A Fault Tolerant Core for Parallel Execution of Ultra Reduced Instruction Set (URISC) and MIPS Instructions

by

Nathan Daniel Pozniak Buchanan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis presents an out-of-order processor core design that accelerates the reliable computation application focused URISC-MIPS core. The URISC-MIPS core provides a hardened Ultra Reduced Instruction Set Core (URISC) along side a simple MIPS core. The URSIC instructions are used to check for faulty MIPS instructions/execution units and retain functionality by replacing faulted MIPS instructions with equivalent URISC instructions. Previous work presents a URISC-MIPS core limited to executing a single URISC or MIPS instruction per cycle. This thesis analyses the causes for this limitation and proposes a core design that accelerates computation by overcoming this limitation. A compiler interleave step is additionally introduced to better utilize execution units. The resulting core better utilizes the hardware and unlocks parallelism by executing URISC and MIPS instructions in parallel. Multiple benchmarks utilizing URISC instructions are executed to compare the performance of the new core relative to the original URISC-MIPS core. The results show speedups between 1.16 and 4.22 (geometric average 1.90 times speedup) with the new design.

Acknowledgements

This thesis would not have been possible without the guidance, advice and help of a number of individuals who assisted me greatly in the completion of this thesis.

First and foremost I would like to thank my advisor, Professor Hiren D. Patel, for his direction and inspiration throughout my master's program. Your strong support and encouragement has been instrumental in my research. I have learned much from you.

I would also like to thank my colleagues in the Computer Architecture Research Group for sharing their presentations, lively discussions, and insights.

A further heartfelt thanks goes out to my employers and managers during the program. Their support and flexibility with my work schedule allowed me to take my classes and attend important presentations.

Last, but definitely not least, I am sincerely appreciative for my parents, Ela and Paul Buchanan, for their unwavering support and love throughout this program, and indeed throughout my life.

Dedication

This thesis is dedicated to my parents, Ela and Paul Buchanan, for their constant support, inspiration and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern safety critical systems require the ability to detect and handle situations where an error has occurred. Common applications needing this capability include health care, transportation (space applications, aviation, vehicles, etc), and automated machinery (industrial control, robotics, nuclear power stations, etc.). Essentially anywhere where a failure of the computer system could cause bodily harm or loss of life needs the computer system to be able to recognize when things go wrong and recover. Depending on the severity of the error, the solution may be to simply retry the transmission or computation or to admit that things have gotten out of hand and automatically shut down the nuclear power plant or disengage the autopilot and turn over control to the operator.

Modern computer and communication systems employ multiple error detection and correction features to protect the communication, storage and computation system components from errors.

External communication links are generally protected by channel coding to detect and correct errors.

Storage elements such as hard drives, external disks, and flash storage employ techniques such as RAID, parity checks, and/or error correcting code (ECC) while storing and retrieving data. On chip memory (including caches) may employ parity checks and ECC. On chip buses can extend the error detection across the bus by carrying parity bits to

protect the control and data lines.

Computation units, such as computer cores, may have a variety of features for fault detection and correction implemented as well. Solutions range from modular redundancy (usually Triple Modular Redundancy where three cores independently compute the result), to fine grained detection and protection on different core control and computation units or structures.

This thesis will focus on one approach to perform fine grained protection of computational core units. The approach in question is to extend a core with a hardened Ultra Reduced Instruction Set [1] (URISC) co-processor, forming a core called TigerMIPS-URSIC [2].

The TigerMIPS-URISC core can protect against both permanent and transient faults effectively. It is designed to be operated in conjunction with a customized compiler and assembler that selectively adds instructions to check or replace MIPS instructions. However, previous designs add significant performance overheads when using URISC instructions to check for or replace defective MIPS instructions.

1.1 Contribution

This thesis analyses the overhead introduced in the TigerMIPS-URISC core design to identify opportunities to accelerate the computation. Based on the analysis we determine that an out-of-order core design would accelerate fault tolerant MIPS and Ultra Reduced Instruction Set Coprocessor (URISC) programs.

We design and build an out of order core supporting both MIPS and URISC instructions. This new core effectively exploits the parallelism available in MIPS-URISC programs and significantly reduces the overhead introduced when checking or substituting URISC instructions for faulted MIPS instructions.

1.2 Thesis Structure

The following chapters of this thesis are structured as follows: Chapter 2 provides background on the causes of permanent and transient errors in chips. It also provides an overview of the approaches examined in literature to address these errors in processor cores. Chapter 3 provides an in depth review of the URISC SUBLEQ instruction and the TigerMIPS-URISC core.

Chapter 4 analyses the TigerMIPS-USRIC implementation described in [2] and outlines opportunities for running instructions in parallel. The challenges of running these instructions in parallel along with solutions are considered to accelerate the core.

Chapter 5 describes the proposed solution core design to accelerate the TigerMIPS-USRIC core. Chapter 6 describes compiler modifications to fully exploit the changes in the new core.

Simulation results comparing our core to the previous implementation are presented in Chapter 7 to demonstrate the speedup of the new core. We conclude in Chapter 8 and provide areas for future research.

Chapter 2

Background and Related Work

Today's digital designs are steadily increasing in complexity and gate count. To keep up with this increasing gate count, CMOS process technology is aggressively scaling down transistor feature sizes. As transistors get smaller they become more susceptible to faults for the reasons described later in this chapter. To produce operational large designs each and every component of a chip needs to operate correctly or be resilient to faults. With more transistors on a chip, the probability that all transistors in a large design are fault free decreases. Without techniques to handle faults, the faults will impact chip yield (permanent faults) and chip reliability (transient faults).

This chapter provides background on fault sources and techniques to detect and recover from faults in the major components in a digital system: communication links, storage, and computation. Some circuit level hardening techniques are described enabling certain circuits to be made more robust and an out of order design technique is outlined that makes better use of core execution units. Using these techniques an operational system can be retained even in the presence of various faults.

2.1 Faults

Faults can be categorized as:

- **Permanent Faults** - where a wire or register is permanently stuck either high or low.
- **Transient Faults** - where a wire or register is sometimes at the wrong value.

2.1.1 Permanent Faults

Permanent faults are primarily caused by design errors or manufacturing/silicon defects.

Design errors are cases where the value of a wire or register is not at the desired value. This may be because of a mistaken specification, incorrect implementation, synthesis, or verification escape. Also included here are data dependent worst case timing paths where the timing is not met under some conditions (although this may present itself as a transient fault, the path should fail every time given the same inputs - this is still a permanent fault).

Manufacturing or silicon defects are another cause of permanent faults. These are localized defects in the silicon substrate (impurities or silicon structure irregularities) or irregularities in a layer's deposition or etching steps. Silicon substrate defects are typically not a function of the transistor size of the process in use. When working with large feature sizes, a given defect may not be large enough to affect the transistor, but as the transistor size shrinks, the same defect will be large enough to cause an fault in the transistor. As the transistor sizes shrink and density increases, the same defect may even be large enough to affect multiple adjacent transistors at once.

Most permanent faults can be detected shortly after fabrication during testing. Those chips with faults that can't be compensated for will decrease the yield.

2.1.2 Transient Faults

Transient faults can be caused by both external and internal interference or ageing effects.

