A Private, Associative Memory Alternative for RISC Systems

by
Madeline Burbage

Professor Duane Bailey, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 23, 2022

Abstract

The concealed inner workings of today's memory hierarchies lead to surprising side effects, in security and performance, for otherwise simple programs on general–purpose machines. For situations where this is untenable, developers may prefer access to an alternative memory system where software has control over the movement of its data. To explore this concept, we designed and implemented the Scratchpad Accelerator, a configurable, associative scratchpad memory for the RISC–V Rocket Chip. We evaluated the security and performance of the device after adding it to lowRISC, a version of the Rocket Chip that has been programmed to an FPGA board and runs full–stack Linux. We show that the Scratchpad Accelerator provides data isolation and uniform memory access, providing a protected, predictable memory system for developers. Additionally, its associativity and structure inspire effective memory models for common data structures in software development. The general–purpose memory acceleration provided by the Scratchpad Accelerator endows software with the important ability to gain freedom and control over its data.

Acknowledgements

I'd like to start with a huge round of thanks to my advisor, Duane Bailey. Beyond years of mentorship and fantastic research guidance, his love of interesting puzzles and finding their creative solutions has opened up a whole new world of computers for me. This thesis would be much poorer without his excellent advice. My CS classes here have helped me delve into the magic behind the computer screen, and I'd like to thank the Computer Science Department for their engaging instruction. Finally, I would like to thank my friends and family, who kept me going during the hardest and best nights of this year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Caches are crucial to mitigating one of the biggest barriers to computer performance, the processor–memory gap. They have situational downsides, however: some programs access memory in ways that violate cache assumptions, so the cache cannot efficiently decrease memory throughput for those programs. Much worse, for some contexts, are the security risks posed by caches. Sensitive data used by supposedly–secure programs can leave traces in the cache that malicious users may seek out. This information leakage can happen in very unexpected places, because modern processors rely on *speculative execution* [15]. Additionally, caches take up space and have overhead in the time needed for maintaining their coherence. But, of course, we still use them for their very tangible benefits [11].

To address these concerns, rather than replacing caches with a new memory hierarchy, this work envisions an additional memory tool—a *scratchpad*—freely available to processes through specialized instructions, which could better serve the applications that caches neglect. We believe the addition of a scratchpad memory will help with performance and security for certain programs, while not hurting the performance of the rest of the programs that a cache serves well.

1.1 Motivation

Cache–inefficiency is a longstanding problem, and there have been many software–based approaches to mitigating it, through better allocation of memory, changing the positioning of hard–to–cache data structures, and changing the design of algorithms that access data irregularly. This is to address the twin assumptions of *spatial* and *temporal locality* that caches make, and not every program follows. The work of Chilimbi et al. demonstrates ways to color data by how frequently it will be used and to cluster structures like trees into logically–accessed sections [6]. Other solutions involve profiling a program's memory accesses and automatically changing its allocation patterns from there [20]. These methods all can allow data structures to work better with cache assumptions, potentially having a major impact on program performance. However, some solutions increase the burden of memory management during software development, and other methods are not able to guarantee increased performance, even slowing down memory accesses in some cases [8]. Approaches also come

at the expense of software overhead in compilation or during runtime. Rather than curating memory accesses to work for the cache, some have explored other memory additions for improving certain classes of algorithms.

Scratchpads, small blocks of memory close to the processor without replication in other memory levels, are commonly used for embedded applications where memory needs are known in advance. They have also been suggested as a means to shrink the processor–memory gap, because they have been designed with different motivations than caches [10]. Features like their lack of a hierarchy—data in a scratchpad only exists once, in the scratchpad—mean they have no redundancy. This eliminates the need for coherence work and its associated overhead. Finding new ways to integrate memory and computation on the same chip may be the only way to surmount, rather than simply mitigate, the gap caused by slow memory accesses [18]. The SPX64 is one such scratchpad implementation targeting Intel processors, designed to increase program performance [21]. This implementation also takes advantage of the isolated nature of the scratchpad to offer a more secure platform for certain sensitive calculations.

1.2 Contributions

With the onset of reconfigurable computing and an accelerator–led model of processor development, we see a pathway for scratchpads to aid in the performance and security of certain programs. This research centers around the design and testing of a scratchpad accelerator for Berkeley's *Rocket Chip*. This processor is an open–source implementation of the open–source *RISC–V ISA*, and it works well on reconfigurable *FPGA* devices [1, 24]. On a reconfigurable platform, the Rocket Chip can easily be updated over time through integration with different modular extensions like coprocessors and hardware accelerators. In this work, we design and implement a scratchpad–based memory system and its custom instruction set through the Rocket Chip–targeted *RoCC accelerator* model. By implementing our scratchpad design in this way, we can measure the effectiveness and utility of our approach to a memory hierarchy alternative.

Our accelerator design allows developers to push memory performance to greater heights. The parallelism between our scratchpad and the *L1 cache* provides developers with fine–grained control when they need it, and the ease of cache–controlled access otherwise. Our scratchpad is a powerful tool for developers because we prioritize developer flexibility and reach through the design of our accelerator and its instructions. It fuels software's freedom and control over its memory usage, while keeping accesses consistent, fast, and protected. The layout of the scratchpad reinforces these principles, with data broken into regions we call `stripes` that developers reserve during accelerator use. These regions enable data isolation and protection, but further allow developers to segment memory to be used for multiple memory models at once. Thinking about storage differently between regions helps the accelerator efficiently power a variety of applications, like optimizing access for important local variables, accelerating hash table lookups, or isolating critical cryptography secrets. Basic access instructions let software interface the Scratchpad Accelerator as it would communicate with the regular memory system. However, we enable development flexibility by configuring extensions to these instructions to allow further memory optimizations. Additionally, we reduce the

burden of accelerating code through the scratchpad by constructing a C library of scratchpad access functions at a larger scale and complexity. Between the layout and interface of the Scratchpad Accelerator, we give developers the power to store data in ways benefiting specialized needs, providing a security and efficiency hidden by the cache. This work envisions an adaptable tool for developers to use when caches alone do not suit their needs.

1.3 What Lies Ahead

Microarchitecture development has drastically improved program performance by many measures, but some programs are more suited than others towards current general–purpose computer designs. After establishing the feasibility of scratchpad–based parallel memory systems, we attempt to determine the usefulness of our Scratchpad Accelerator across a range of potential applications. We will transition from the theory into empirical results as follows. In Chapter 2, we highlight related literature: prominent problems facing modern memory hierarchies, alternative hierarchies, and scratchpad approaches to fixing selected issues. Chapter 3 explains the environment that our research takes place in, especially concerning the hardware design of Berkeley's Rocket Chip. We approach the design of our accelerator, through its hardware and instructions, in Chapter 4. Chapter 5 considers various applications and the approach towards implementing them with the Scratchpad Accelerator. In Chapter 6, we examine the performance and utility of the accelerator through experimentation. Finally, Chapter 7 draws conclusions and suggests some areas ripe for further investigation.

Chapter 2

Related Work

In preparing to build our scratchpad, we first review unmet needs in today's memory systems. Modern memory hierarchies rely on caching to reduce memory access times to tolerable levels, so we additionally delve into CPU caches' effects on system performance. Two particular application areas stand out here: security in the memory hierarchy and effective storage of common software constructs. With a thorough understanding of the problems, we consider potential solutions, investigating alternative general–purpose–computer memory hierarchies and the motivations for their appearance. Finally, we highlight research on device acceleration, along with the RISC–V acceleration platform and toolchain that we will be using for our work.

2.1 The Processor–Memory Bottleneck

Improving memory access latency is an active area of research because any delay limits processor performance. These articles document known memory issues and propose inventive solutions.

For decades, the gap between memory and processor speeds has grown exponentially, causing processor performance to become ever more reliant on improvements to memory speed; observe the trends of Figure 2.1. Mahapatra and Venkatrao illustrate the reasons for this growing bottleneck, which unfortunately are deeply rooted in the design priorities and production processes of the specialized memory (DRAM) and microprocessor semiconductor industries [18]. In order to achieve better memory bandwidth, memory latency may need to be aggressively improved or simply tolerated better. This article surveys proposed solutions in the forms of improved hardware materials and innovative microarchitectural designs. In particular, the authors critique various common cache models. Caches are powerful through their harnessing of the redundancy of memory accesses due to temporal and spatial locality. Yet the article views this redundancy as a sign that memory accesses should be redesigned to be more efficient in the first place. Beyond techniques that enable caches to perform better for the contexts they traditionally struggle with, Mahapatra and Venkatrao cite compression techniques that can improve bandwidth at various stages of the memory hierarchy. Underlying many of these microarchitectural solutions is a shift of complexity and control in handling memory accesses. Compression might involve new intermediary units between processor and

memory or happen in the software issuing requests in the first place. While caches remain dominant forces behind today's memory hierarchies, the opportunity to selectively shift this complexity, only in applications that can make good use of the extra control, is a motivating reason to build our scratchpad unit.

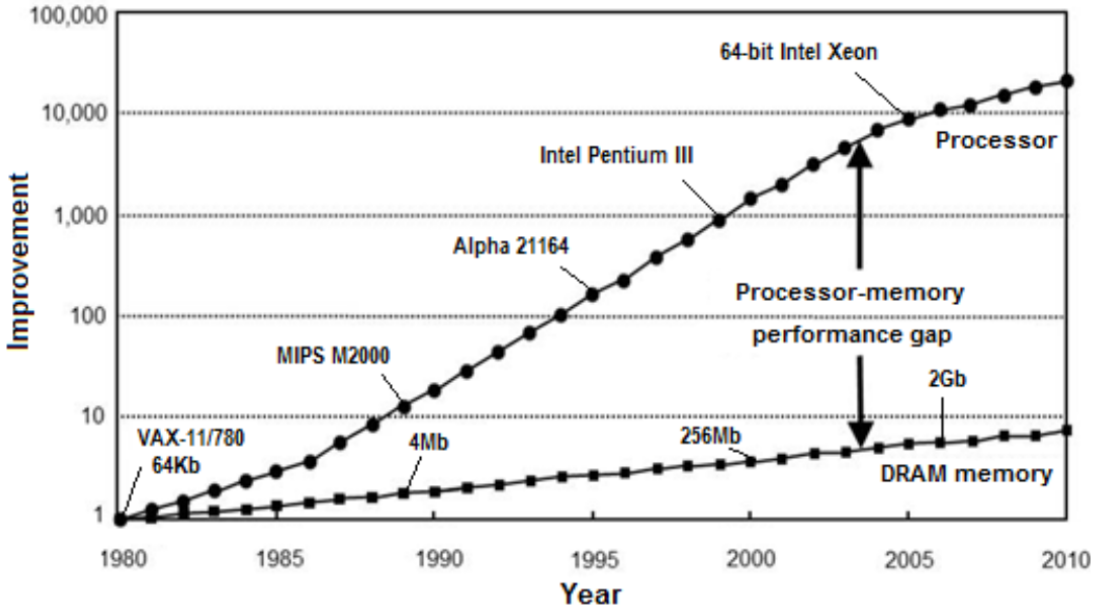

Figure 2.1: The Processor–Memory Gap over time. Figure from Efnusheva, Cholakoska, and Tentov [10].

Efnusheva, Cholakoska, and Tentov review recent approaches for reducing or tolerating memory latency and increasing memory bandwidth. Here, approaches to increasing memory speed are divided into those that reduce latency and those that merely tolerate it better. One of the promising solutions they highlight are memory–centric architectures, which prioritize the placement of memory regions nearby processing units. This can be done in a variety of ways, all to improve on the tricky relationship between computation and storage. Merely through the physical proximity of processing and storage, memory latency can fall. Additionally, reredesigning the relationship between the processor and memory can reduce the complexity of accesses, enabling greater memory bandwidth. Many of these systems have not gained widespread popularity for general–purpose computing because of their drawbacks: either limited memory capacity or additional software complexity for their use. Because of this, certain systems like *scratchpads* or integrated processor–in–memory (or memory–in–processor) chips are seen as best applicable to embedded computing environments. There, software needs are often known and controlled ahead–of–time, unlike in general–purpose computing. Nevertheless, the narrowed domain of an embedded system aligns with the narrowed domain of hardware accelerators for general–purpose computers. Accelerators also are provided with software needs ahead of time and only apply to specific computational tasks fitting their constraints. Therefore, these methods of mixing computation and memory could fit well with accelerators, which

also make a software complexity trade–off in turn for higher performance.

Of course, caches have long proved to be the most popular structural method for tolerating memory latency. Liptay's description of one of the earliest CPU caches, in 1968, is remarkably similar to cache models we use today [16]. Although the hierarchy only included a single cache level, with a now–low hit rate of 96.8%, the investigation into total cache size, block size, associativity, and replacement policies captures the building blocks of modern caches. Liptay accurately foresaw this innovation, finally made possible through refinements in cache technology, as a groundbreaking improvement to processors. Since then, as replacement policies have evolved and memory hierarchies have grown and changed, caches remain one of the best memory latency–hiding techniques in our computers [11]. Yet, over the ensuing decades, the state of the processor–memory gap has drastically changed, for better, then worse. This raises the question of whether this method of latency tolerance is enough, or if other trade–offs and innovations must be made to continue improving our computers.

2.2 Cache Conflicts

Harnessing the intensely tangible benefits of caches is sometimes easier said than done, since caches are designed to be transparent to software users. It is likewise hard for software designers to avoid cache drawbacks. This may enable supposedly–secure data to leak across protection domains or cause program data to inefficiently traverse the memory hierarchy, slowing the system down. Thus considerable research has been done on documenting these flaws, and harnessing caches better through clever software redesign.

This focus on caches during the software design stage is often labeled cache–conscious programming, which tries to organize data and its access patterns so they cooperate well with the structure and replacement policies of the cache. As Chilimbi et al. note, this paradigm has been easier to implement for programs using array–like data structures [6]. These structures have uniform random access to their elements, on top of having dependencies that can be statically analyzed. Both provide avenues to reorganizing storage between what programmers envision and caches prefer. Pointer–based structures are very popular programming tools as well, but their lopsided access patterns make early cache–conscious techniques for reordering data accesses ineffective. Instead, Chilimbi et al. suggest increasing locality between these structures' elements by reorganizing the structures in memory. They provide specialized C *memory allocators* and data structure *reorganizers* for programmers to use. These are capable of clustering related elements to the same cache block, or coloring related elements to non–conflicting cache regions. Both techniques reduce the probability that a cache will evict soon–to–be–used data. By increasing software complexity and resource usage, these methods can significantly speed up programs relying on pointer–based data structures. One constraint is that this speedup depends on how well the data structure and cache can be modeled and aligned by the allocators. This motivates the development of further memory–access strategies for data structures that cannot be modeled well.

