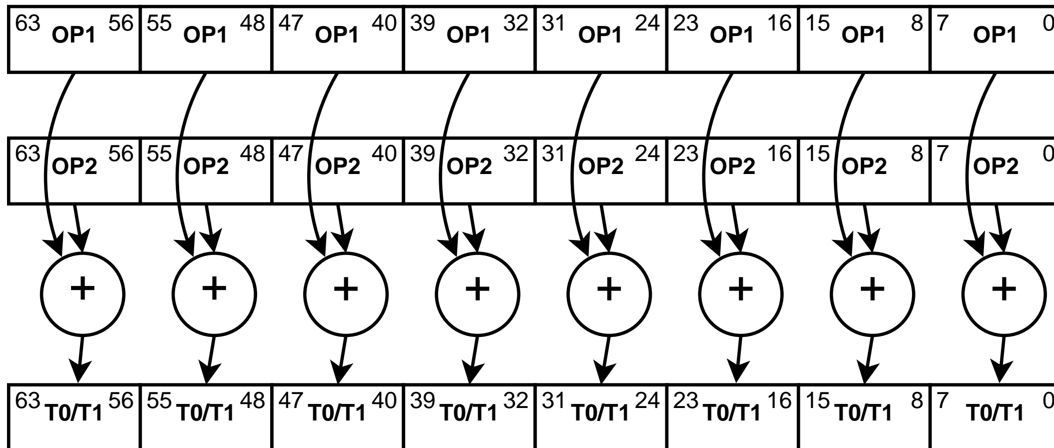

Figure 7: ADD8 Operation

Exceptions:
None.

Notes:
None.

C MUL2 - Packed Multiply 2

Format: G:2
Instruction Type: General
Number of Targets: 2
Primary Opcode: 23
Extended Opcode: 28

Figure 8: MUL2 Format

Description:
The OP0 value is multiplied by the OP1 value via two 32-bit multipliers. The result is two 32-bit product corresponding to the 32-bit multiplication of bits 0-31 of OP0 and OP1, and of bits 32-63 of OP0 and OP1. Arithmetic overflows are ignored on 32-bit boundaries.

Operation:
$T0, T1 \leftarrow OP0 *_{2x32-bit} OP1$

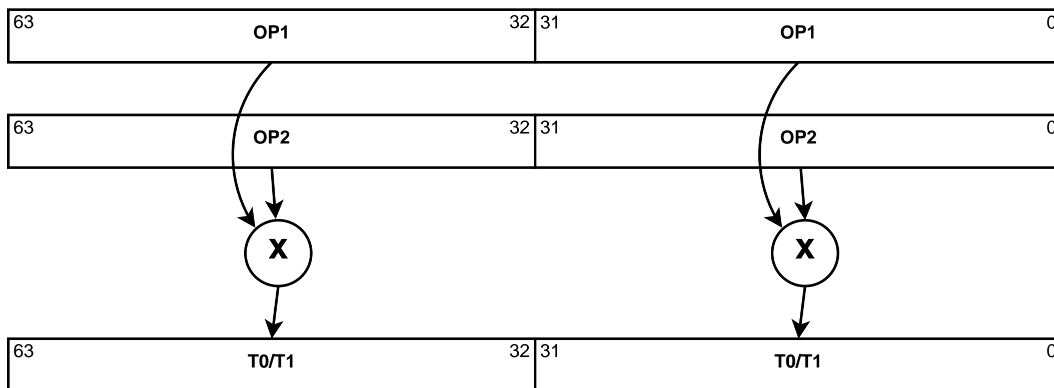

Figure 9: MUL2 Operation

Exceptions:
None.

Notes:
None.