Magnetic fields or an energetic particles can cause transient faults. A strong enough external magnetic field can introduce a momentary incorrect value on a trace. A stray highly charged particle that impacts a transistor can deposit charge where it should not be and cause a momentarily incorrect value. If this impacts the transistor in a flip flop, the stored value could be altered. These external effects are typically a rare occurrence in most environments, however some harsh environments (space, nuclear) will see a higher rate of these faults.

Internal transient faults are caused by either crosstalk or timing performance degradation. Crosstalk between adjacent wires could introduce a transient error in the wire value and potentially the flip flop capturing the value. Timing performance problems such as Electromigration and Bias Temperature Instability (BTI) are caused by use and age. Electromigration is the gradual movement of metal ions in traces due to current flows, gradually thinning portions of the conductors. This first leads to higher resistance resulting in slower voltage changes, and eventually an open circuit or short if enough metal migrates. Bias Temperature Instability changes the threshold voltages of transistors over time causing small shifts in timing. Both these affects can be exacerbated due to manufacturing defects that can leave transistors in a marginal state - sometimes causing an error and sometimes operating correctly. As the transistor continues to age, the degradation may become severe enough to become a permanent fault.

Environmental conditions, such as temperature and voltage changes can exacerbate the problems described in the previous paragraphs. While the area affected by sources such as highly charged particles and silicon defects remains the same, the number of transistors in that same area are increasing, leading to more transistors subject to fault in equal area designs.

2.2 Fault Protection in Communication Links

Communication links in digital systems range from internal chip buses (custom serial, AHB, APB, AXI, ...), chip to chip buses (serial, PCI, SCSI, I2C, SPI, DDR, GDDR, ...) all the way to noisy radio links. Faults on these links are primarily transient faults due to

external interference.

On internal chip buses and chip to chip buses the error rate is generally small and parity bits, CRC checksums, or ECC (typically Hamming code) are sufficient to detect them. Recovery is accomplished by applying the ECC or by repeating the transaction.

On noisy radio or other external links with high error rates more powerful error correcting code is applied to both increase reliability and better utilize the channel capacity. Techniques such as convolutional coding, Viterbi, turbo codes, and Low Density Parity Check (LDPC) coding allow communication channels to approach the channel capacity limit for various error rates. As with low error rate channels, retransmissions are employed if the error correction code is unable to recover the data given a particular set of errors.

2.3 Fault Protection in Storage Components

Storage components range from removable disks (optical or magnetic), magnetic hard drives, flash (SSD) drives, RAM (either on or off chip), and caches.

In these components we are concerned with both permanent faults rendering a portion of the memory unusable and ageing of the storage media or interference changing the state of stored information.

Large storage volumes can use parity check approaches such as RAID (1-6) to protect against loss due to errors. RAID either duplicates and distributes the data or stores enough parity bits on different disks to recover the data from a lost disk. Although historically applied to multiple separate disks, the same techniques can be applied to the multiple chips in flash SSDs. Individual words (or blocks of words) stored on disk are often stored with some form of small ECC to protect against single (or a few) bit flips in the underlying storage media. With rapid advance of flash storage and use of denser and more error prone multi-level cells for storage, ECC (often BCH code) has become a necessity.

On chip memories (RAM and caches) store smaller pieces of information for shorter periods of time. Protection is applied in smaller groups of bytes, typically in the form of parity bits or small ECC (such as Hamming code) such as that used in ECC RAM.

All of the above storage components can also have a way to mark sections of the storage media as faulted so the affected addresses can either be avoided or remapped to another backup block of memory.

As the amount of storage in today's systems continues to grow (increasing the area of the chip dedicated to memory), the error detection and correction techniques mentioned above become a necessity to achieve acceptable yields.

2.4 Fault Protection in Computation Components

With error protection available for the communication links (subsection 2.2) and storage (subsection 2.3), the majority of remaining unprotected digital area on the chip is in the computation components - processor cores and accelerators. In most cases a faulted special purpose accelerator's function can be taken over by another accelerator or the more general purpose processor core. However, the processor core's function usually can't be taken over by the special purpose accelerator. Since a fault in an unprotected processor core can not be compensated for with any other component on the chip, we'll focus on fault protection techniques for processor cores.

The techniques detailed in this section range from techniques designed purely to detect errors to those that both detect and correct errors and to those that seek to increase throughput in the fault free case while gracefully degrading their performance while remaining operational in the presence of faults.

2.4.1 Dual and Triple Modular Redundancy

One of the oldest approaches to fault detection is to create multiple copies of the core. Each core is asked to do the same computation and the results are compared. If the results differ, an error has been detected and some error handling process begins. With Dual Modular Redundancy (DMR) there are two copies of the core that can detect errors. Adding a third core produces Triple Modular Redundancy (TMR) and allows voting on the results to judge one of the three results in error.

The NASA Space Shuttle uses a triple redundant system for flight control with the flight control actuators using the three results from the three systems to vote on the action taken. The system that returns a different answer is out voted and an error is indicated to the crew so they can power down or reset the faulty system. This approach protects against both permanent faults and transient faults.

The biggest disadvantage of DMR or TMR is the the amount of overhead involved - area and power are both doubled or tripled. The applications where this increase in area and power are accepted has historically been limited to risk averse applications such as flight control, nuclear power plants, or machine control where a failure would cause risk injury or death.

2.4.2 Razor

The Razor framework[3][4] proposes increased performance by adding special detection circuits to the longest timing paths to detect when timing is violated. The clock rate is dynamically increased until timing errors are observed then reduced to keep the system just barely meeting timing. When a timing error is detected, recovery happens by slowing the clock rate and repeating the calculation.

When chips are designed they are typically designed to their worst case temperature/voltage delay corner and some margin is added for timing uncertainty of the process. The Razor approach is able to extract additional performance out of a chip by operating the chip at the maximum clock rate for the actual operating temperature/voltage conditions and adapting when they change.

Additional overhead is modest, as circuitry is only added to the longest timing paths. This approach is able to detect transient faults due to timing, but is unable to detect or correct permanent faults or faults due to interference.

2.4.3 Detouring and Salvaging

The Detouring and Salvaging approaches are methods to retain functionality in the presence of permanent faults. They rely on a built in self test (BIST) component or a test

program and a set of pre-defined test vectors to identify which parts of the core are in broken. Once a hard fault has been detected, yield can be increased by working around the defective component. These approaches are targeted at smaller cores.

The Detouring [5] approach chooses to recompile a program to avoid broken instructions and therefore can continue to run on a core with defective instructions. While many instructions are replaceable by other instructions in the instruction set, it is limited by the fact that not all instructions have substitute instructions in the instruction set.

The Salvaging [6] approach is applicable to a multi-core system. It requires an OS aware of the faulted instructions for each core and which instructions are needed by which portions of a program or thread. When a program (or thread) needs to execute instructions that are broken in the current core, the OS swaps the thread to a different core where those instructions are not broken.

The Necromancer [7] approach uses a broken complex core and a known good simple core to try to predict when the broken core gets off track. The simple core trails the complex core's execution utilizing the complex core's execution path to avoid stalls around branch prediction and memory access. If an incorrect path is detected a recovery is then started.