Instead of increasing programming complexity in order to cater to the cache, Savage and Jones propose an automatic system for creating cache–conscious software. Their system, HALO, cleverly allocates memory for a program's data structures based on *profiling* [20]. After compilation, a

program gets profiled by HALO in order to identify temporally–related data. Then, the most–related data is grouped and labeled in the program's binary. During actual program runtime, the program calls a custom allocator that HALO has produced for this program. The allocator can cluster data by its group for efficient cache placement. HALO produces up to 28% runtime speedup over a common allocator (`jemalloc`) on a variety of heap–accessing benchmarks, with a variety of data structures. It produces no runtime speedup in the worst case, where careful placement of small data objects does not actually affect program performance. So, at the cost of significant compilation slowdown and increased heap fragmentation, this technique performs fast runtime cache–conscious data placement that often benefits a wide range of data structures.

2.3 Data Security

As research into cache–conscious programming demonstrates, programmers can be surprised by the effects of caches on code performance, since caches are designed to be transparent, hidden from a programmer's gaze. Beyond having confusing effects, this makes it hard for software design to pivot to actually fixing cache–caused problems in their code. Another architectural mechanism hidden from the programmer's gaze, *speculative execution*, causes similarly problematic effects. When combined with the hidden peculiarities of the cache, undesirable and insecure behavior can plague code that programmers believe to be safe.

Speculative execution enables processors to run instructions from code before it is known if they will be used. This is like trying the code within an if–statement before the processor has determined which branch of the statement to take. The predicted instructions do not commit their results to the memory system until the processor determines that they are actually the correct instructions to run. If they are not correct, the processor has to discard these instructions and run a different set instead [11]. This has the potential to vastly speed up code execution time, and indeed has driven a lot of processor performance gains in the last decade. Unfortunately, attacks like *Spectre* have proven that speculative execution is also a huge vulnerability in modern architectures. With Spectre, Kocher et al. demonstrate that attackers can uncover confidential data by speculatively executing code that leaks data into the cache [15]. Even though speculatively–accessed instructions should not write changes to memory state, these instructions do change microarchitecture state as they are executed. For instance, memory they access is loaded through the cache hierarchy, so an access may be faster or slower depending on where the data is, and the data may move to higher levels of the cache as it is retrieved. A clever attacker can determine secret data if it is involved in these speculative accesses. The attacker can detect which lines have been loaded into the cache, perhaps by trying to access a set of memory locations and seeing which ones are loaded the fastest (since they were already loaded by the victim). This is a type of cache–based *side–channel attack*, where an attacker can retrieve secret information from a victim, just because the victim's program must use the cache. Since the same cache hardware is shared between different security contexts, whether different processes or sandboxed agents in one process, then if other hardware constructs like speculative execution undermine the cache's guaranteed protections, it becomes a convenient side–channel for leaking data. Other microarchitectural side–channels exist, and many have been

used for demonstrations of Spectre. Yet the large bandwidth of data passed through the cache make it a very convenient covert channel for stealing secret data.

The Spectre attacks were additionally threatening because they were demonstrated across a wide variety of programs. The original paper shows JavaScript code that a malicious website could use to attack the browser running it [15]. On the other end of applications, it demonstrates C code that slowly steals information from other processes. Proposed solutions try to balance performance impacts with the security implications caused by Spectre. These revolve around restricting speculative execution or reducing how much information can be leaked through covert channels. On one hand, software can implement methods like *speculative barriers* to protect certain regions of code from speculation through barrier instructions like `fence`. Alternatively, we have hardware–based fixes, either by tracking whether data came from speculative operations and protecting it better until the speculation ends, or by limiting the ease with which data can be leaked across different parts of the processor. Unfortunately, a full fix requires significant processor redesign and a reevaluation of the balance between security and performance.

One architectural solution to Spectre attacks involving the memory hierarchy is InvisiSpec [26]. This solution hides speculative memory accesses from the cache hierarchy by loading their retrieved memory to a speculative buffer where only speculative instructions can operate. Data is exported from the buffer to the memory hierarchy once it is known that the accesses will not be undone. This serves to secure data from cache–based speculative attacks across processes, but it cannot protect data from leaking out of other microarchitectural covert channels. InvisiSpec has two designs: one to protect against speculation after branches, which Spectre made use of, and one to protect against speculation caused by any means. Running speculation benchmarks under InvisiSpec, there is a 7.6% slowdown for protecting against branch–based speculation, and a 18.2% slowdown for protecting against all speculation [27]. Overall, InvisiSpec is able to protect cache data from certain threats with a significant, but not insurmountable, performance hit.

Other memory architectures attempt to selectively increase data security so they do not need to sacrifice as much speed. This is done by pushing control over data security to the software developer. The SPX64, designed by Singh et al., proposes a modification to general–purpose computers that adds scratchpad memory to Intel–based systems [21]. Scratchpad memory is a small, fast memory close to the CPU. Its contents are managed by software instructions, not automated in hardware, so it does not need to have its contents replicated in further memory levels. Therefore, scratchpads benefit from increased speed and a lack of cache–coherence traffic in return for increased software complexity in managing a limited storage. Here, the SPX64 is placed alongside the *L1 cache* as a complementary memory device. It seeks gains in performance for specific workloads while protecting its data from cache–based side–channel attacks [21]. Data within the scratchpad is accessed in fast, uniform time. This makes scratchpads useful for programs with frequent access to a predefined region of data, or programs that do not want accesses to a region of data to visibly affect the shared resource of the cache.

The Intel scratchpad is designed to subtly transfer data to and from the outer memory systems when directed by users using simple instructions [21]. Most of SPX64's design works to safely integrate the scratchpad (SD$) with the instruction pipeline and surrounding memory system in

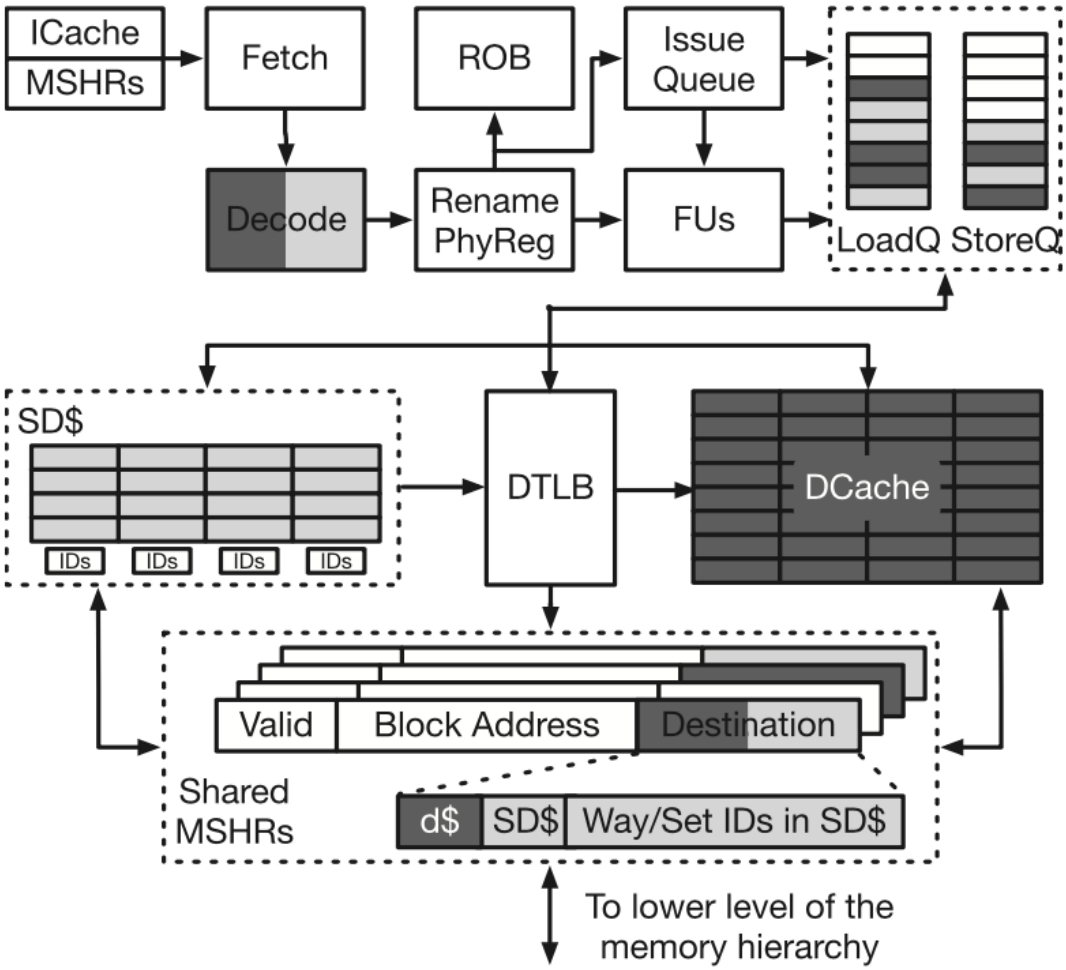

Figure 2.2: High–level architecture of the SPX64. The device connects to speculative and reordered instruction data at top, and larger memory systems at bottom. Figure from Singh et al. [21].

Figure 2.2. Once data is in the scratchpad, it is not extracted except at the user's instruction or during a store to kernelspace on a context switch. Even this transfer may be restricted, as the scratchpad must be configured from the start to allow different context–switch management policies. Although the scratchpad's data has process–level protection, it must be configured to use a non–speculative read instruction to avoid speculative attacks on scratchpad data from within a process. As with any speculative execution mitigation, that non–speculative instruction comes with performance drawbacks. Allowing speculative execution, the scratchpad can accelerate a variety of benchmarks with suitable data structures. It performs much better than a system that simply has an enlargened L1 cache. With the speculation–safe instruction included, the scratchpad performs roughly in line with secure–cache proposals like InvisiSpec on security benchmarks. Here, security is treated as an application parameter much like the scratchpad's storage structure and context switch policies. This makes the scratchpad useful for many contexts, but only if it has been configured for the correct purpose beforehand.

2.4 Programming Constructs

In the right context, scratchpads demonstrably bring performance and security benefits to a range of applications. Finding amenable applications can be tricky, however, when data favored for fast retrieval through the scratchpad needs to also be small enough to fit without excessive work in managing it. The following research examines some common and well–ordered structures that are frequently used to support programming models and thus may be amenable to specialization in scratchpad memory.

Stacks lie at the heart of how our programs run, so efficient stack management is a crucial task for good performance. In MiBench, a common embedded systems benchmarking suite, 65.29% of data accesses targeted the stack [14]. For programs with high stack usage on systems that desire low power usage, one of the most efficient storage devices for the stack might be a scratchpad with an effective stack–management mechanism. Kannan et al. introduce a software–based stack manager for scratchpad memories that decides frame placement during runtime based on simple dependency knowledge and frame sizes calculated during compile time. When the stack outgrows the scratchpad, the manager evicts the oldest frames to slower memory, and attempts to bring them back later, filling the scratchpad in a circular fashion. This design cannot effectively handle every stack possibility, like storing partial frames when a frame's size is larger than the whole scratchpad. However, the possibilities it can handle allow it to produce 13% speedup over a comparable architecture using a small cache instead, along with a 32% reduction in power usage. Knowledge about the layout and access patterns of the stack may benefit the scratchpad stack manager in ways that the cache cannot account for.

The *circular stack* aligned to a small memory device recalls a similar construct in the SPARC architecture. Here, the register file is treated as a large circular stack of *register windows*, where each contains local registers connected to a software stack frame [25]. Software can only access the registers of the current window, and windows overlap for easy sharing of parameters and return values across functions. Software is also responsible for managing these windows, with function calls

spurring the shift of the current window. In tricky cases of overfilling the register file, traps handle cases where old window registers must be evicted to memory or recalled. Unlike a generalized stack manager, this design accounts for static window sizes, and it causes stack frames to grow larger to account for the register window data that they may have to save later. However, this is also a commercially–viable example of managing some call stack data as a circular stack in hardware.

Yet another application domain arises with function calls, in cases of dynamic dispatch. Dynamically–bound functions may actually need to look up the address of the proper method to call at runtime. This happens especially through polymorphic functions, where one call–site could invoke different methods [12]. Clever inline caching mechanisms demonstrated by Hölzle et al. enable these lookups at low cost. Rather than searching through all possible functions, each call–site caches the last few methods called there by their receiver types. Then on a new call, the function type is compared to the cache, and the system only executes a more comprehensive, but expensive, lookup if a matching type is not cached. Using these polymorphic inline caches trades increased code complexity for speed, and an additional benefit is that these methods collect useful type information. However, the linear search across the inline caches for the matching types is a repetitive source of inefficiency, especially if cache order is not rearranged to match the frequency of method calls. An accelerated lookup here would provide significant benefit to programs with many dynamically–bound polymorphic functions.

2.5 Accelerators

We see scratchpads as a high–potential method to safely accelerate memory accesses for certain programs. In a way, the scratchpad will act as a hardware accelerator, but with a larger class of clients than a normal accelerator would entail. It is, arguably, general–purpose. Here we examine prior work in acceleration and how it works on our target platform.

Hardware acceleration in general–purpose computers is finally reaching its stride, with *heterogeneous architectures* undergoing rapid research and commercial success. By combining many specialized hardware units, general–purpose computers can handle their most common applications better. Choosing the most effective applications to accelerate is still an important question, because it is hard to answer accurately at an early stage of the application design process. Yet, when accelerators are chosen and designed correctly, a significant amount of development time can be saved. There are a variety of tools available to help with finding, generating, and testing effective accelerators. Zacharopoulos et al. present a partitioning tool that analyzes a C program as it compiles in the LLVM toolchain and defines effective portions of code for acceleration in hardware [28]. They find this tool produces better–performing accelerators for their system, the Xilinx Zynq Ultrascale+ PSoC, than if they had profiled the code to choose acceleration candidates. Syrowik et al.'s research on evaluating early–stage accelerator designs through profiling confirms that it is hard to do this accurately without extra information from the HLS tool that generated the accelerator [22]. There are also tools available to partition, construct, and integrate accelerators from C programs in several steps, but it is harder to find effective open–source versions. Vogt et al. present a GCC–based approach to HLS on the ZedBoard that can automate the entire accelerator–production process, but

unfortunately cannot provide speedup before it supports multiple accelerators at once [23]. LegUp is a popular open–source option that produces accelerator designs after profiling C programs, and can integrate accelerator communication into the original program [5]. Under a variety of hardware placement schemes, LegUp is able to effectively produce accelerator sets that explore a variety of program divisions between software and hardware.