D MUL8 - Packed Multiply 8

Format: G:2
Instruction Type: General
Number of Targets: 2
Primary Opcode: 23
Extended Opcode: 29

Figure 10: MUL8 Format

Description:
The OP0 value is multiplied by the OP1 value via eight 8-bit multipliers. The result is eight 8-bit products corresponding to the 8-bit multiplication of bits 0-7, 8-15, 16-23, 24-31, 32-39, 40-47, 48-55, and 56-63 of OP0 and OP1. Arithmetic overflows are ignored on 8-bit boundaries.

Operation:
T0, T1 $\leftarrow$ *OP0* $*_{8x8-bit}$ *OP1*

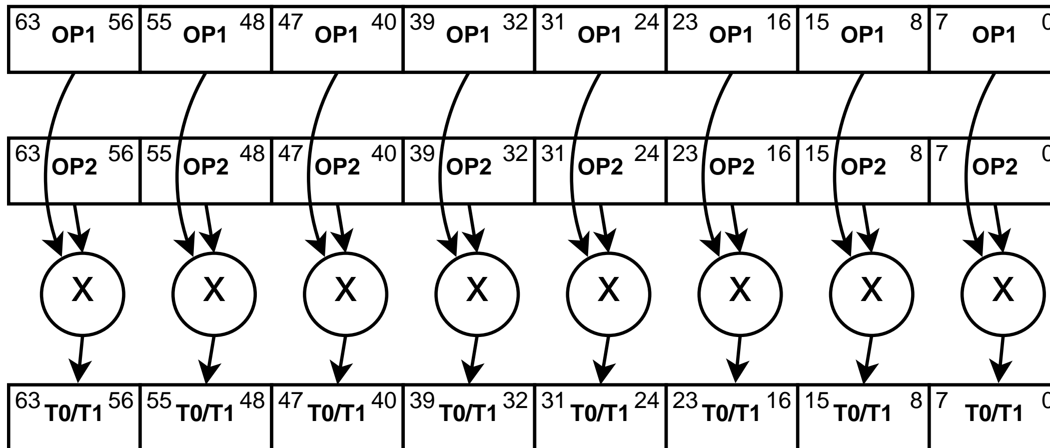

Figure 11: MUL8 Operation

Exceptions:
None.

Notes:
None.

E ROTL - Rotate Left

Format: G:2
Instruction Type: General
Number of Targets: 2
Primary Opcode: 23
Extended Opcode: 30

Figure 12: ROTL Format

Description:
The OP0 value is rotated (logically) left by the number of bits specified by the unsigned representation of the lowest 7 bits of OP1. All remaining bits of OP1 are ignored. A logical shift-left is performed, however, the padded low-order bits are replaced by the lost high-order bits. Maximum shift range of [0-63].

Operation:
T0, T1 $\leftarrow$ *OP0* $Rotate - Left_{OP1-bits}$

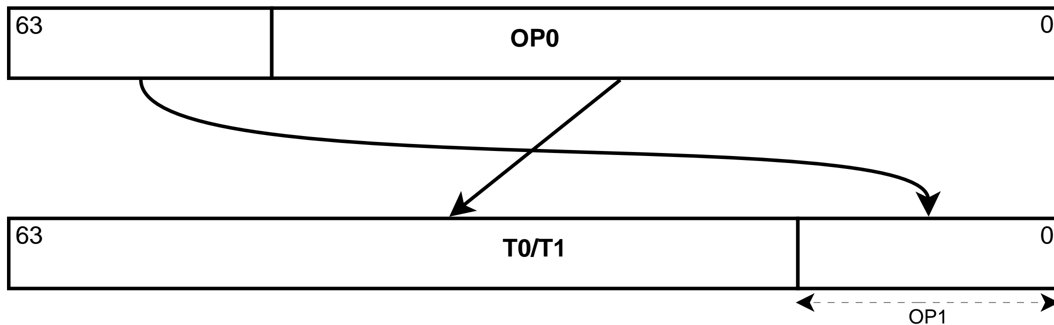

Figure 13: ROTL Operation

Exceptions:
None.
Notes:
None.