2.4.4 Microarchitectural Redundancy and Structural Duplication

With super-scalar processors many components of the core are already duplicated for performance reasons (such as execution units, cache blocks, etc). If a unit in the core is found to be defective, it is simply disabled, and other identical units are scheduled to perform the same operation [8]. This approach allows a core to operate at full capacity when no faults are present but to decay gracefully when certain faults occur. It is limited by the number of duplicated components in a core. Further work [9] proposes cores designs that intentionally ensure each component is fully duplicated for better fault tolerance.

The Bulletproof[10] approach for super-scalar processors introduces a checkpointing approach with built in self tests being run on execution units during idle cycles. The

purpose is to detect hard faults developing in a core with minimal impact to the running programs. Once a full self test has completed successfully, a new checkpoint is created and processing continues from the new checkpoint. If the self test fails, the failing execution unit is marked as failed preventing instructions from being issued to it, the saved checkpoint is restored and the computation is restarted from that point.

2.4.5 DIVA

The DIVA concept[11] proposes adding a small in-order checker core to a superscalar out-of-order core that checks each instruction in the ROB as it retires, preventing an instruction from retiring if it isn't correct.

The DIVA concept was further extended to perform online checks of a core and disable faulty components[12].

2.4.6 Argus

The Argus [13] approach looks at small simple cores and adds small hardware checkers to check hardware invariants. It only detects a percentage of faults and can not correct them.

2.4.7 URISC

The DIVA approach works well for large cores as the addition of one very small in-order core adds little overhead. When DIVA is applied to a simple core the overhead of the DIVA core becomes similar to that of the simple core.

An approach that reduces the overhead suitably is to use a very small core, such as an Ultra Reduced Instruction Set (URISC) [1] core along side the simple core. This approach is detailed in [2][14][15] and can both detect and recover from faults. It uses a URISC instruction and a compiler that can be tuned to output programs with different levels of protection. A program can be compiled to detects and corrects all errors for cases where maximum reliability is required at the cost of additional instructions. A self test program

can be combined with the detouring approach[5] to allow the core to work around a broken instruction. Notably, a URISC core is Turing complete, so it can always work around a broken instruction.

A URISC core adds little hardware overhead as the hardware to add the URISC instruction is small, even relative to a simple core. The main disadvantage of a URISC core is the time overhead of checking instructions is large.

In this work we observe that the URISC checker instructions have significant instruction level parallelism (as described in the next section), and we can execute URISC and other instructions in parallel to provide significant speedups.

2.5 Circuit Level Hardening Techniques

Critical circuits can be hardened at the circuit level with specially designed logic and storage cells. Dual Interlocked storage Cell (DICE) [16] hardened designs show reduced error rates by two orders of magnitude by using redundant interlocked storage cells. Theses cells use state restoring circuitry that corrects an errant value on one of the constituent storage cells. This technique is a circuit level modular redundancy approach with the redundancy applied at the logic cell level instead of at a system level. Further layout optimization produced the Layout design through Error Aware Positioning (LEAP-DICE) design[17] that maintains the error tolerance down to smaller feature sizes[18].

As these techniques are a form of modular redundancy they impose a significant area and power overhead (approximately double), so should be used where the extra protection is most required. Depending on the design all or only a portion of the circuits of a design may be protected.

2.6 Out-of-Order Execution

The first core designs used in-order execution where instructions are fetched and executed in program order. If an instruction or instruction or operand is not available, execution

stops at that point in the pipeline until the missing item is available. In-order core designs are still used today for many low power applications.

As core designs progressed and got bigger, techniques first described by Tomasulo[19] introduced out-of-order execution to make better use of the available execution units. In out-of-order execution, instructions are continually fetched, decoded and the referenced operands are matched to the instruction. The instructions are added to a reorder buffer, assigned to an execution unit, and the next instructions are fetched. The reorder buffer allows the fetch unit to continue fetching instructions even if some operands for an instruction are not yet available. When the operands for an instruction become available, then that instruction can complete. To keep track of intermediate register values, the registers are renamed. The reorder buffer keeps track of the instruction order and enables the instructions to exit the core (or retire) in program order.

Speculation past branches is typically used to allow instructions after a branch to be executed in anticipation of a certain branch direction. If the branch was predicted correctly, the already executed instructions already have the work completed and can be retired. If the branch was mis-predicted, the core simply throws out the speculative calculations and fetches the actual instructions it needs to execute and starts work on those.

Chapter 3

TigerMIPS-URISC Core

The concept of an Ultra Reduced Instruction Set core was first proposed in [1] as a teaching tool for microprocessor design. It is unique because the simplest version of the core consists of a single Turing complete instruction. The fact that it is Turing complete means the core can execute any program that a more complex core can compute, albeit with more instructions. These cores are also known as One Instruction Set Cores (OISC). We will use the term URISC when referring to them.

A URISC core can be created with an instruction that both does a computation and can affect the next instruction executed. The most common instruction of this type is the subleq instruction as originally proposed in [1] and used extensively in [2][14][15].

3.1 Subleq

The subleq instruction is defined as follows:

subleq a, b, c;

First, a is subtracted from b, and the result stored in b. If the result is less than or equal to zero, the machine jumps to the next instruction location specified by c. If the result is greater than zero, the machine continues executing the next instruction without jumping.

3.2 URISC for Fault Detection and Recovery

Since a URISC core can perform the same tasks as any other program, we can use it to check for errors and recover from them.

The TigerMIPS-URISC core as proposed in [14] pairs an in-order MIPS core with a hardened URISC core as a co-processor. The hardened URISC core would be constructed with more reliable (or duplicated) logic cells and would therefore be much more robust to faults. As it only includes one instruction, the amount of logic to protect is small and the area and power overhead is not large.

Each MIPS instruction has an equivalent sequence of subleq instructions that perform the same operation. For example, the MIPS add instruction can be computed as follows:

MIPS Instruction	**Equivalent URISC Instructions**
ADD rd, rs, rt	mtu0 rs u1
	mtu0 rt u2
	subleqi u3 u3 0
	subleqi u2 u3 0
	subleqi u3 u1 0
	mfu0 u1 rd

The MIPS instruction adds rs and rt and stores the result in rd. In this example, the equivalent URISC instructions first move the rs and rt operands to URISC registers u1 and u2. u3 is then zeroed, u2 is negated into u3, then the negated u2 in u3 is subtracted from u1 producing the sum of rs and rt. The result u1 is then copied back out to rd.

A program can be compiled such that it checks MIPS instructions by adding the equivalent URISC instructions and URISC instructions to compare the MIPS instruction result. If the check fails a recovery may be performed. Such a recovery would normally take the form of using the more robust answer from the URISC core in place of the MIPS result. If a particular core has an execution unit that is faulted, the program can be recompiled using a Detouring[5] like approach to avoid that execution unit. Unlike the original detouring paper, the TigerMIPS-URISC core has the capability to replace any faulty instruction since any instruction can be replaced with a series of subleq instructions.

Instruction checks can be applied to all instructions or only a subset, simply by providing the compiler the appropriate arguments. This enables solutions that check and recover on each instruction (both permanent and transient fault protection) to solutions only handling permanent faults. Additionally, only a portion of the instructions could be checked, allowing approaches that statistically check for failing instructions.