Key to many of these acceleration toolchains is the use of reconfigurable devices like *FPGAs*, which can be continually reloaded to represent different hardware devices [9]. FPGAs enable powerful physical tests of accelerators before they are produced as permanent circuitry in devices like *ASICs*. Beyond this, FPGAs can sustain adaptable devices that change their processors or attached accelerators over time to suit changing needs. Kang's *SOAR* architecture does precisely this for a FPGA–based *RISC–V* system, the *Rocket Chip* [13]. SOAR frequently evaluates the usage and effectiveness of currently–configured accelerators. This lets it determine whether the accelerators should remain available or if power should be saved by switching availability to their slower software counterparts instead. The accelerator set is usually a large collection of small, targeted accelerators like Burbage's binary sequence–generating accelerator [3]. Over longer periods, the system proposes to compare on–chip accelerators to designs stored in memory or the cloud: if a different combination of accelerators would serve the system better, a new collection of accelerators could be written to the FPGA during a period of system downtime. Although this accelerator framework primarily focuses on smaller, functional accelerators, rather than the large memory–focused accelerator that we plan to implement, it does use the same environment that we plan on using, demonstrating the feasibility of adapting this architecture for further use.

The accelerator toolchain we will use is part of a suite of open–source hardware development tools produced at UC Berkeley. The RISC–V ISA drives this suite: it is open–source and prioritizes the *reduced instruction set computing* (RISC) model of instruction set architectures while still providing enough flexibility for designers to add a few of their own custom instructions [24]. Berkeley's Rocket Chip *system–on–a–chip* (SoC) implements this ISA and is configurable, allowing the adjustment of processor cores and the memory system [1]. One configuration even allows replacing the memory hierarchy with a scratchpad memory, although it is limited to a strict scratchpad implementation, with only a single processor core, one scratchpad, and no external DRAM connected [7].

The Rocket Chip's other strength lies in the ease of its integration with hardware accelerators through its *Rocket Custom Coprocessor* (RoCC) interface. This system fluidly transfers custom RISC–V instructions from the core to designated accelerators [1]. Customizing the Rocket Chip is made simpler by the fact that this whole system is produced in *Chisel*, a hardware design language powered by Scala [2]. Chisel is used to write accelerators as well, allowing their creators to use high–level programming constructs to generate circuitry efficiently. Developers can transform a finished design into *Verilog*, which then can be simulated using cycle–accurate simulators like Verilator, mapped to FPGAs through a tool like Vivado, or even fabricated in silicon. Further extensions to the Rocket Chip by the *lowRISC* project prepare the chip for Xilinx's Nexys A7 FPGA and enable support for Linux [17, 9].

2.6 Summary

Memory latency has been a problem for a long time, with extensive research done on every possibility for reducing or tolerating it. However, with its consequences increasing, and less growth in architectural mitigations, the performance costs of memory access are quickly becoming more visible. The transparent nature and confusing effects of parts of the memory hierarchy have likewise opened an opportunity for progress. Sometimes programs are executed less–than–ideally due to the simplifying assumptions of caches, whether that is due to undesirable cache evictions or the sharing of sensitive information. Additionally, when tested, several of these applications seem to benefit from increased developer control over the memory the applications use. We have also seen examples of scratchpad memories handling complex workloads while increasing performance, and cooperating in memory hierarchies alongside more traditional caches. Providing more control over memory to the programmers that desire it through a secure and richly–featured scratchpad is thus feasible and, in some cases, desirable. In the next chapter we discuss the environment we will use for implementing and testing our scratchpad–based accelerator.

Chapter 3

Environment

Our acceleration platform has been chosen to provide the Scratchpad Accelerator with a development environment that supports detailed, fast configuration and powerful testing. This primarily prompted our choice of the Rocket Chip for our accelerator's base SoC. The Rocket Chip's support for accelerator development has been proven by the plentiful array of coprocessors built for it since its creation. Its RISC–V ISA is a great representative of the RISC architecture that we hoped to target. Together, this lets the Rocket Chip provide a solid base for implementing our accelerator's hardware. To test the effects of our device on software, we also needed to examine the accelerator in a fuller system. This is why we targeted the lowRISC extension of Rocket, which lets us run our accelerator on an FPGA and lets it interact with the processes of a full Linux distribution. Each of these systems has an effect on the shape and performance of our accelerator. In this chapter, we will explore our development environment by expanding on important features of RISC–V, the Rocket Chip, lowRISC, and the Nexys A7.

3.1 The RISC–V Rocket Core

The design needs of our accelerator drove our consideration of which acceleration platform to target. Ideally, the memory accesses that our accelerator supports will be low–latency, consistent, and flexible. These traits ensure the accelerator is usable enough that it has a chance of enabling memory acceleration. Because we need fast responses whenever an accelerator instruction arises, we need a SoC where our accelerator can be tightly coupled to the operation of the core processor. Another environmental consideration comes from the pattern of use of accelerator instructions. In particular, the *Instruction Set Architecture* (ISA) that we target has a profound impact on how the accelerator will be accessed.

With that in mind, we chose to implement the accelerator for a processor implementing the RISC–V ISA. Firstly, it is an open–source standard, with many open–source hardware implementations and supporting ecosystems [24]. Its design, especially its basis in a RISC model of computing, also provides useful opportunities for the accelerator. RISC–V is a *load–store architecture*—meaning that data operations may only happen in registers, with data drawn from the actual memory hierarchy.

Because all RISC ISAs are composed of short instructions running simple operations, these memory accesses can be frequent and need to be quite efficient. Our accelerator's instructions should enable the same support for this style of computation. So our access instructions must be similarly fast, short and as simple as the instructions they aim to replace.

We have chosen the Rocket Chip for our base system because it is open–source, explicitly supports accelerator development, and enables a fast and flexible hardware development cycle [1]. Its design makes it easy to generate hardware repeatedly and relatively quickly, particularly for FPGA boards, but also for silicon. The Rocket Chip also makes augmentation easier because it is written in a high–level *Hardware Description Language* (HDL). All designs are constructed through configurable circuit generators written in Chisel. Since Chisel is an extension of the Scala programming language, it gives hardware developers access to powerful Scala programming constructs [2]. These abstractions are useful for making the building blocks of hardware, making development much less tedious than in more common HDLs like Verilog.

The Rocket Chip also aids accelerator development through its RoCC interface. This allows coprocessors to be designed independently from the processor. RoCC's protocol governs instructions and responses transferred between the accelerator and the core, linking the accelerator's operation to the main execution pipeline. An instruction transfer happens on clock edges where two conditions are met: the accelerator signals it is ready for a new instruction, and the core signals that a relevant instruction is valid. This transfer is typical of the *Ready–Valid* protocol, a construct used throughout the Rocket Chip. RoCC also allows accelerators to raise hardware interrupts to the processor, which is useful for error or notification sharing. Accelerators can read important core status data through access to a selected portion of *Control and Status Registers* (CSRs). Finally, accelerators can implement other parts of the RoCC interface to connect to the memory hierarchy. This model of development prescribes the connections between accelerators and the surrounding system. It lets developers focus on the accelerator they create, rather than the hardware design of the Rocket Chip as a whole, while still keeping the accelerator close to the core. Figure 3.1 shows the layout of the configurable Rocket SoC, with `Tile2` containing a Rocket core and linked RoCC accelerator.

RISC–V supports accelerator development as well, in that its ISA reserves several instructions as *custom* instructions, designated for hardware extensions on a system–by–system basis. RoCC allows coprocessor developers to designate which of these should be used by their accelerator. Then, whenever the core decodes a chosen instruction, it will be forwarded to their accelerator and not acted on anywhere else. Figure 3.2 shows the layout of the `Custom0` instruction. The least–significant 7 bits designate the instruction type, with the other 25 left for the transmission of data according to the ISA extender's wishes. Our simple access instructions need to entirely fit into one of these instructions so accesses stay fast. We also want all other instructions to our accelerator to fit into the same custom instruction so that we avoid using up all the custom instruction space of the Rocket Chip. It makes instruction encoding trickier but allows our accelerator to be more versatile. Thus any instructions to our accelerator will be subtypes of the `Custom0` RISC–V instruction.

One downside to the RoCC system, for our use, is in the manner that instructions are forwarded to the accelerator. Our accelerator should parallel the L1 cache because it supports accesses to similar types of data: a moderate amount of highly–used information. However, the handling of

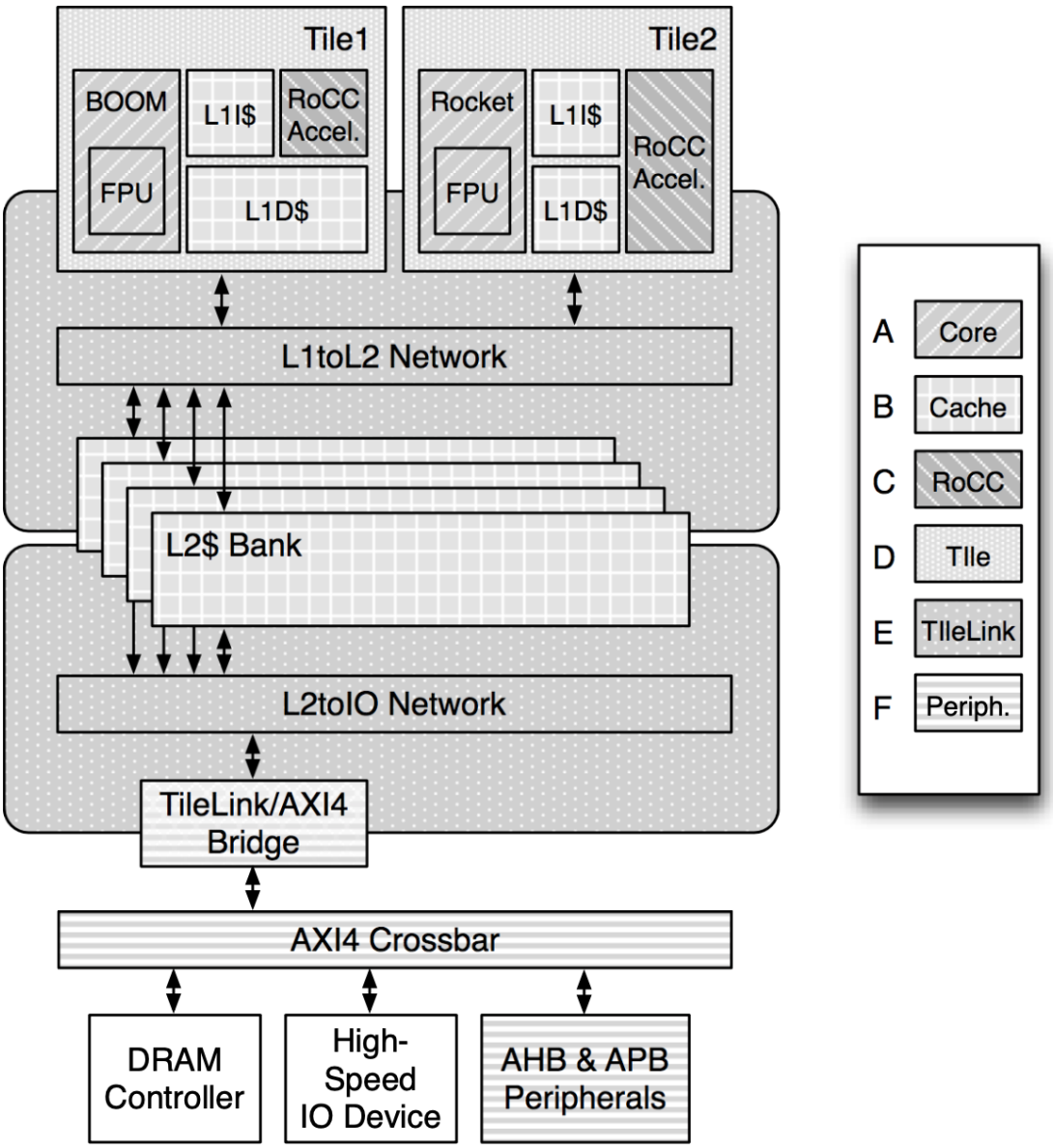

Figure 3.1: Block Diagram of the Rocket SoC, with a RoCC–accelerated Rocket core in Tile2. Figure from Asanović et al. [1]

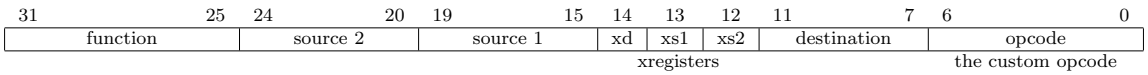

Figure 3.2: Format of a RoCC Custom Instruction.

accelerator and memory instructions by the Rocket Chip's pipeline do not lie in parallel. Memory stores and loads access data in the *memory* (MEM) pipeline stage, the fourth of five pipeline stages implemented by the Rocket Chip and displayed in Figure 3.3. However, the RoCC interface does not forward instructions to accelerators until the *writeback* (WB) stage of the pipeline. This simplifies instruction handling for accelerators, since instructions are not speculative in this stage, and any instruction the accelerator receives will be retired. However, it impedes communication between the accelerator and core, especially when a response is needed from the accelerator. In the Rocket Chip, RoCC instructions that cause data hazards induce large pipeline stalls, of at least four cycles if the hazarding instruction is immediately after the RoCC instruction [19]. This hurts the efficiency of accelerated instructions. Ideally the benefits of the attached accelerator, or the quantity of the work done per instruction, outweigh this latency overhead. However, the cost is significant enough to merit exploration of alternative placements of the accelerator within the instruction pipeline.

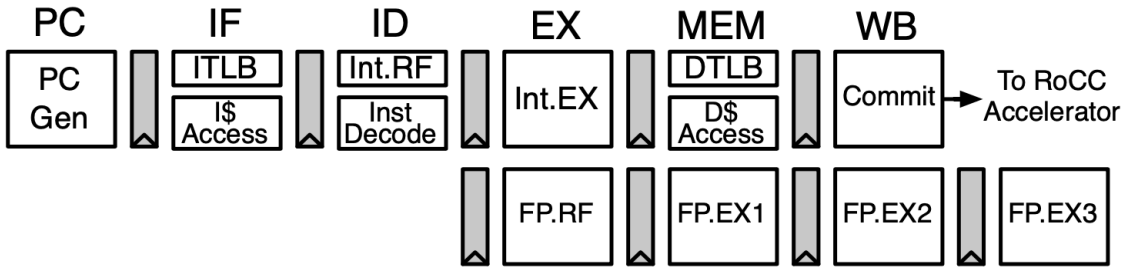

Figure 3.3: Rocket core pipeline, with figure from Asanović et al. [1]. After the writeback stage, custom instructions are forwarded to the relevant RoCC accelerator.

3.2 lowRISC

The next consideration for our accelerator is its testing environment. To create a processor that is fully ready–for–use in hardware, the easiest option is to use the lowRISC extension of Rocket. LowRISC connects the basic design of the processor to the hardware resources of the *Nexys A7* FPGA board, allowing the processor to run on a physical board [17]. It also prepares the SoC to boot with a RISC–V version of Ubuntu Linux, transforming the Rocket Chip from a bare metal design into a standard computer.

In the development workflow of lowRISC, Rocket Chip circuit generators written in high–level Chisel code are first elaborated into Verilog circuit designs. Then the *Vivado High–Level–Synthesis* (HLS) tool crafts hardware circuit descriptions from these designs that actually target the Nexys A7. These can be flashed to the board when it is plugged into the development computer. From there, the

board can be run autonomously, with Linux booting whenever the power is turned on. Reconfiguring the board simply involves recreating a new hardware bitstream and flashing it to the device [9].

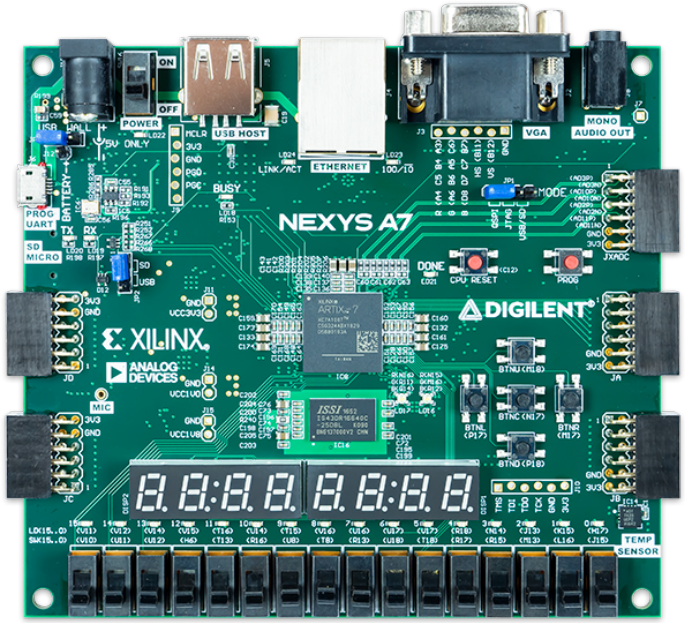

Figure 3.4: The Nexys A7–100T by Xilinx. We use this FPGA board for testing our accelerator within its broader system–on–chip. Figure from Digilent [9].

An overview of the Nexys A7 is provided in Figure 3.4. Most of the peripherals are not connected to lowRISC's circuitry, but we use the VGA and USB ports to connect a keyboard and display to the computer for basic input. For a better interaction experience, with a more sophisticated terminal, we also connected an ethernet cable to a NUC Linux development computer and communicated with lowRISC through `ssh`.

Another form of user interaction comes through the board's eight–digit display, linked to lowRISC and the RoCC system for debugging. In previous research, we developed the At–a–Glance Debugger for the Nexys A7 and the Rocket Chip [4]. It allows accelerator designers to connect important internal wires to the display diagrammed in Figure 3.5. Display digits can show hexadecimal values for larger bus or register values, but digits can also be broken into their eight segments to communicate individual binary signals. Data can also be captured in different ways: displayed in realtime or recorded from events like an incoming instruction or an error. This visibility aids in hardware development and debugging, and was invaluable in helping us test the hardware for this research. However, it also provides an aid to the software developer who interfaces with the accelerator. OS support for RoCC constructs is vital for their proper functioning in a fully–running system. Since we do not modify the operating system in this work, peripherals like the display can help make up for the lack of full accelerator support in the OS. Error signals that aren't yet handled by the core, for instance, can be communicated with users through the lights as they debug invalid accelerator instructions. Hypothetical display state after a given instruction is rendered in Figure

3.6, and Appendix B provides a reference for debugging with the At–a–Glance Debugger.

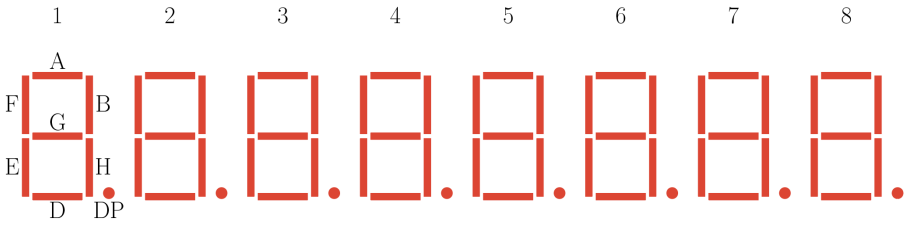

Figure 3.5: To validate Scratchpad performance, we used the Nexys A7's 8–digit display to view wire and register values. Here each segment is labeled according to its position and digit.

Figure 3.6: With our use of the digit display to show single–bit wire values and display more complex register data through hexadecimal values, this is what it could look like after a given Scratchpad Instruction. Segment 2A would have been on if the last instruction erred.

Beyond the board's peripherals, the FPGA resources of the Nexys A7 also place constraints on the design of lowRISC and its accelerators. The FPGA on board has 63,400 6–input *Lookup Tables* (LUTs), 126,800 *Flip–Flops*, and 4,860 Kb of *Block RAM* (BRAM) [9]. These set the amount of logical circuitry, registers, and RAM–based large–scale storage that our design can use, and much of it is already taken up by lowRISC with its single–core Rocket Chip. During synthesis of our circuit, Vivado does its best to map our design to these resources in an efficient manner, while keeping *data path delay* low. Path delay is caused by the numbers of logic levels that signals pass through within one cycle. To keep a system running accurately with higher delay, clock frequency must be reduced, an undesirable constraint. So there are multiple reasons to keep our resource usage small, even as the storage size of our scratchpad motivates growth in our accelerator's size. Therefore, the features and constraints of the Nexys A7 and lowRISC influence our accelerator design beyond our original acceleration goals.

Both the hardware design of the Rocket Chip and the actual hardware of the Nexys A7 shape how our accelerator can positively influence its system. In this implementation, the main goal of the Scratchpad Accelerator is to demonstrate how scratchpad–augmented memory systems can be a boon to RISC–V systems in general. In the next chapter we discuss the principles underlying our accelerator design. We eventually describe the Instruction Set Architecture and hardware model of the resulting accelerator.

Chapter 4

Scratchpad Design

The design of our accelerator follows our primary motivations in embarking on this research. Programmers don't have much control over the memory hierarchies that determine so strongly how their systems perform. More control could be used to increase performance, strengthen security guarantees, and enable new memory abstraction models for popular data structures. The capabilities of the Scratchpad Accelerator directly address each of these principles of security, performance, and abstraction support, with programmer usability as the overarching goal.

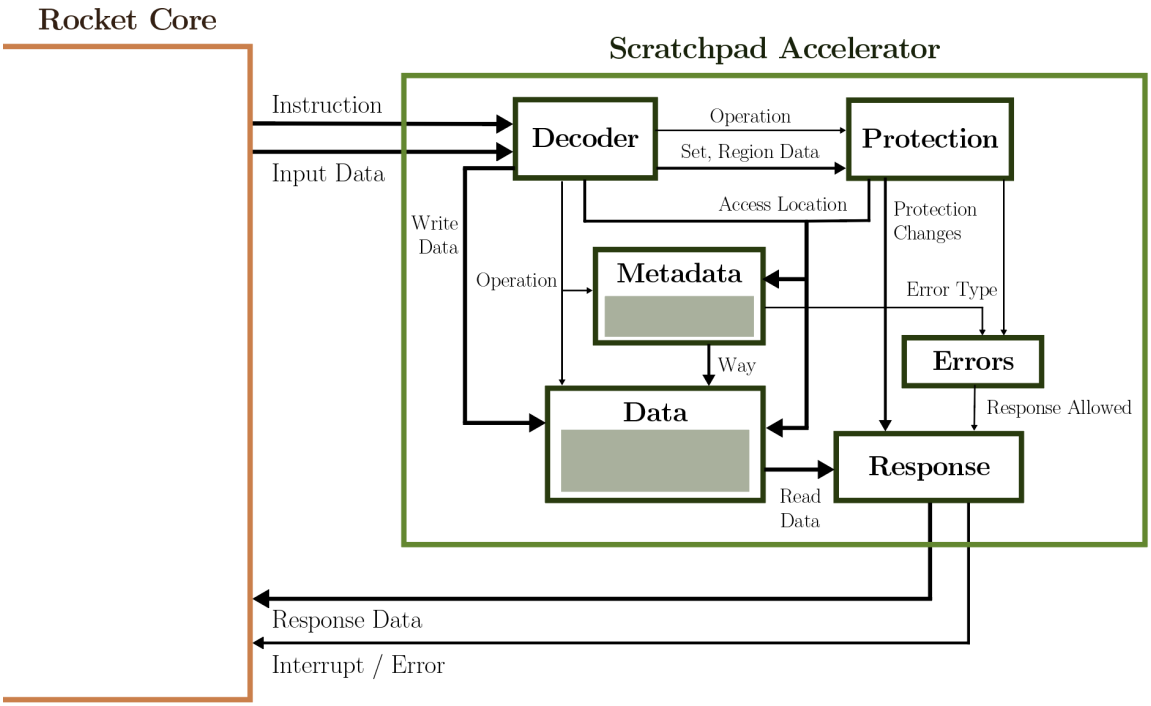

Figure 4.1: Block Diagram of the Scratchpad Accelerator, showing the most important functions and data. Arrow size indicates the width of dataflow.

Figure 4.1 shows the SCRATCHPAD ACCELERATOR's basic design, with basic functions broken into separate blocks and important dataflow shown as arrows between them. The design ensures efficient, safe manipulation of user data in the DATA unit. This is controlled through the METADATA and PROTECTION units. Safe communication with the computer core, and thus the user, is guided by the ERROR and RESPONSE blocks.

The regular flow of instruction processing is as follows. **(1)** Instructions from the core first get decoded, with input data unbundled into usable sections. **(2)** The next stop for most instructions is the PROTECTION unit, which checks that data–accessing instructions have the right credentials to trigger their requests. **(3)** Once authority is guaranteed, the instruction can carry out its effects. **(a)** Data–accessing instructions would access the METADATA unit, locating the data accessed by the operation and verifying logical validity, updating the metadata as needed. **(b)** These instructions would finally access the DATA unit itself, and forward any requested data to the response unit for eventual transmission to the core. **(3')** Other instructions may change settings in other units, or probe for their data, so response data can come from other sources as well. **(4)** Finally, if an instruction is improperly formatted or makes invalid requests, an error can be produced instead of a response. The error will be transmitted to the core, where an interrupt is raised to notify the environment. It is also visible on the AT–A–GLANCE DEBUGGER's display.

To understand the details of each of these units, we first examine the operational needs of the scratchpad. The next section will delve into the SCRATCHPAD ACCELERATOR's ISA, describing each operation and how they guide the accelerator as a whole. Finally, we will return to the accelerator's logical layout.

4.1 Instruction Set Architecture

The highest priority of the SCRATCHPAD ACCELERATOR is to provide control over the data storage powering a developer's program, so it is important for us to develop useful data–access operations. This need aligns well with the memory model underlying scratchpad memories. They don't control their own data replacement like caches do, so software instead must control which data is moved in and out of this memory. To provide fine–grained control over our scratchpad, we need several types of access operations. Beyond providing ways to insert and edit information, we need developers to be able to remove data as well, so they can use the scratchpad for other projects. Thus we have the motivation for three different data–access operations: `Put`, `Get`, and `Remove`.

Data Access

Each operation must be able to access data anywhere in the scratchpad, but what types of access are allowed? The only data portal into the scratchpad lies in registers, which are 64 bits wide on our system. Thus access operations are limited to 64 bits so that each operation can be done in a single transaction using one register. This is a little limited for a design that prioritizes programmer control and flexibility. We don't want to limit programmers to retrieving 64–bit *longs* each time they use the scratchpad. This is because RISC–based load–store architectures perform data manipulation in

registers only. If scratchpad data is moved to registers before being operated on, it is most efficient to move them in the form they'll need for each manipulation. Preparing the data efficiently involves giving programmers room to specify what data size they want (ranging from a *char*, 8 bits, to a long, 64), and addressing the data location in a flexible manner. Assembly instructions address data in many ways, but two of the most common are the *base–offset* style and *immediate* access styles, and we enable both. Thus, access instructions can either index data by its complete address, or provide a base pointer and a smaller offset amount to easily access array– or stack–based data. Additionally, we constrain access addresses to be size–aligned to their data type, so while the storage is byte–addressable, odd–offsets may only be indexed by char–sized access instructions. This is a common strategy for RISC–V systems and simplifies storage control.

The final concern in defining accesses lies in our *address space*. The regular memory hierarchy supports a much wider virtual address space than it actually has room for, so virtual addresses must be transformed into physical addresses during data retrieval. However, our accelerator is designed to be a very small memory system, since it only parallels the size of the L1 cache. Is such a big virtual address space desirable? We believe so, because if the accelerator were used to augment normal program operation, programmers will prefer to change their memory access patterns with minimal effort. Memory is often accessed using pointers up to 64 bits long, the size of a lowRISC register. If our Scratchpad Accelerator supported the same extent of virtual addressing, it would allow for simple translation between the two storage systems. This is very useful for shadow memory applications in particular. More specifically, we should be able to support the same address width as the main memory system uses, so for the 48–bit virtual address space of the Rocket Chip, our accelerator must also support at least 48–bit addressing.

To provide further functionality through the accelerator, we also enable atomicity by providing *Load–Reserved* and *Store–Conditional* instructions. These are not offset–addressable, but still allow four different sizes of access along any address in the scratchpad. Only one address can be reserved at once. The only conditions causing a conditional store to fail arise when the reserved data has been modified, which is required, or if another `Load Reserved` instruction has been issued. Upon a successful store, `Store Conditional` will return 0, and on failure it returns 1. These access instructions are all summarized in Table 4.1, which lists the parameters and return values for every Scratchpad Accelerator instruction.

Stripes and Regions

With so much storage capability given to each instruction, we need to consider how to protect the data of each process from instructions issued by others. This is where protection handling comes in: we use isolation, authorization, and authentication to prevent unwanted accesses. First, Scratchpad Accelerator instructions must be inseparably linked with the process making them. We support this by storing the current *process identifier* (PID) in the Scratchpad Accelerator, and using the PID to authorize data accesses. To keep the current PID accurate, it is only changed through the `Set PID` instruction. This instruction is secured because in the normal configuration of our accelerator, `Set PID` can only be called from *Supervisor* mode. Then for proper protection,

	Parameters		
	Immediate	Source Registers	Destination Register
Access			
Put	Size, Offset	Value, Address	———
Get	Size, Offset	Address	Value
Remove	Size, Offset	Address	Value
Protection			
Reserve Region	Stripes	———	Region Index
Set Region		Region Index	———
Clear Region		Region Index	———
Free Region		Region Index	———
Atomics			
Load Reserved	Size	Address	Value
Store Conditional	Size	Value, Address	Success
Administrative			
Investigate Error		———	Error Code
Get Parameters		———	Parameters
Get Owned Regions		———	Region Mask
Set PID	Privilege Level	PID	———

Table 4.1: Scratchpad Operations

the operating system should issue this authentication change during every context switch. Since PIDs are unique during any program's execution, there is no way for one process to imitate another through its operations when both are running. If the OS ensures that a process' scratchpad data is cleared when the process finishes, then there is no way for other processes to access that data. So if the OS is trusted, Scratchpad Accelerator data can only be accessed by the process that owns it.