3.3 Instruction Check Overhead

The main overhead of checking an instruction is in computing the result twice, followed by a comparison. Unlike a DMR system that doubles the hardware to check the result, the TigerMIPS-URISC core uses the simple URISC core that has a very small hardware area in exchange for running a sequence of instructions to perform the check.

By looking at the add instruction above we note that the MIPS instruction takes 1 cycle, while the URISC equivalent takes 3 cycles plus an additional 3 cycles to move the arguments between cores.

As both the URISC and MIPS core can access the same registers or cause a jump, the TigerMIPS-URISC core in [14] executes a single instruction at a time - either a MIPS or a URISC instruction. This means that adding a check for an instruction will add at least 100% overhead to any instruction check. In the case of the add instruction above, the overhead is 600%.

Chapter 4

Objective

The objective of this thesis is to present a method to accelerate the TigerMIPS-URISC core. We do this by noting three main optimization areas:

- In the TigerMIPS-URISC core: when a MIPS instruction is executing the URISC hardware is idle, and conversely when a URISC instruction is executing the MIPS hardware is idle. The URISC and MIPS instructions are meant to compute the same result, thus there are opportunities to run these instructions in parallel.
- In most programs, a statistical check will likely be used where only a portion of the MIPS instructions are checked. In these cases, there will be many periods where no corresponding URISC check instructions are generated for a given MIPS instruction. When a MIPS instruction does need checking there will be a longer sequence of URISC instructions that currently prevent MIPS instructions from running until the entire URISC check sequence is complete. It would be advantageous to execute the URISC check instructions for a MIPS instruction in parallel with the following MIPS instructions, thereby hiding the time taken to check the instruction.
- A MIPS instruction replaced with URISC instructions should be able to be run in parallel with other MIPS instructions. This offers the opportunity to speed up existing MIPS programs.

Therefore we would like a core that can execute the MIPS instruction in parallel with the URISC check instructions. In addition, we would like the instruction check not to prevent the MIPS core from making forward progress - basically, we'd like to defer committing the results of the MIPS instruction until the check has completed.

4.1 Challenges

There are two basic challenges to running subleq instructions in parallel with MIPS instructions. First each and every subleq instruction can branch, and secondly, register values need to be transferred from the MIPS register set to the URISC register set leading to hazards when instructions are executed in parallel.

4.1.1 Challenge 1: Each Subleq is a Branch

Since each subleq instruction

subleq a, b, c;

jumps to instruction c if the result of a-b is less than or equal to zero, each and every subleq instruction is a conditional branch. We also note that MIPS instructions may be branches as well. To successfully run MIPS and URISC instructions in parallel we need to handle jumps and branches efficiently. For example, we can not simply ensure that branches are executed on their own because this would mean each subleq instruction (a branch) could not be run in parallel with another instruction.

To handle this we introduce speculation into the solution core. We choose the simplest form of speculation. A subleq rarely has the branch taken, so we speculate that the branch is not taken for any conditional branch (including subleqs). Jumps are always taken, and therefore are speculated as taken.

To implement speculation we introduce a Reorder Buffer (ROB) into the core. As instructions are completed, they are marked complete in the ROB (along with whether the branch was taken). As the instructions are retired from the ROB, the branch speculation

is checked. If the speculation was incorrect, the processor pipeline is cleared and restarted from the correct address.

4.1.2 Challenge 2: Hazards

Since URISC and MIPS instructions need to access the same argument and result registers we recognize the need to protect against Read After Write (RAW) and Write After Write (WAW) hazards when executing instructions in parallel.

To protect against these hazards we implement register remapping and a one entry issue queue for each execution unit in the core.

For each architectural register we store it's remapped location. The location for register values are in the register file or in the ROB.

When an instruction is issued, the input arguments are taken from the register file (if they aren't remapped), or from the remapped ROB location. The ROB location could be still pending.

An instruction will remain in the execution unit issue queue until all of it's arguments are available (we may be waiting on another instruction to write it's result to the ROB).

When an instruction is retired from the ROB, the register value stored in the ROB is written out to the architectural register file. In addition, the register remapping for the destination register is checked to see if the current ROB entry is the last remapping of the register location. If the last register remapping points to the current ROB entry, we remove the remapping as we retire the ROB entry.

Chapter 5

Solution Core

The core shown in Figure 5.1 shows the core design proposed to accelerate MIPS and URISC instructions. The core is an out-of-order core with a single fetch unit, separate decode circuitry for MIPS and URISC instructions, separate register files, and separate execution units for MIPS and URISC instructions. The memory access unit is treated as simply another execution unit. A reorder buffer (ROB) and common data bus (CDB) complete the core.

5.1 Fetch

The core's fetch unit is shown in Figure 5.2.

The core fetches from a 32 bit address space, fetching two instructions (64 bits) on each fetch. By fetching two instructions on each cycle, approximately twice the throughput of instructions can be processed when they are of different types. The fetched instructions are captured and presented to the decode/issue unit along with whether the instruction needs to be issued (not already issued) on the next cycle. Feedback is provided from the decode/issue unit as to whether one or both instructions were issued in the current cycle.

The instruction address to fetch, iAddr, increments by 8 for on each cycle with the following exceptions:

Figure 5.1: Proposed Core

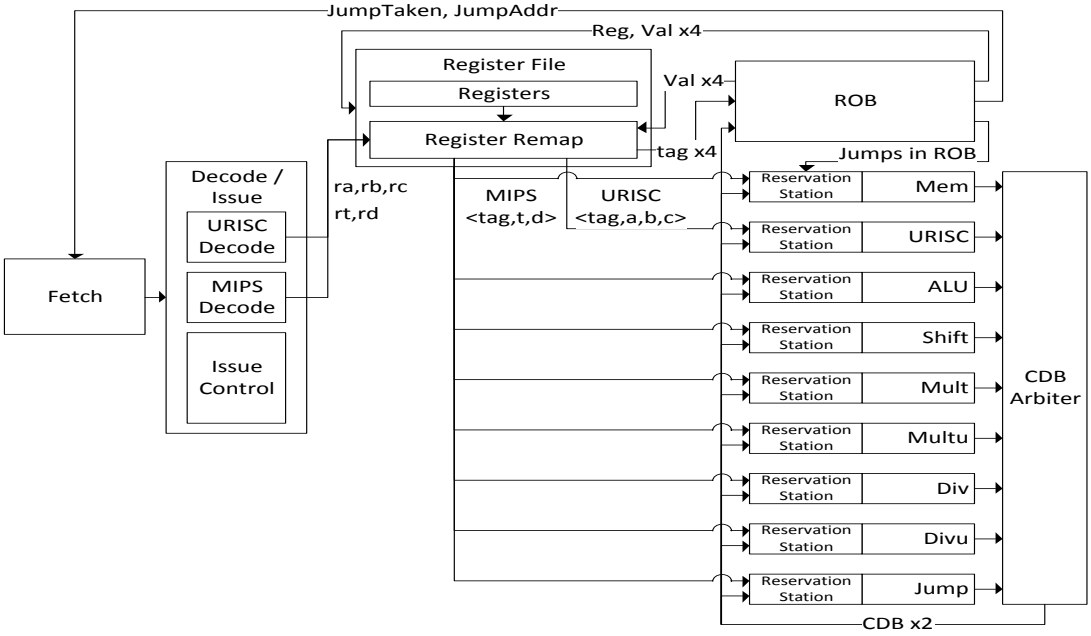
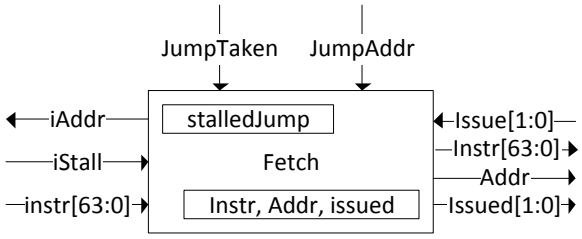


Figure 5.2: Fetch



1. A core reset resets the instruction fetch address to the reset vector.

2. The fetch has stalled (iStall). In this case the iAddr is retained until the iStall is de-asserted.

3. A jump has been taken. In this case the iAddr changes to the jump address and the fetched instructions are cleared.

    - If a jump is taken while the instruction fetch is stalled, we have a stalled jump where the JumpAddr is captured and applied on the next unstalled cycle.