The data itself stays isolated to a particular program through the protection of scratchpad portions called *stripes*. The scratchpad is divided into a set number of these stripes, each identical in size. Programs can request a certain number of stripes to operate in through the `Reserve Region` instruction. If there are enough stripes available, the scratchpad will reserve them and return their unique region index back to the program for later reference. Upon reservation, those stripes now record which process they belong to. Their data may only be accessed if the scratchpad's current PID is set to their PID, and the current region is set to part of those stripes through the `Set Region` instruction. To reuse a region quickly, a program can wipe it using the `Clear Region` call. A region can also be safely freed up for another program's use through the `Free Region` instruction. With these four instructions, storage can be shared between multiple programs, fluidly changing hands beyond context switches, while still keeping data isolated and protected.

There are several reasons why our protection is tied to stripe regions, rather than a simpler protection scheme. We could simply isolate scratchpad data through temporal or spatial divisions. This would mean restricting use of the whole accelerator to one program at a time, or keeping each program to a designated set of addresses, like those of one stripe. But both extremes place

undesirable limitations on the use of the accelerator. The latter would always constrain programs to small portions of the scratchpad, and the former would force some programs to wait unnecessarily for scratchpad access when prior programs were using only part of the accelerator. Instead, we want to prioritize acceleration flexibility, letting developers decide when to prioritize storage space or fast reservations. The striping system's mixture of temporal and spatial isolation lets developers use the SCRATCHPAD ACCELERATOR efficiently for their needs. Since we also want SCRATCHPAD ACCELERATOR accesses to be fast, our striping system strikes a good balance between flexibility and hardware complexity. Authentication for an access can be done quickly by examining simple region and stripe protection state. A more granular division of data protection would require more resources. Together, these flexibility decisions are useful compromises for a new memory system. They prioritize the utility of the accelerator across a variety of developer needs, reinforcing the general–purpose nature of this device. As programs are developed for the SCRATCHPAD ACCELERATOR, we can catalog their use of the accelerator and fine–tune the balance struck here between efficiency, flexibility, and shareability.

Security is only one reason for the current striping system, however. The accelerator has multiple models for its access: where some may prefer its vault–like security properties, others could prefer the built–in–support for key–value storage structures. This raises the possibility that developers would use the scratchpad with multiple models in mind during the same program. Stripe–based accesses allow programmers to separate these models from each other. One program thus can request regions multiple times, at different periods or overlapping ones, and each region's data and access patterns are kept completely separate. This is particularly important because the scratchpad's actual storage size is considerably smaller than its address space. We maintain this structure by ensuring that addresses fall within the current region by shifting them to an allowable line index. Then if the same program wanted to shadow memory within the scratchpad and use it elsewhere for storing a simple array–based data structure, the array's data could conflict with mapped memory locations if the programmer didn't also reserve space for that array in the memory hierarchy. By isolating each use case to its own region, our hardware handles that issue for the user.

Region–based access is an important part of our accelerator design, and in the block model of our accelerator, it has its own control unit devoted to it, in the form of the PROTECTION unit. We will later delve into the details of how it works, from protection to address translation. For now, it is enough to know that regions are referenced through IDs given by the `Reserve Region` operation. With those IDs, regions can be set, their data cleared, or even totally freed, in similar manner to how memory is allocated from the heap in C programs. Region controls take place with familiar constructs and simple single–instruction transitions between memory access cases. This minimizes the instructional overhead in preparing for scratchpad use.

Administration

The final set of instructions for the accelerator are purely informational, but still very useful for development with the SCRATCHPAD ACCELERATOR. The first, `Investigate Error`, identifies the most recent accelerator error and its cause. There are five current error types, enumerated in Table

4.2. Errors can result from improper data addressing, unauthorized region manipulation, attempted data insertion to a full location, invalid or unauthorized stripe IDs, and requesting stripes that are already in use. The first four are completely within the control of the programmer to avoid, but are easy enough mistakes to make that a notification of their occurrence is useful. The last error could simply be due to poor runtime timing, and could block any use of the scratchpad until a portion has been freed—a much more significant problem. We aim to inform the user immediately when an error has happened by sending a hardware interrupt signal to the Rocket core, whereby an appropriately–configured operating system will send a signal to the program. Programs could include signal handlers that catch these signals and use `Investigate Error` to verify that an accelerator error has indeed happened, and determine what to do from there. They could also include this instruction after attempting to reserve a region. However, a faster way to check this is by remembering that on an error, the scratchpad will only ever return the value 0. Since no region's index is ever 0, a program could simply check that call's return value to quickly see its success.

Code	**Type**
0	No Error
1	Out of Space
2	Unauthorized Instruction
3	Out of Stripes
4	Bad Location Reference
5	Bad Stripe Reference

Table 4.2: Scratchpad errors and their codes.

The next administrative instruction is `Get Owned Regions`, an instruction that is mostly useful for cleaning up any reserved regions that have been mistakenly left reserved after they are finished for use. The instruction returns a stripe mask, with 1s covering each stripe that is reserved for the current process. It is hugely important not to have storage leaks with the scratchpad, because no other program could gain access to that hardware once one program incorrectly cleaned up or unexpectedly terminated early. Thus, we foresee the `Get Owned Regions` instruction being useful during program teardown, where it can be called by the operating system to verify that the current process has freed all of its scratchpad regions. If it hasn't, then the operating system can simply free the regions for the program.

The final instruction, `Get Parameters`, simply returns the scratchpad's current dimensions and configuration to the user. In basic use, this provides a sanity check to the programmer that the data storage is of a large–enough size. This instruction provides further information, however. All of the key scratchpad dimensions, from the number of kilobytes it stores, to the address–bits it expects, can be easily parameterized from our Chisel code, as a set of 6 parameters to the scratchpad generator. Although we have settled on our preferred combination of characteristics, designed to work well with the hardware constraints of the Nexys A7 and to maximize usability, other systems may desire a different configuration. Some programs will be flexible enough, without needing manual adjustments, to work with a range of scratchpad configurations. But others may require a given size or layout to work correctly, so a developer would want to verify the parameters on a system before use. This

is especially relevant on a fast–development device like an FPGA, where new computers and their changing accelerators can be synthesized and flashed to the device on demand.

4.2 Instruction Encoding

Encoding each of these desired functions, along with room to expand the set and provide all their necessary parameters, meant carefully carving space from a single 32–bit RISC–V instruction. Figure 4.2 shows the `Custom0` instruction, where only bits 31-15 and 11-7—so 22 bits total—are completely free for our use. Additionally, they are only free for use under certain constraints: if an instruction requires a source or destination register, then five of our remaining bits are needed to encode the register used. With three registers, up to 15 bits of the 22 may be lost. Luckily, not all of our instructions need all three extra registers, which we can use to create an efficient, logical, and extensible instruction set. Table 4.3 lists the encodings for all 13 of our Scratchpad Accelerator instructions.

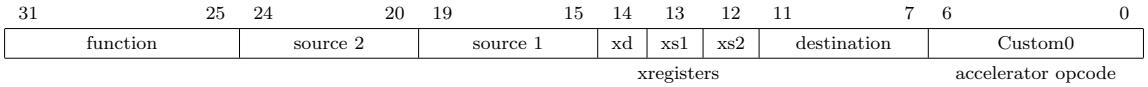

Figure 4.2: Format of a RoCC `Custom0` instruction.

31	30 29	28 25	24 20	19 15	14	13	12	11 7	6 0	
0	size	offset[8:5]	Address	Value	0	1	1	offset[4:0]	Custom0	Put
0	size	offset[8:5]	offset[4:0]	Address	1	1	0	Value	Custom0	Get
0	size	offset[8:5]	Address	offset[4:0]	1	1	1	Value	Custom0	Remove
1		0100	stripes		1	0	0	Index	Custom0	Reserve Region
1		0101	Index		0	1	1		Custom0	Set Region
1		0110	Index		0	1	1		Custom0	Clear Region
1		0111	Index		0	1	1		Custom0	Free Region
1	size	1001		Address	1	1	0	Value	Custom0	Load Reserved
1	size	1000	Address	Value	1	1	1	Success	Custom0	Store Conditional
1		1010			1	0	0	Error	Custom0	Investigate Error
1		1011			1	0	0	Params	Custom0	Get Parameters
1		1100			1	0	0	Mask	Custom0	Get Owned Regions
1		1111	PID		0	1	1		Custom0	Set PID

Table 4.3: Scratchpad ISA Listing. Capitalized variables are stored in the corresponding register, and lowercased are immediate values.

Access Instructions

Our first task involved encoding the access instructions, since they would receive the bulk of use, and they consistently require more registers and parameters than the other instructions. We also wanted one of those parameters, the offset, to be encoded as an immediate value in these instructions. This

makes program execution more efficient for accessing data like local variables or structure members, where offsets are known at compile–time. To increase the offset's usefulness, we needed to make room for as many offset bits as possible. For instance, if the offset is used to access a local variable of the current stack frame, the offset must have enough bits to cover a useful number of frame widths. The key insight for creating these instructions is that every access instruction, `Put`, `Get`, and `Remove`, only needs two registers. They are either used for two inputs, or applied so that one is an input and one is an output. This frees the bits that are normally used to point to the third register. So on top of the seven `function` bits left free to use, we have five extras. How can we maximize their value? First, we take the most–significant bit as the `mode` bit, signifying whether the instruction is indeed an access, rather than an administrative or atomic instruction. Then we need two bits to encode which of four data sizes we are accessing. Finally, the last nine bits are used to represent the `offset`, even if these bits are broken across different regions of the instruction. Splitting the offset in this way is simply a small added compile–time cost, since reassembling the offset in hardware can be done easily within a cycle. Likewise, using 9 bits gives us an offset up to 512 bits from the base register, enabling a wide variety of uses.

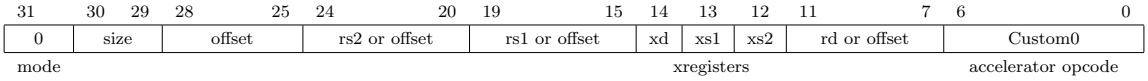

Figure 4.3: Format of an access instruction.

The general format of a Scratchpad Accelerator access instruction is outlined in Figure 4.3. We haven't explained how to differentiate the three access instructions from each other, but that can be done simply by checking the `xregisters` bits. These three bits of the `Custom0` instruction must be set in accordance with the register types used. The `Put` instruction is the only one that doesn't use a destination register, so `xd` is 0. If we make `Get` use source register 1, then `Get` is the only instruction with `xs2` set to 0. Finally, `Remove` will put its data in source register 2 rather than 1. However, this instruction must still set `xs1`, because on the Rocket Chip the second source register can only be used if the first is. Even though `xs1` is set unnecessarily, `Remove` still uses a different combination of `xregisters` than the other two. These three unique settings of `xregisters` allow us to differentiate between the access instructions using a minimal number of bits.

Special Instructions

It was far simpler to encode the remaining instructions, since they use fewer immediate values. These instructions all have the `mode` bit set to 1, and we distinguish between them by four `opcode` bits, with a unique code created for each instruction discussed earlier. Figure 4.4 shows how these special instructions are formatted. Since we only have ten special instructions at the moment, this leaves room for six more without any change to our decoder. The only noteworthy parameter handling happens for the `Reserve Region` instruction, which encodes the number of stripes requested as an immediate value. Since we support configuration of the scratchpad into up to 16 stripes, we need 5 bits to encode this value safely. It also is convenient to encode this value as an immediate. The

region size used by a program is a key, rarely–changed value for the program, and can thus be easily deduced at compile–time.

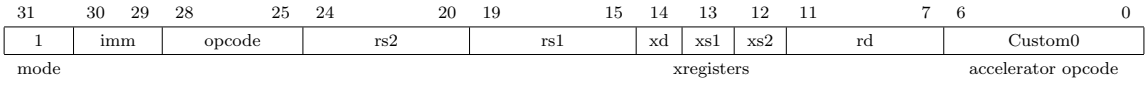

Figure 4.4: Format of a special instruction.

Since we have four `opcode` bits we can easily support 19 instructions. With some modifications to the hardware, but no modifications to these defined encodings, we could extend the number of `opcode` bits to the left and add even more instructions if desired. This will be useful for the future as we determine what scratchpad functions are most useful to developers and compiler–writers. More information–oriented instructions could help with optimizing scratchpad use, determining how much of the scratchpad is full at a given time and when and why it is overburdened or under–utilized. Special scratchpad operations could also further its use as a vault or sandbox, keeping key operations efficient and private to the scratchpad. If it turns out it is most useful as a key–value data store for example, accelerated search, insertion, and deletion operations across wider ranges of data than simply one scratchpad line would be desirable for developers. With many free hardware resources and enough flexible instruction bits, we can expand the accelerator's functionality to meet these future needs.

4.3 Hardware Design

With our instructions defined and encoded, we can return to the discussion of their implementation in hardware. Figure 4.5 revisits the Scratchpad Accelerator's design with the added dimension of time, showing how different units interact over different clock cycles.

Instruction processing begins with the Decoder unit, which accepts input from the Rocket core every time a `Custom0` instruction is fired. Eventually, instructions make their way to the relevant blocks, changing state or accessing data. Finally, the instruction results in an error or correct response, at which point, control is returned back to the core along with any extra data. Since some of our units require a clock edge to capture data and make changes—as a hint, they involve the use of BRAM, a *synchronous memory*—then instruction latency lies within two cycles. However, the Scratchpad Accelerator pipelines instruction execution so that one instruction can finish execution each cycle. Then, in every cycle, the accelerator can accept a new RoCC command and return another's response data. Each block of the Scratchpad Accelerator was carefully designed to minimize latency and support this pipeline. Units communicate early and operate in parallel to keep delays short. In the rest of this section, we delve into the design and operation of each block.

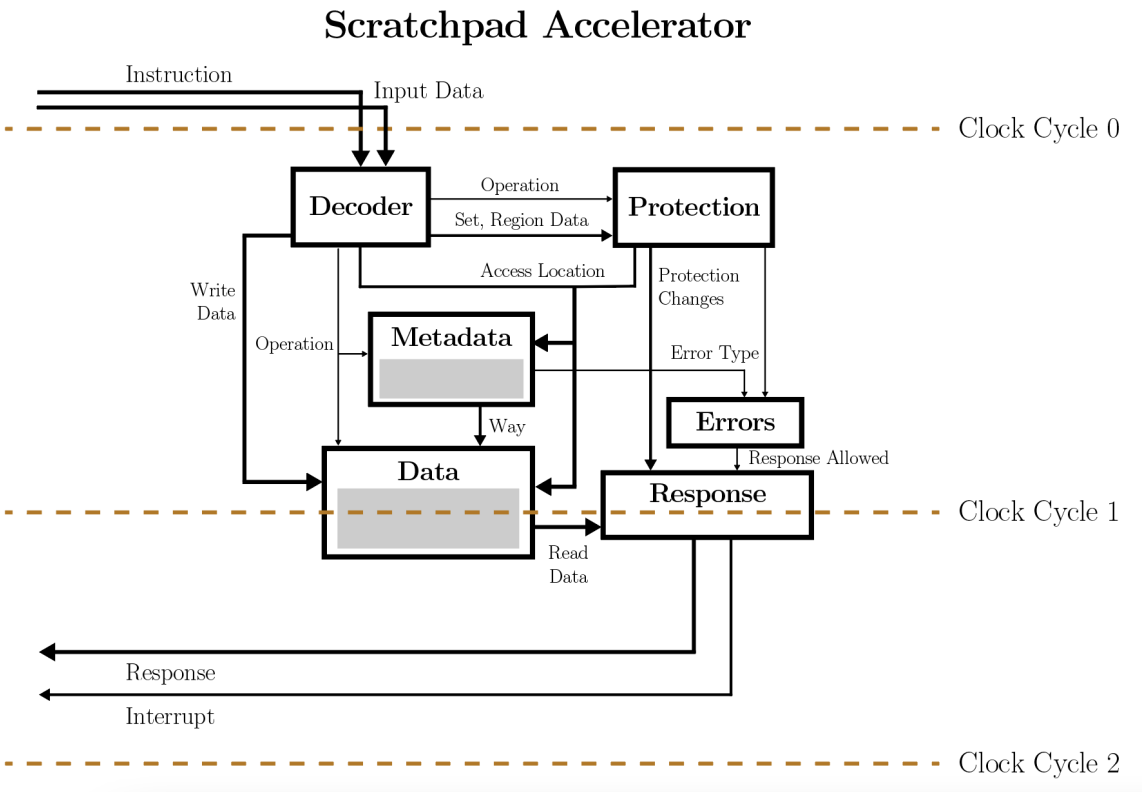

Figure 4.5: Block diagram of the SCRATCHPAD ACCELERATOR pipeline, receiving instruction and input data from the Rocket core and outputting the response and interrupt back. Pipeline progress at different clock cycles is labeled.

Decoding Instructions

The DECODER unit is designed to quickly and reliably determine the accelerator's state of operation, at the same time uncovering all required input data for use in the rest of the scratchpad. The instruction code, along with up to two registers of source data, connect as inputs to this block. On a rising clock edge, when the accelerator is ready for a new instruction and the core signals that a `Custom0` instruction is transmitted, the DECODER block captures incoming data and sets to work. It deciphers immediate–encoded parameters and deterministically interprets the source registers used by the current instruction. This is where the ISA encoding is translated into hardware action, and the DECODER must make use of all the subtle differentiating bits between each instruction type in order to unbundle the instruction's relevant parameters from simple input wires. Figure 4.6 shows how the examination of up to five bits allows the unit to determine instructions with minimal effort, which is useful when we can only pack so many logical units between clock cycles without introducing excessive gate delay. Listing 1 excerpts some of the Chisel code that describes the DECODER unit.

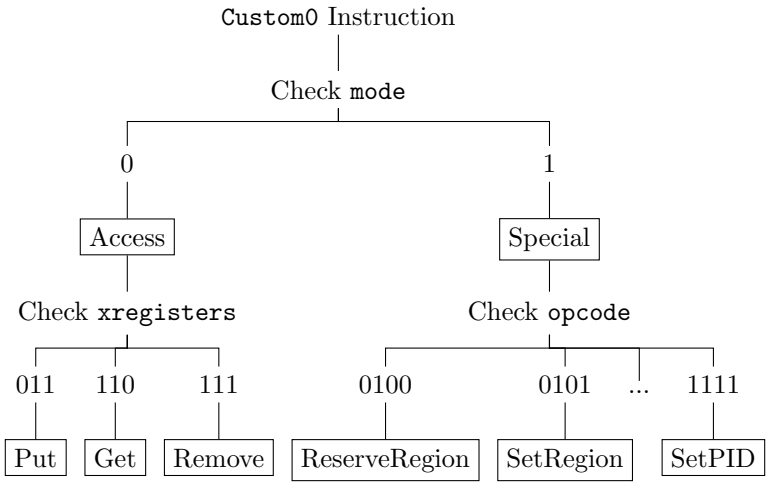

Figure 4.6: Instruction decoding, providing room for 19 unique instructions.

With the instruction determined, the DECODER focuses on outputting clear data to the rest of the accelerator. First, the next cycle of accelerator state is prepared based on whether the instruction will need to ensure a response is sent. This could take some cycles to handle if the core is stalling. The accelerator always needs to be aware of when it is busy handling instructions, erring on an instruction, or sitting idly, so that its instruction pipeline works properly and efficiently. The state of individual blocks is also prepared from the DECODER, with a large number of state–bit wires sent out. An example of this for the `Load-Reserved` and `Reserve Region` instructions is displayed in Listing 1. In general, the protection and metadata units will need to know exactly what operation is about to happen so that accesses and modifications are properly vetted for the right prerequisites. So the DECODER sends out bits determining the type of access to attempt, like a `read`, `write`, `reserve`, `free`, or `delete`. Finally, the DECODER pipes input data to various units, whether write data to the Data unit, requested stripe size to the PROTECTION unit, or data addresses to the METADATA

block. Control and information are thus secured through the DECODER.

```
// Examine the opcode bits of an instruction
switch(opcode) {
    ...
    // Match an instruction type, sending key signals and data to other blocks
    is(o_lr) { // Load-Reserved
      rawAddress := rs1_1
      padRead := true.B
      padLoadLink := true.B
      // When simulated in software, can print here for debugging information
      printf("Got lr instruction: %x \n", io.cmd.bits.inst.funct)
    }
    ...
    is(o_reserve) { // Reserve Region
      requestedWidth := code_1.rs2(regionBits,0)
      when(!reserveRegion(requestedWidth)) {
        newError := e_no_stripes
      }
      printf("Got reserve instruction: %x \n", io.cmd.bits.inst.funct)
    }
    ...
}
```

Listing 1: A switch statement is used to easily identify an instruction's opcode and act based on the operation type.

One special case for the DECODER lies in handling the `SetPID` instruction. This is the only instruction that requires a certain privilege level to execute, at least in the properly–secured configuration of the accelerator. We do allow the SCRATCHPAD ACCELERATOR to be configured without this protection, but it is only for demonstration purposes. Leaving protection off allows the accelerator to be tested in systems whose OSes do not properly secure the accelerator during context switches and process lifecycles. In the Rocket Chip, privilege level is recorded in CSRs that the core always has access to. Luckily, an important subset of these registers is passed along to RoCC accelerators for use at any given time. These CSRs are in sync with instructions passed from the core, so our accelerator can restrict use of the `SetPID` instruction by checking the privilege CSR for a value above *user mode*. On the Rocket Chip, these correspond to supervisor, hypervisor, or machine modes.

Protecting Data

The `Set PID` instruction operates on the mechanisms of the PROTECTION block, which restricts the range of data–accessing instructions. That's why PROTECTION is the second block involved in handling any instruction, and its handling of a data–access instruction is visualized in Figure 4.7. Data isolation, protection, and specialization is secured here, so it is important that its functionality is never bypassed, always locked–down, and quickly available to the units who take actions based on its dictums. Scratchpad data's ownership state is completely described by the `Stripe Table`, which

recalls whether each stripe is in use or not. If a stripe is reserved, the table records which process owns it. During an attempted data reference, the current PID for the scratchpad and the current scratchpad region are compared with the stripe table, to see if all stripes for the region are in use and owned by the process. A simple valid access bit is generated from this check, but it has power: it allows accesses to happen and causes `bad location reference` errors upon invalid attempts.

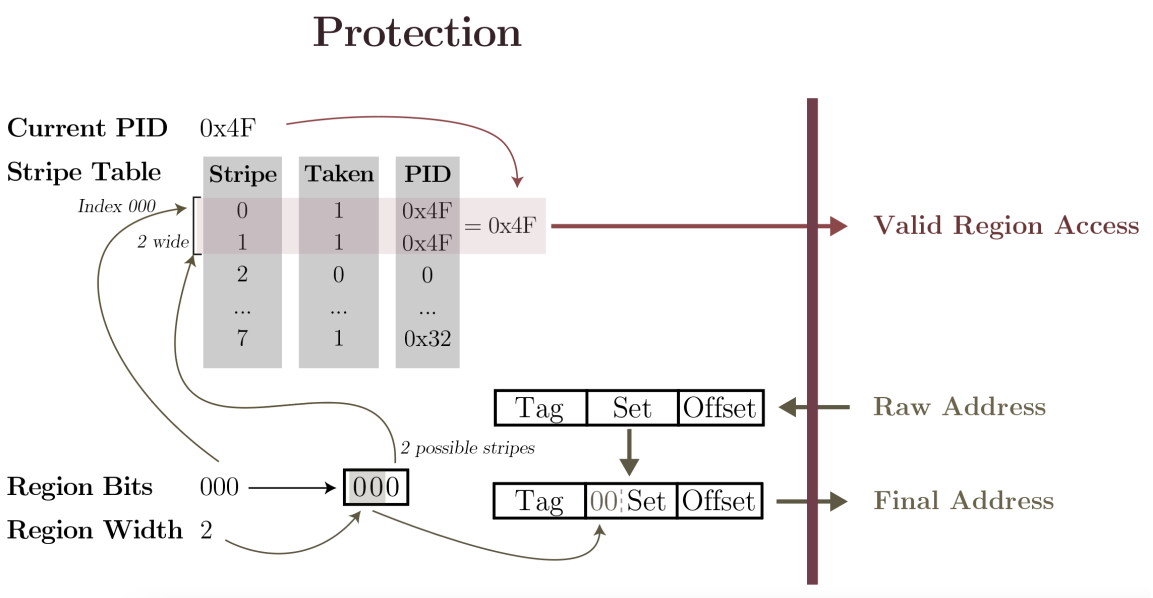

Figure 4.7: Layout of the PROTECTION block, showing the process of vetting a data access instruction.

Data accesses are not fully secured until we know that their addresses lie within the current region. We do not vet instructions for this—instead, the hardware alters these access addresses to fit the current region. One benefit of this approach is that the hardware can easily translate non–scratchpad pointers into scratchpad pointers, and vice versa, so that developers do not need to reserve special regions of the scratchpad for a given variable. Likewise, hardware designers don't need to configure huge scratchpads in order for pointers of given shapes and sizes to merely work with our system. Since scratchpads are partitioned into regions numbered by a power of two, it is actually quite convenient to modify access addresses into the correct region. The top $\log_2$ stripes bits of the set index given by any address should simply point to the index of a stripe within the region. Yet these should not just be wired to point to a single stripe, since the region may be of a greater size, and this would leave portions of the scratchpad unusable. Rather, we replace a different number of bits depending on the region size, so that the lower bits are free to point to different stripes in the same region, and more replaced bits means a region composed of fewer stripes. The state registers `Region Bits` and `Region Width` keep track of this, with the former storing the index of the lowest stripe contained in the region, and the latter tracking how many of the topmost set index bits to replace. Every time a new address is sent to the PROTECTION block, the top `Region Width` bits of the address' set index are replaced with those upper bits of `Region Bits`.

The PROTECTION interface also has a set of operations devoted to configuring it: the `Reserve`, `Set`, `Clear`, and `Free Region` operations, along with the `Set PID` instruction. These simply change the state of the PROTECTION block when allowed. Simple hardware masks check that region operations are valid, with regions that are powers of two in size, properly aligned, and free or owned by the current process when they should be. If any of those preconditions are not true, a `bad stripe reference` signal is sent from the PROTECTION unit to the error unit, and rather than returning successfully, the error unit will coordinate a failure response. Likewise, on an unauthorized attempt to change the PID, an `unauthorized instruction` failure is sent to the ERROR block, and no PID change will occur, with an error sent back to the core.

Storing Metadata

If a data–accessing instruction succeeds in passing through the PROTECTION block, then METADATA is next in the path of control flow. The METADATA unit mimics the structure of the DATA unit, but provides important information about each byte of storage: whether it is in use or free to fill, and what data is associated with it. These values are stored through per–line metadata, where each has an in–use bit, a tag bitstring, and valid bits specifying which bytes of the line store accurate data. This metadata is needed because the scratchpad is organized like a *set–associative cache*. Its associativity depends on matching each used line of data to one of many potential addresses. Since the scratchpad's data replacement is controlled by explicit data insertion and removal through software instructions, the valid bits are needed so the hardware knows exactly when storage is free to be reused differently.

The main role of the METADATA unit is to determine whether data accesses should succeed or not. Then input data is needed to point to the relevant storage location and specify the desired metadata through the access type that should take place. For most accesses, the address is transmitted to the METADATA unit, where the set index is used to search every way simultaneously for the correct lines of information. If a read–related access is requested, then the correct way will have a line that is in use, storing a tag that matches the address's tag. Additionally, the bytes indicated by the address' offset and the access size should all be valid. If they are not, a `bad location reference` error is sent to the ERROR block and no access will take place. Figure 4.8 demonstrates how a successful metadata read access would take place. Other accesses require different metadata. A write to a given set either needs an unused line or an in–use line with the correct tag. If such a line is available, the proper valid bits will be set for that line. Writes to new lines will additionally set the line to in–use. However, if no line is available for a write, a `out of space` error will be shared, with no write occurring. `Remove` operations operate in opposition to `Write`. They unset valid bits for the relevant line, and even unset a line's in–use bit when none of its bytes are valid anymore. As with a `Read`, an improper address will result in a `bad location reference` error instead. This covers basic metadata accesses, but some instructions are worth further investigation.