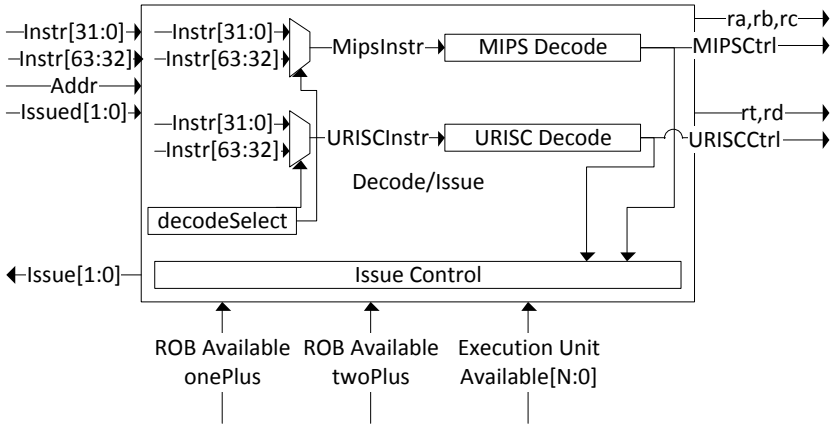
4. The decode unit has yet to consume the last fetched instructions. In this case the previous fetch address is retained.

In the case of an odd address instruction jump, the even instruction is marked as already issued so that it is not decoded and not issued by the decode/issue unit twice.

5.2 Decode/Issue Unit

The Decode/Issue unit is shown in Figure 5.3.

Figure 5.3: Decode/Issue



It consists of decode circuitry for a single MIPS and a single URISC instruction and the logic to issue these instructions to both the ROB and execution unit queues. A small

common decode circuit determines if the (up to) two instructions presented from the fetch unit are MIPS or URISC instructions. The Decode/Issue unit can issue a maximum of one MIPS and one URISC instruction in parallel. When two MIPS or two URISC instructions are fetched in the same 64 bit fetch, the two instructions are issued over two cycles. This avoids the need to duplicate the MIPS or URISC decode circuitry, while still allowing two instructions to be issued if they are not the same type.

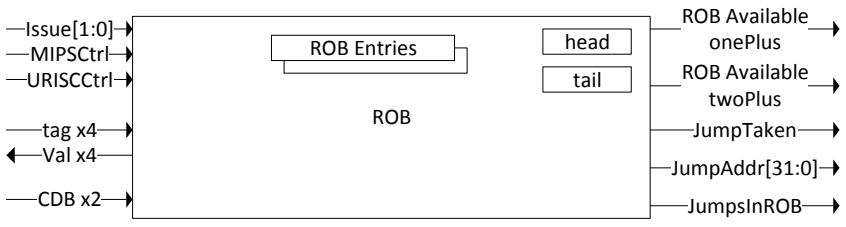
The Decode/Issue unit also performs register remapping with reference to the register file remap registers. Each architectural register is either replaced with the value of the register or, if the value is not computed yet, the architectural register is replaced with the ROB location that will contain the value when it becomes available. The issue unit resolves register remapping when the second of the two fetched instructions relies on the results of the first instruction in the fetched instruction pair.

The issue unit ensures that there are entries available in the ROB and that the execution units are available before issuing an instruction. Instructions are issued in order, thus if the first instruction's execution unit is not ready, the issue unit will wait to issue both the first and second instruction until the first instruction's execution unit is ready.

5.2.1 Reorder Buffer

The Reorder Buffer used in this core is a 16 entry circular ROB. It is shown in Figure 5.4.

Figure 5.4: Reorder Buffer



Most MIPS instructions have a single target register, however a few have two destination registers, specifically:

- divide instructions store to both the HI and LO registers
- MULT and MULTU instructions store to both the HI and LO registers

In addition, the branch and link instructions have two output values - the jump destination address (if taken), and the link register.

Similarly, the subleq instruction also has two output values - the jump destination (if taken), and the result register (rb).

Since a sufficient number of instructions need to return two values, two destination register and value fields are included in each ROB entry. A side effect of this is that moves to and from HI and LO registers now act similarly to a MIPS IV specification - no longer requiring they be separated by a certain number of instructions to prevent UNPREDICTABLE behaviour. In the MIPS specifications I through III, UNPREDICTABLE behaviour would result if these instructions were not separated by at least two instructions.

Each ROB entry consists of:

- Two destination register fields
- Two destination value fields
- A valid flag, set if the ROB entry is associated with a valid in flight instruction
- A complete flag, set to indicate the execution unit has written the value fields
- A speculated jump not taken flag, set to indicate the current instruction is a jump and we have speculated that it was not taken

There are two ROB pointers - head and tail. Instructions are enqueued at the head and retired at the tail.

Retire

The ROB monitors the tail entries to determine if they can be retired. It has the capability to retire up to two instructions per clock cycle: the tail entry and the tail -1 entry of the ROB.

If the tail entry is not a taken jump/branch instruction, it is retired if it is valid and complete. If it is a taken jump/branch instruction it is retired if both it and any delay slot instruction (the tail -1 instruction) are both valid and complete. This restriction ensures that branch delay slot instruction will be executed.

The same criteria are applied for the tail-1 ROB entry.

When a ROB entry is retired, the destination registers in the register file are updated to the values in the ROB. If a jump was mis-predicted (taken), the ROB (less the branch delay slot), execution unit queues, and fetched instructions are reset. The correct jump address is then passed to the fetch unit and execution resumes at the correct address. The jumpTaken signal is used to reset the execution units.

5.3 Execution Units

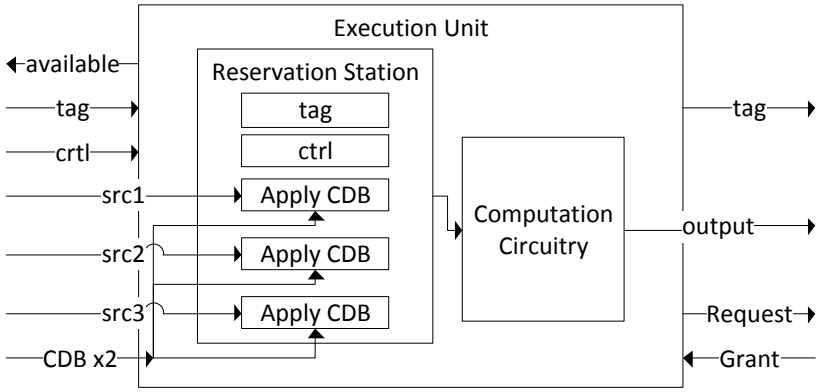
The execution units from both the MIPS and URISC cores are kept separate and not duplicated, however a one entry queue is added at the input of each execution unit for the Issue stage to issue into. It is shown in Figure 5.5.