Special instructions affect the METADATA block as well. These instructions work in non–standard ways. The `Clear` and `Free Region` instructions will invalidate every line for the selected region, setting in–use bits and valid bits to zero for the whole of the selected region. Finally, atomic

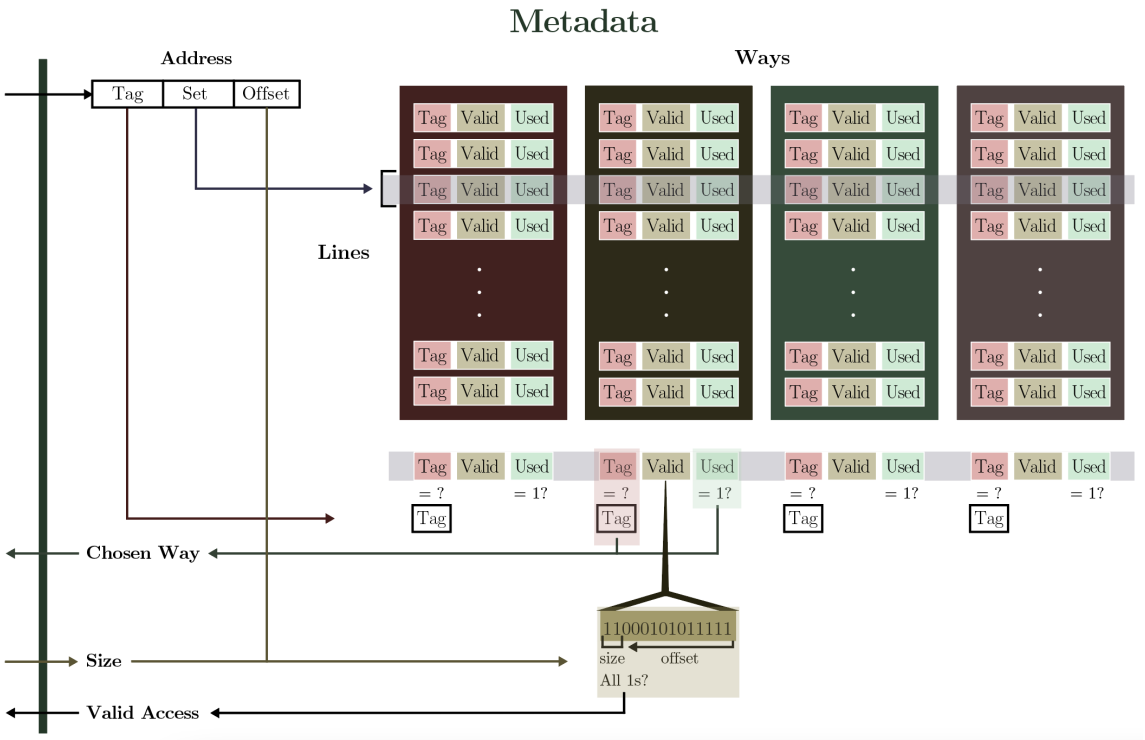

Figure 4.8: Layout of the METADATA block. Here, we demonstrate the verification needed for an attempted data read. An address and size are supplied, which are used to locate the right metadata location, find the way storing the desired data, and check that the metadata's valid bits are set at the right location.

instructions act like their corresponding accesses, so `Load Reserved` performs a read and `Store Conditional` performs a write. However, there is also state in the METADATA block to remember the currently–reserved line, along with which bytes within it are reserved. One bit, the `stillReserved` register, records whether any modification to that line has taken place since its reservation. On a `Load Reserved`, these are set, and a successful `Store Conditional` must access a reserved location that remains unchanged. If `stillReserved` is not still true, or the reserved address does not match the address of `Store Conditional`, then the store will not happen. In either case, the METADATA block controls data modification through its signalling.

Scratchpad Data Storage

Finally, valid access instructions receive access to the DATA block. Data is organized into BRAM–based ways. Each way has a single port available for reading or writing, but not during the same cycle. One clock edge is required to capture data for insertion or retrieve data from the desired location. So catching up on this latency cost is a major priority for the access pipeline. Once access locations are vetted through the PROTECTION and METADATA units, the signal to read or write is quickly granted to the DATA unit, ensuring that the access can complete in the minimum required time. We demonstrate the effects of a write operation in the DATA block in Figure 4.9.

Accessed lines need to be processed to properly share their data with the core's registers. The METADATA unit supplies the correct way to use in a given access. For write operations, data must be masked to the correct access size and moved to the correct offset in its line. So inputs like the write data, a mask of which bytes to write, and the line index to modify, are shared with the correct way's BRAM block to make the proposed modifications. Reading follows inverse logic. Since the BRAM always reads when not writing, each way returns the data stored at the current line index input. From these outputs, a cycle later, the desired way's data is masked and shifted by the correct offset amount. If the METADATA and PROTECTION units are satisfied, this data can then be safely returned to the core.

Connecting Details

These storage units shape the bulk of the accelerator's configuration options. Table 4.4 displays which dimensions our ISA and hardware design natively support. The shape of the scratchpad is extremely flexible, with the line size, number of ways, and total scratchpad size easily configured. Protection can be tuned by changing the number of stripes in the pad, and for OSes that do not support accelerator protection, like ours, the protection of the `Set PID` operation can be disabled. Finally, the address space can be configured, whether developers want it to match that of their system (ours is 48 bits), the width of a register, or to only cover the scratchpad's size itself.

The final sections of the accelerator are quite simple: the ERROR and RESPONSE units simply transfer data back to the core in safe and efficient ways. Since data retrieval from BRAM storage takes a cycle, the accelerator enables request pipelining so throughput is increased. However, this, along with sometimes–delayed core requests, meant that we needed a queue of waiting–responses to ensure every instruction could be handled by the core. If any accelerator unit signalled an error, the

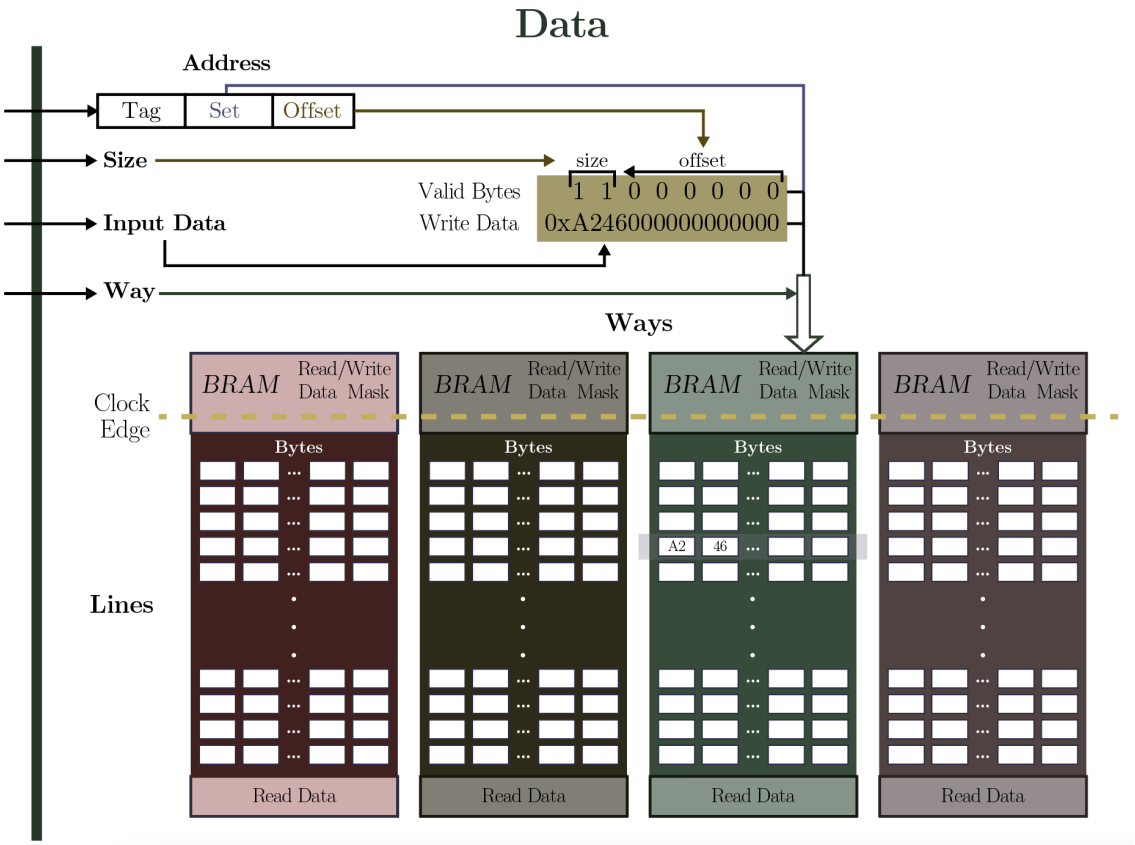

Figure 4.9: Layout of the DATA block, visualizing the process of inserting data into the third way.

Hardware Setting	Options
Total Size	**1**, 2, 4, 8, 16, 32 kilobytes
Line Size	**8**, 16, 32, 64 bytes
Associativity	1, 2, 4, **8** ways
Address Bits	$\leq$ 64 bits
Protection Enabled	on or **off**
Stripes	1, 2, **4**, 8, 16 stripes

Table 4.4: Ready–to–go hardware configuration options. Bolded options represent those used in our accelerator configuration during experimentation.

Error unit would instead record the error, the accelerator state would change, and a response of 0, rather than any potentially–leaked sensitive data, would be returned. All in–flight instructions would be cancelled and an interrupt signal to the core is raised, so that the OS can handle the trap accordingly, with safe software using a signal handler to catch these error states. The `Investigate Error` instruction is particularly useful here, because it retrieves the latest error code for the current process from the accelerator, explaining what went wrong. Together, the Error and Response units ensure that exactly the right information is released from the scratchpad, ensuring vault safety.

Once fully–implemented, our hardware design can be viewed in Vivado in terms of the blocks of FPGA resources touched by the accelerator module. Figure 4.10 highlights the FPGA regions used by our accelerator within the larger Rocket core. Further detail is provided in Figure 4.11.

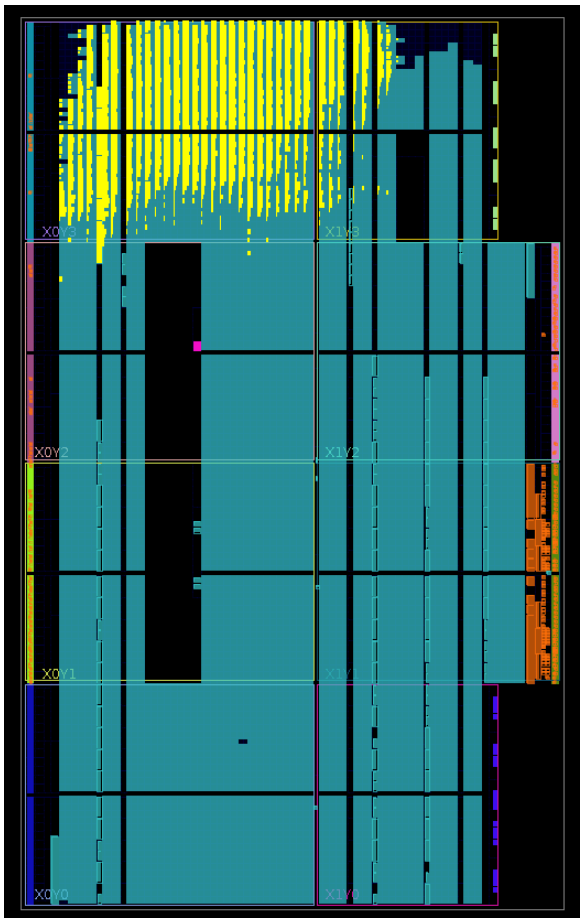

Figure 4.10: After elaboration for the Artix 7 FPGA, we use Vivado's Implementation visualizer to show the resources taken up by the scratchpad, highlighted in yellow. This accelerator is configured with 1 kilobyte of storage space, 8–byte lines, and 8 ways of associativity.

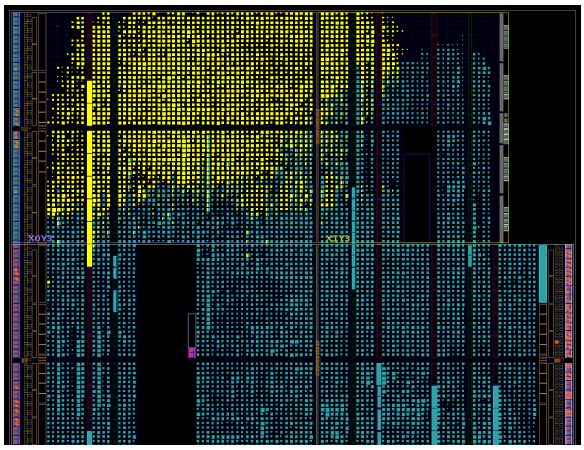

Figure 4.11: Another view of the same elaborated scratchpad; here, the scratchpad's region is blown up to show detail.

Each design unit emphasizes efficiency and protection, allowing the accelerator as a whole to expose enough storage information to users to be useful and novel. At the same time the Scratchpad Accelerator keeps their data safe and ready–for–use. By adapting current storage structures to enable greater access and flexibility for developers, the accelerator is put in a good position to enable exciting new models of memory use for software developers. In the next chapter we explore some of these potential applications.

Chapter 5

Applications

By granting software increased control over memory organization, our accelerator gives programmers new freedom in how they conceptualize and work with memory. This can help them adapt their use of memory to help with needs like high–performance or high–security computing. The scratchpad's layout also enables new addressing techniques for data stored within it. These additional methods of accessing memory encourage new memory models to have in mind during software development, aiding in the implementation of certain applications.

5.1 Levels of Access

Scratchpad data accesses are designed to expand flexibility in working with memory while still staying intuitive, similar to regular RISC–V loads and stores. Our `Put` and `Get` instructions correspond neatly to the original `Store` and `Load` instructions, each taking a base pointer, offset value, and data width to access. Our `Remove` instruction takes the same parameters as `Get`, likewise matching `Load`. The only difference between our set and the original is that our offset parameter is bounded by 9, rather than 12, bits. During the design process we found this limitation to be manageable, because across a variety of compiled C programs, we rarely found offsets that exceeded $2^9 = 512$. Then in the majority of cases, regular memory accesses can be translated between the two systems by simply swapping the original assembly instructions for accelerator instructions. Many of these memory instructions are beyond the control of the C programmer, inserted during compilation time. To migrate these instructions to the scratchpad, the compiler would have to be altered, or the assembly code it generates must changed through an external program. As we have not embarked on compiler changes in this work, we will instead start with applications involving memory instructions that are directly controlled from C programs. These are pointer–based accesses of data, which we can translate into scratchpad accesses by replacing each pointer dereference with a scratchpad instruction.

To facilitate user interaction in C we've written several interface levels for the scratchpad. The lowest level is a set of C macros which take an instruction type and its parameter values to produce a `Custom0` inline assembly instruction. Then users can control the accelerator through simple

function–based syntax, easing the learning curve and readability of scratchpad–targeted development. Portability between accelerated and regular code is also covered by these macros, because at the flip of a compiler flag, they can produce pointer–based accesses instead of scratchpad calls. This allows a program designed for the scratchpad to instead target the normal memory hierarchy when desired.