Each execution unit indicates whether it is available to accept an instruction. This signal is used by the decode/issue unit to determine if it can isue an instruction to the execution unit. This logic take into account cases where the execution unit is not granted the CDB when a result is ready.

The execution units in the core are as follows (as shown in 5.1):

- Memory Access
- ALU

Figure 5.5: Execution Units



- Divide (DIV, DIVU)
- Multiply (MULT, MULTU, MUL)
- URISC (subleq, MFU, MTU)
- Branch

5.3.1 Memory Access Unit

The Memory Access unit takes a bit of additional care. Loads may execute at any time and may be speculatively executed. Stores, on the other hand can not be started until all speculative branches before the store are no longer speculative. The handle this, we issue stores into the single entry Memory Access instruction queue, but monitor the ROB entries from the ROB tail to the store instruction. Only once all these instructions are either retired or not branches, will the store instruction continue through the Memory Access execution unit.

Simply allowing memory access stalls to affect the Memory Access unit (and not the rest of the core immediately) provides a significant speedup. This is primarily because

instructions continue to enter the ROB and get speculatively executed, especially past branches. Any loads past the branches would then get speculatively loaded and are then ready as soon as they are needed.

5.4 Common Data Buses

The core has two Common Data Buses used for transferring the results of the execution units to the ROB (and to the input queues of other execution units), thus two values can be transferred on any one cycle.

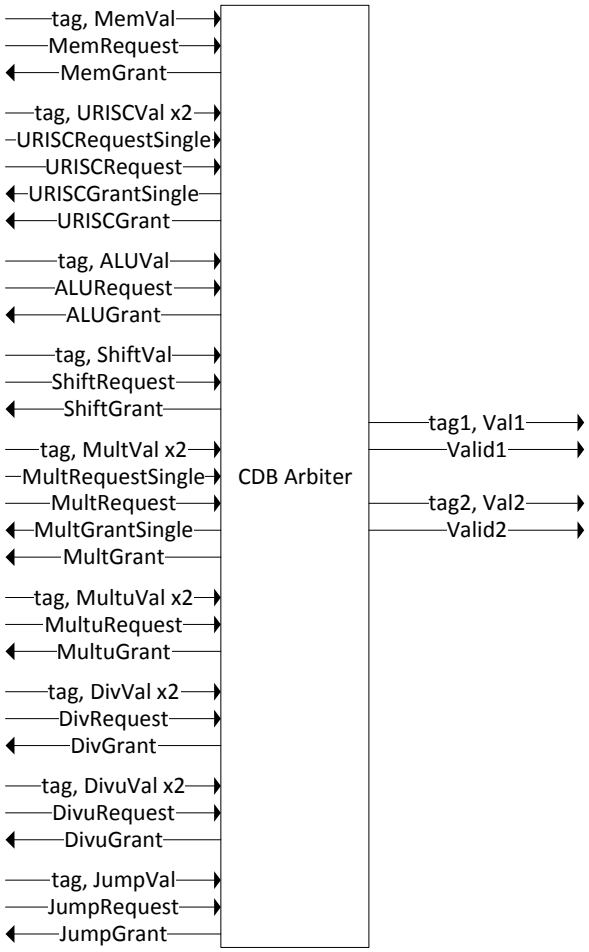
We note that each ROB entry has provision for two values and an instruction that writes to two registers will need to send two values to the ROB. This means that such an instruction will need both CDBs or to split the two values over two cycles to transfer the two values to the ROB. To simplify ROB handling we require the following of the CDB arbitration:

- An Execution Unit that needs to write two values to the ROB shall be granted both CDBs or none of them. This ensures that the results are written atomically.

One might be concerned that using only two CDBs on a core intended to execute two instructions in parallel would present a bottleneck, as each instruction could theoretically need to send two values to the ROB. In this worst case only a single instruction would occupy all CDBs are we would only be able to process one instruction at a time. An obvious solution to this concern is to use more CDBs, however CDBs are expensive since they need to go to every execution unit.

The solution taken in this core is to observe the only instructions with two outputs are divides, multiplies, jumps/branches(with link), and subleqs. All other instructions only have out result register. The divides and multiplies can't be optimized because they truly need two outputs, however the jumps/branches/subleqs have already been speculated on, and we only need to send information back when the speculation is incorrect. Thus we can initialize the 2nd value (next instruction address) in the ROB when the instruction is issued. The CDB will only have a value for this value if the speculated value is incorrect.

Figure 5.6: Common Data Buses



In the case of a subleq, normally the branch will not be taken (or will be taken to the next instruction in sequence - not really a branch). In these cases, the next instruction will be pre-populated in the ROB, and when the subleq instruction completes, the subleq execution unit will only place the rb value on the CDB. If the branch is taken, both the rb value and jump destination will be placed on the CDBs. (And since the CDB arbitration ensures we will get both CDBs if needed, the ROB can mark the instruction as complete

if it receives one or both values in a cycle.)

CDB requesters are given priority as follows:

- Memory access. The memory access unit is given first priority as memory accesses typically end up being the bottleneck and to simplify the memory request pipeline logic. This priority allocation allows the memory access unit to operate at full speed without worrying about being stalled by the CDB.
- Execution units requesting both CDBs. To ensure a unit that needs to transfer two values simultaneously can do so, it is given priority over units that only need one CDB (memory accesses excepted).
- Execution units requesting a single CDB.

Chapter 6

Compiler

The compiler used with the new core is a modified version of the TigerMIPS-URISC compiler used in [2]. The basic URISC compiler adds two passes to the compiler chain:

- A MIPS substitution pass which replaces defective MIPS instructions with their URISC equivalent instruction sequences based on a user supplied list of faulted instructions.
- A pre-assembler pass that runs before the MIPS assembler and assembles the URISC instructions into encoded words. These assembled words then pass unmodified through the MIPS assembler.

These passes effectively utilize the URISC hardware on the TigerMIPS-URISC core, however an additional interleave pass is added for the proposed parallel core.

6.1 Interleave

To take full advantage of the parallel execution capabilities of the proposed core we require the MIPS and URISC instructions to be interleaved so that a double fetch retrieves both a

URISC and a MIPS where possible. As described in Chapter 5, when both a URISC and a MIPS instruction are fetched they can be decoded and issued simultaneously.

To achieve this interleave, a compiler pass is added after URISC instructions are substituted for defective MIPS instructions. This new pass looks for two USRIC instructions in a row and then searches for a nearby MIPS instruction that can be moved between the two URISC instructions. Register dependencies are checked to ensure the moved instruction can be safely relocated without creating a data hazard (RAW, WAW, or WAR). Control flow is also examined to prevent instructions being moved across control flow instructions. Load and store instruction ordering is maintained.

Chapter 7

Experimental Results

To measure the effectiveness of the new core a number of simulations were run with and without faulted instructions.

7.1 Benchmarks

The benchmarks were derived from Mibench[20] benchmark suite, and are essentially the same ones used in previous URISC work. There are 4 benchmarks used:

- Dijkstra: A benchmark implementing Dijkstra's algorithm representing graph traversal
- Bubble Sort: An implementation of the Bubble Sort algorithm
- RSA: A benchmark representing cryptographic performance
- String Search: A benchmark for evaluating string (byte) searching performance

The benchmarks were revised to operate entirely out of memory on a bare-metal environment. The benchmark runs are repeatable and run for the same number of cycles each time.

7.2 Methodology

To compare the two cores we recognize that the two cores use significantly different structures.

- The TigerMIPS-URISC core is a simple 5 stage in-order core with no speculation. We will refer to this core as the vanilla core, as it contains no modifications from the original work.
- The proposed out-of-order (ooo) core makes heavy use of speculation and is able to execute instructions out of order.

To establish a baseline, we compile the benchmarks detailed in section 7.1 with the same compiler (and flags) with no faulted instructions. We run these benchmarks on the two cores to compare their performance.

With this as a baseline, we then recompile the benchmarks but this time tell the compiler that various instructions are faulted and therefore need to be replaced with their equivalent URISC instructions. The benchmarks with the faulted instructions replaced with URISC instructions are then run again so the performance can be compared.

7.3 Results: Faulted Addu

The first results shows the performance of the cores with the Addu instruction faulted, and therefore replaced with equivalent URISC instructions. The results are shown in Table 7.1 and Figure 7.1. Results are normalized to the performance of the TigerMIPS-URISC (vanilla) core.

We observe that the new ooo core outperforms the vanilla core even in the un-faulted case with speedups between 1.08 and 2.22 times the performance, with a geometric average speedup of 1.49. As the ooo core is still not able to issue more than one MIPS instruction per cycle we note that the majority of the speedup is due to the ability of the ooo core to speculate past branches and execute multiple instructions per cycle. Depending on the

benchmark, we also observe noticeable gains with the memory access unit and fetch units no longer stalling the entire core on a memory stall. This allows more effective use of the memory interfaces and increases performance.

Once the faulted instruction is introduced we note a significant performance penalty in the vanilla core, with the dijkstra, rsa, and stringsearch benchmarks loosing approximately half their performance. The ooo core, however, is able to retain much more of its performance despite the fault and is even able to beat the performance of an un-faulted vanilla core most of the time. When compared with the faulted vanilla core speedups range between 1.28 and 2.77 times, with a geometric average speedup of 1.93.

Figure 7.1: Benchmark Performance With Faults: addu

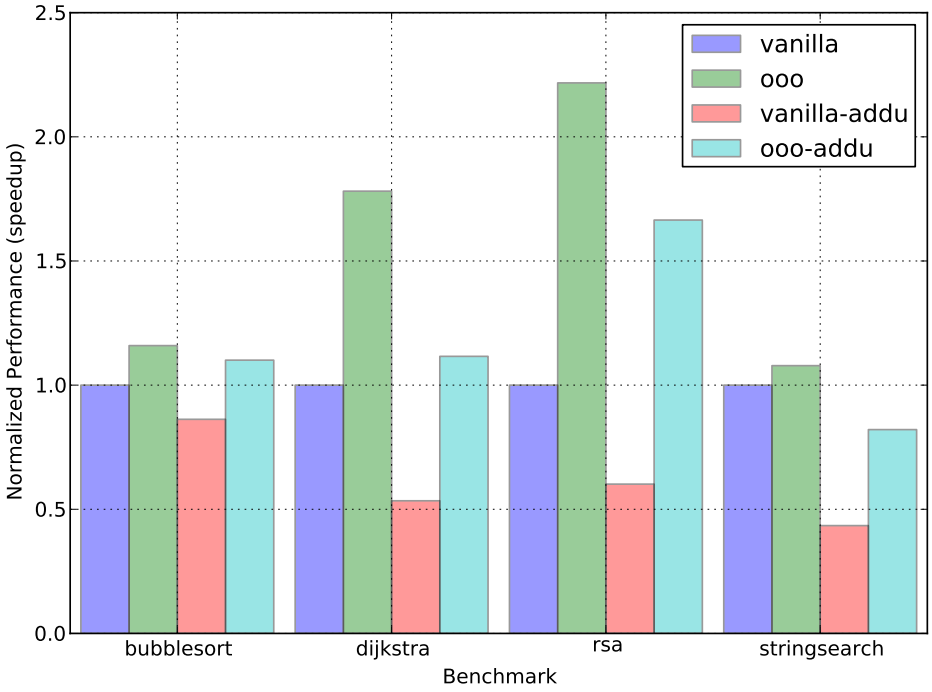


Table 7.1: Performance Results, Faulted: addu

Benchmark	Speedup (higher is better)				Speedup Faulted
	vanilla	ooo	vanilla-addu	ooo-addu	(ooo vs vanilla)
bubblesort	1.00	1.16	0.86	1.10	1.28
dijkstra	1.00	1.78	0.53	1.12	2.09
rsa	1.00	2.22	0.60	1.66	2.77
stringsearch	1.00	1.08	0.43	0.82	1.89

7.4 Results: Faulted Subu

Results for faulted Subu instruction are shown in Table 7.2 and Figure 7.2. The ooo core has speedups of 1.16 to 4.22 times when compared with the faulted vanilla core, a geometric average speedup of 1.82.

Figure 7.2: Benchmark Performance With Faults: subu

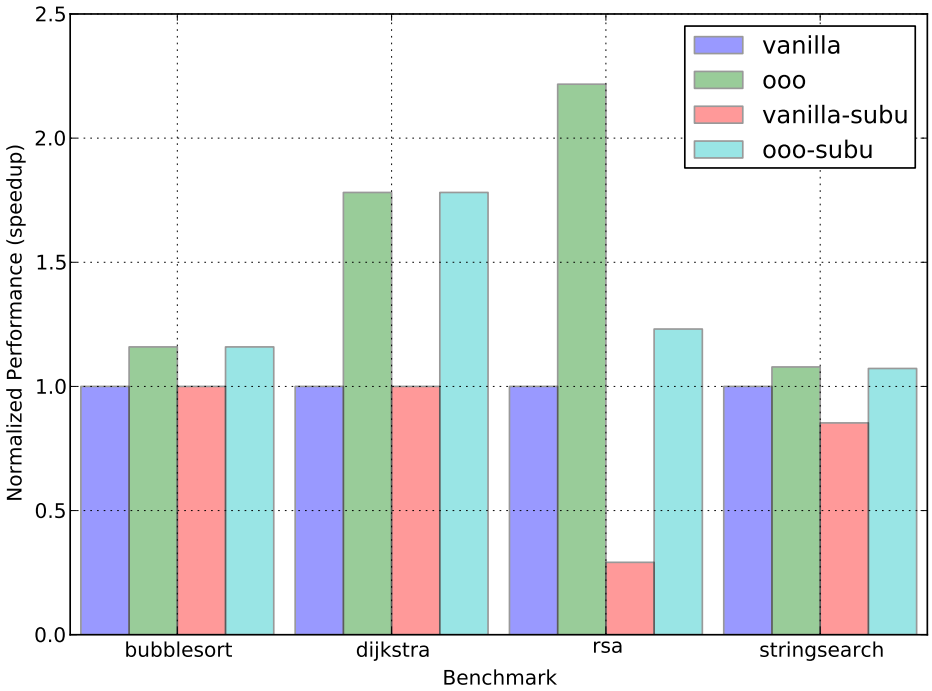


Table 7.2: Performance Results, Faulted: subu

Benchmark	Speedup (higher is better)				Speedup Faulted
	vanilla	ooo	vanilla-subu	ooo-subu	(ooo vs vanilla)
bubblesort	1.00	1.16	1.00	1.16	1.16
dijkstra	1.00	1.78	1.00	1.78	1.78
rsa	1.00	2.22	0.29	1.23	4.22
stringsearch	1.00	1.08	0.85	1.07	1.26

7.5 Results: Faulted Addu and Subu

Results with both Addu and Subu instruction faulted are shown in Table 7.3 and Figure 7.3. The ooo core has speedups of 1.28 to 3.22 times when compared with the faulted vanilla core, a geometric average speedup of 1.94.

Figure 7.3: Benchmark Performance With Faults: addu-subu

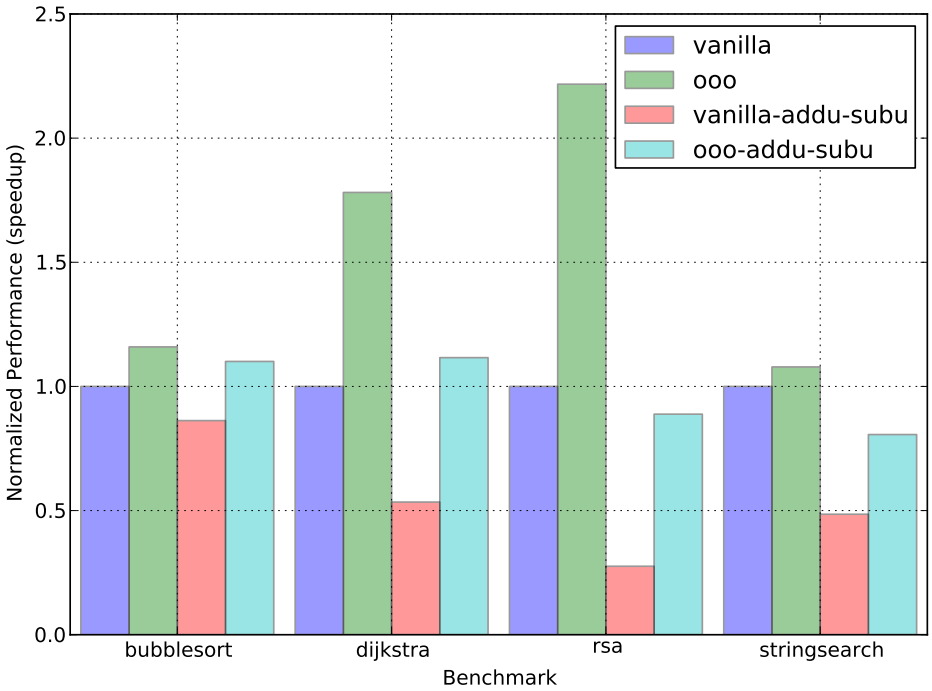


Table 7.3: Performance Results, Faulted: addu-subu

Benchmark	Speedup (higher is better)				Speedup Faulted (ooo vs vanilla)
	vanilla	ooo	vanilla-addu-subu	ooo-addu-subu	
bubblesort	1.00	1.16	0.86	1.10	1.28
dijkstra	1.00	1.78	0.53	1.12	2.09
rsa	1.00	2.22	0.28	0.89	3.22
stringsearch	1.00	1.08	0.49	0.81	1.66

Chapter 8

Conclusion

This thesis analysed the TigerMIPS-URISC core to identify ways to accelerate computation. Opportunities to execute instructions in parallel were identified, primarily restricted by data hazard handling and control constraints around the subleq branch.

8.1 Contributions

A core that solves the challenges identified is proposed. It accelerates programs with URISC and MIPS instructions used for fault tolerant applications. Various benchmarks were run on both the original core and the new core for comparison purposes. The results show speedups of between 1.16 and 4.22 times, with a geometric average speedup of 1.90.

8.2 Future Work

This work compares the execution of the same subleq check code on both the urisc and parallel urisc processor.

An area of future work is the investigation of runtime selection of instructions to check. The current work relies on a compiler to select the set of instructions to check at compiler

time. It also requires the compiler to insert the associated check code. These selected instructions are then always checked, while those instructions not selected are never checked. It should be possible to provide the core with a region of memory containing the check instruction code and add intelligence into the core so that it chooses which instructions to check. This would relieve the compiler of the task of selecting which instructions to check and allow any program compiled with a standard MIPS compiler to be checked without binary modification. It would also allow the types of instructions to be checked by the core to vary over time so as to achieve higher coverage rates.

References

[1] Farhad Mavaddat and Behrooz Parhami. Urisc: The ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25:327–334, 1988.

[2] D. Wang, A. Rajendiran, S. Ananthanarayanan, H. Patel, M.V. Tripunitara, and S. Garg. Reliable computing with ultra-reduced instruction set coprocessors. *Micro, IEEE*, 34(6):86–94, Nov 2014.

[3] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, Dec 2003.

[4] S. Das, G.S. Dasika, K. Shivashankar, and D. Bull. A 1 ghz hardware loop-accelerator with razor-based dynamic adaptation for energy-efficient operation. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 61(8):2290–2298, Aug 2014.

[5] A. Meixner and D.J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 80–89, June 2008.

[6] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[7] Amin Ansari, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Necromancer: Enhancing system throughput by animating dead cores. *SIGARCH Comput. Archit. News*, 38(3):473–484, June 2010.

[8] P. Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 481–488, Oct 2003.

[9] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 520–531, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. *SIGPLAN Not.*, 41(11):73–82, October 2006.

[11] T.M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[12] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 411–420, July 2001.

[13] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 210–222, Dec 2007.

[14] Aravindkumar Rajendiran, Sundaram Ananthanarayanan, Hiren D. Patel, Mahesh V. Tripunitara, and Siddharth Garg. Reliable computing with ultra-reduced instruction set co-processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 697–702, New York, NY, USA, 2012. ACM.

[15] Sundaram Ananthanarayan, Siddharth Garg, and Hiren D. Patel. Low cost permanent fault detection using ultra-reduced instruction set co-processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 933–938, San Jose, CA, USA, 2013. EDA Consortium.

[16] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *IEEE Transactions on Nuclear Science*, 43(6):2874–2878, Dec 1996.

[17] L. Hsiao-Heng Kelin, L. Klas, B. Mounaim, R. Prasanthi, I. R. Linscott, U. S. Inan, and M. Subhasish. Leap: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pages 203–212, May 2010.

[18] K. Lilja, M. Bounasser, S. J. Wen, R. Wong, J. Holst, N. Gaspard, S. Jagannathan, D. Loveless, and B. Bhuva. Single-event performance and layout optimization of flip-flops in a 28-nm bulk technology. *IEEE Transactions on Nuclear Science*, 60(4):2782–2788, Aug 2013.

[19] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[20] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.