Complex applications call for a further abstraction layer, which we have implemented as a library for higher–level memory accesses. Here we replicate useful memory manipulation functions which the C standard library provides for interfacing data in the normal memory hierarchy. From memory–control functions like `memcpy` to scratchpad–specific needs like hardware configuration validation, these interface functions provide users with foundational tools for building complex programs. This library makes it easy to translate between scratchpad and regular memory targets, since where our library replicates standard memory functions, it requires the same data in the same expected form. This limits the development burden for developing accelerated programs to the initial transition to this library. If a developer decides to adapt their code for use with the accelerator, then they simply need to ensure that scratchpad data is accessed only through our custom functions rather than the standard methods. A future tool could modify these calls automatically to ease the tedium of translation, with the programmer simply designating which data should be kept in the scratchpad. By raising the abstraction level with this library, it is more convenient and safe for C developers to implement data structures atop the scratchpad on demand.

5.2 Basic Access

One of the most common pointer–based data accesses in C happens when storage is allocated from the heap for large data structures. This can both be easily migrated to the scratchpad and provides the potential for increased performance. In the process of using a heap–based data structure, we need to first reserve its memory, initialize and interact with the data, then free the data later. How we'd do this in regular memory is well–known: first attempt to `malloc` for the size of the array, then access its elements, and finally `free` the pointer. It's as simple to migrate this use to the Scratchpad Accelerator: first attempt to reserve a region of storage, then use scratchpad macros for accessing data, and free the region in the end. Figure 5.1 provides a comparison of these two methods: the C statements required and the complexity of each is commensurate. Once a developer understands the interface for our scratchpad, it is very simple to use it as a replacement for long–term storage like that of the heap. The only concerns to worry about are size: the scratchpad has a smaller limit than the heap and must be reserved in set stripe widths, so only certain data structures can or should be ported. However, this benefit sweetens the tradeoff: for these highly–used data structures, instant access to their elements is now guaranteed. Coherency troubles won't plague the access of this data structure even if it is large or access patterns to it are irregular, because the scratchpad won't move the data from fast–access memory like the cache might. For memory applications where the programmer, rather than the cache, knows best, the scratchpad can help users by simply standing in for sections of the heap.

```
// Regular access
// Setup
int* largeStructure = malloc(32000);

// Access
largeStructure[0] = 1;
int newValue = largeStructure[0];

// Cleanup
free(largeStructure);
```

```
// Scratchpad access
// Setup
uint16_t regionId = reserveRegion(8);
int* largeStructure = 0;

// Access
putArrayInt(largeStructure, 0, 1);
newValue = getArrayInt(largeStructure, 0);

// Cleanup
releaseRegion(regionId);
```

Figure 5.1: Simple transfer of a heap–based data structure into the scratchpad.

5.3 Accelerating Data Access

The size and access pattern of a data structure determines the efficiency of implementing it on the scratchpad rather than the normal memory system. Acceleration takes extra development effort and can only apply to certain structure layouts, so there are material and pragmatic limits to which programs would use the Scratchpad Accelerator. The development effort involved in acceleration for latency's sake is best justified in high–performance applications. Here, programmers have a sizeable interest in increasing efficiency and are intimately familiar with the details of their program's data layout. These developers are in the right position to manually control which data is fast to access, since they are aware of the constraints of these methods and are willing to invest development time to save computation time.

To maximize access efficiency, high–performance developers would wish to use as much of the scratchpad's area in as few time–intensive operations as possible for their needs. The applications that will find the scratchpad useful are then defined by the size and shape of their important data structures. These must be small enough to fit in the scratchpad without much data reorganization but large enough to make the burden of reloading their data to caches significant. The most–suited applications will access these structures in ways particularly unsuited for normal cache operation, causing many unnecessary conflict misses during program operation. Patterns of access are likely non–linear and spread out over enough time that data which is often reused may be evicted from the cache before it is needed again.

Applications that meet these guidelines cover a wide range of data structures and themes. Any data which serves as a frequent reference over the program's lifecycle could qualify. If reloading that reference will be costly for the cache, it is even better–suited. Running binary searches over a sorted array is a case where the technique's non–linear access pattern, even if it is simple for us to define, will result in many cache misses during a new search if enough time has passed since the last overlapping

one. Traversing a linked list, where nodes could lie in any pattern throughout memory, is another simple and common operation which performs contrary to the cache's expectations. Recursive structures like trees, when not accessed in the same order they are stored, also cause issues. Accessed frequently enough, with not too large a size, they make great candidates for translation to the scratchpad. These structures can also be adapted for better cache performance at the cost of instructional latency. These reorganizations also can change the intuition behind data structures and their access, which might not be desirable. The alternative tradeoffs that follow memory acceleration through a scratchpad can thus provide an attractive alternative for increasing a data structure's memory performance.

```
// Regular access                          // Scratchpad access
int* largeArray = malloc(32000);           uint16_t regionId = reserveRegion(8);
                                           int* largeArray = 0; // Can be any address

largeArray[0] = 1;                         putInt(largeArray, 1);
int newValue = largeArray[0];              newValue = getInt(largeArray);

free(largeArray);                          releaseRegion(regionId);
```

Figure 5.2: Using the scratchpad to store an array.

The scratchpad prioritizes working with varied data structures through its symmetric design. Every scratchpad location has the same constant access speed. Furthermore, locations in the scratchpad do not need to be accessed by memory pointer: they are accessed by whatever model the programmer sees as useful. If the programmer wants to treat their region of the scratchpad as a single array, as in Figure 5.2, they can address it by simply providing the element's location. If the developer conceptualizes the pad as containing multiple data structures, they can provide the base location of the relevant structure and an offset into the desired element, as in Figure 5.3. These references, built from constants, can be determined at compile time. This is unlike most regular memory accesses and thus can reduce the amount of work done in assembly to set up a memory access. There is, of course, still the option of treating each spot as a stand–in for an address in the full memory hierarchy, but the programmer has the flexibility to use the accelerator with whatever access model they envision, and the accelerator will oblige.

```
// Regular access
int* firstArray = malloc(8000);
int* secondArray = malloc(16000);


firstArray[0] = 1;
int newValue = firstArray[0];
secondArray[1] = 3;

free(firstArray);
free(secondArray);
```

```
// Scratchpad access
uint16_t regionId = reserveRegion(8);
int* firstArray = 0;
int* secondArray = 8000;

putArrayInt(firstArray, 0, 1);
newValue = getArrayInt(firstArray, 0);
putArrayInt(secondArray, 1, 3);

releaseRegion(regionId);
```

Figure 5.3: Using the scratchpad to store multiple arrays.

Key–Value Stores

One type of data structure works particularly well for the scratchpad: the key–value store. That is because we have another available model of scratchpad addressing: where a given address represents a table pointer and a key. Together, this points to a set of key–value pairs, and if the key matches any of them, the access will target the value stored there. This works because of the scratchpad's associativity and lack of coherence policies: the user can decide that the tags that are stored for each value in a set are instead storing a value's key. When a scratchpad address is given, the table address works as its line index, pointing to the proper set, and the key fills in the address's tag bits. The structure of the scratchpad handles the rest. Figure 5.4 shows how a key–value store's interface can be constructed atop the scratchpad interface.

```
// Obtaining a key-value pair from one bucket
void insertPair(int key, int value) {
    int* bucketIndex = getBucket(key);
    putKVInt(bucketIndex, key, value);
}

int getValue(int key) {
    int* bucketIndex = getBucket(key);
    return getKVInt(bucketIndex, key);
}
```

Figure 5.4: Using the scratchpad to host a key–value store.

The Scratchpad Accelerator increases key–value lookup efficiency through its associativity. In hardware, the provided address's tag is compared to every tag stored in a given set, all in parallel.

So when we treat these tags as keys, we search through the whole set in one instruction call to find the matching value, and this provides hardware–level acceleration of key–value searches. Limitations to this parallelism come from the configuration of the scratchpad, and what level of associativity it has. If it has eight–way set associativity, and the key–value data store has been broken up into tables of eight key–value pairs, then searching for a given value using its key can be sped up by a factor of eight. All that is required is to consider how blocking the datastore into sets of eight will affect the formation or later access of the data structure, deciding where the proper set will need to be found in the access algorithms. This is quite feasible to do efficiently for large key–value stores like hash tables. Additionally, useful data structures like dispatch tables can be small enough to fit entirely into one bucket, ignoring this overhead.

Stack Management

One final approach to increasing data placement efficiency is less in the hands of the C developer than the compiler. Rather than moving heap–based structures, the compiler instead could focus on migrating portions of the stack, whose local variables and frequently–accessed data is certain to be in high use. Moving local variables cannot easily be done by a C programmer because so much of the work to load and store them from memory is done through assembly instructions, left out of control of the C program. These access instructions, along with *prologue* and *epilogue* handling of the stack frame, could be translated into instructions that work with the scratchpad instead. Since the scratchpad is limited in size, but most functions' stack requirements are known at compile time, then the compiler can determine how much of the stack is a good candidate for scratchpad storage. If more than one call frame is going to be stored in the scratchpad, then accesses to lower call frames through passed pointers must be careful to reference the right storage system, the scratchpad or regular memory. However, storing part of the stack in the scratchpad can provide efficiency benefits for programs that can't consistently keep important local variables in the L1 cache. Entirely removing these frequently–needed variables from the cache ensures these accesses are fast, while freeing up room in the cache for traversing large heap–based data structures.

5.4 Securing Data in the Vault

Keeping important data out of the regular memory hierarchy provides further benefits through its increased security. Side channel attacks that exploit different parts of the memory hierarchy can be devastating to security applications such as cryptographic functions. Although computer architecture is designed to keep process data isolated from other processes, the architecture of the memory hierarchy has repeatedly exhibited vulnerabilities that allow malicious processes to snoop on supposedly–protected data. This happens through sometimes–unmet assumptions held by the shared resources of the memory hierarchy. Since RISC–V is a load–store architecture, a program's secure data is often moved between registers and the rest of the memory hierarchy. When exploits like Spectre are discovered, this data is vulnerable to exposure. If a call frame can be moved entirely to the scratchpad, then all its local data can be kept separate from the regular memory hierarchy,

entirely invisible to it. Additionally, it is easier to secure the smaller, simpler storage system of the scratchpad to keep this data safe.

When sensitive data is stored in the scratchpad, then the memory hierarchy no longer has access to the data and thus can't share it. With proper securing of the scratchpad through the OS, data is only accessible to its owning process. The only visibility that one process can obtain into another's actual use of the scratchpad is very limited. A process could first determine which regions of the scratchpad are in use. This can be done quickly, by attempting to request different region sizes and remembering successes and failures. However, knowing how many stripes a process has reserved gives very little information about the data within. A process can also determine some information about when another process is using atomics. By attempting atomic instructions across context switches, a process can determine if other processes have used atomics recently. This is because any `Load-Reserved` cancels the last reservation. Yet this still does not give much insight into the data that one process has stored in the scratchpad. The actual data stored there is kept separate from all other processes by the striping design and OS support. In its proper configuration, stripes of data are only available to the process which requested them, and are wiped after the process is finished. This system of data storage means that accesses leave very few side effects that are visible across processes. So, by using the scratchpad, cryptographic operations can shield themselves from many side–channel attacks by avoiding the intimacy of sharing the resources of the broader memory hierarchy.

Memory Hierarchy Sidecar

There is one other benefit to this data isolation provided by the scratchpad: the invisibility of scratchpad use to the regular memory hierarchy makes it an impartial bystander during regular memory operations. This allows for interesting measurement and commentary on a system's memory hierarchy. It can be hard to measure statistics about the memory hierarchy through software because the measuring program itself affects memory state. If these programs instead stored their data in the scratchpad, with their stack itself migrated to scratchpad storage, then the hierarchy would be minimally affected by the observer. This can be extended through the use of the scratchpad for memory shadowing, where a portion of memory, and any accesses to it, are replicated within the scratchpad. If the original memory becomes unsuitable at any point, the scratchpad provides an isolated backup. If metadata about this memory, rather than duplicate data, is stored in the scratchpad, then this can help programs check for memory–control errors during operation. Although this is a narrower application than memory acceleration, we see it as an interesting and powerful capability of the accelerator.

This concludes our selection of potential Scratchpad Accelerator applications. Each wields an interesting feature of the scratchpad's layout, from its associativity to its stripes. Hopefully, these are well–suited to support various goals in memory acceleration, security, or integrity. In the next chapter we evaluate the performance of these applications along these lines.

Chapter 6

Experiments

The Scratchpad Accelerator has a broad range of potential applications where developers may be happy to trade the added complexity of acceleration for other benefits. In this chapter we measure the feasibility and performance of some of these use cases. Each application was tested on the same configuration of the accelerator, so our experiments demonstrate the bounds of useful applications for this configuration. Beyond that, these results suggest useful adjustments to this configuration of the accelerator. Changing the Scratchpad Accelerator's parameters helps prioritize the performance of different accelerator use cases. Before preparing the accelerator for long-term use in a system, its users and their needs should be considered to determine the appropriate configuration.

In this chapter we test the Scratchpad Accelerator on a variety of programs, comparing the performance of their standard software-based versions with their accelerated counterparts. Every experiment is conducted on the Ubuntu–running lowRISC machine (single-core), paired with a Scratchpad Accelerator. The scratchpad is configured with no OS protection, 8 associative ways, lines of 8 bytes, 48–bit addresses, 4 stripes, and only 1024 bytes of storage total. These settings mean the scratchpad will have 16 sets. LowRISC's L1 cache, by contrast, has 128 sets, 16 ways, 64 bytes per line, and 32 kilobytes of space total. However, it is split into an instruction and data cache, so the data cache only has 64 sets and 16 kilobytes of space.

The experiments in this chapter test our accelerator's three main application areas: memory access acceleration, key–value store lookups, and memory shadowing. Tests are composed of C programs targeted to software and hardware. Each experiment is timed using clock cycles, read from the CPU's clock cycle CSR directly surrounding each test. To keep tests consistent across hardware and software, we wrote each program targeted to software. To make the hardware tests, we compiled these programs into assembly language. From there, we identified each memory access instruction that targeted the data structure we would be placing in the scratchpad. We replaced each of these loads and stores with the corresponding Scratchpad Accelerator instruction. Finally, we assembled the hand-edited code. Software tests simply take the original C code and compile it directly into binaries.

6.1 Instruction Latency

Before considering our Scratchpad Accelerator's performance as part of broader applications, we wanted to gauge the performance of its most important instructions. The access instructions `Get` and `Put` are used the most frequently by scratchpad–accelerated programs. Significant runtime differences between regular programs and their accelerated counterparts will be due to these instructions. To test their timing differences, we timed the cycles used in each instruction call, reading clock cycles directly before and after a `Load`, `Store`, `Get`, or `Put` call. We used a separate program for each test, compiling them at optimization level 2, and verifying in the assembly code that cycles were really recorded directly surrounding each measured operation.

Table 6.1 shows the minimum cycles needed for these calls after 100 trials, representing the best instruction latency that could be expected. Times are higher when data is not loaded into the L1 cache or the CPU pipeline stalls. We found that for both hardware and software, loads, and stores, a single access instruction can take take one cycle by this scheme. However, regular memory hierarchy accesses can have a much greater maximum access time than the scratchpad accelerator. Across the 100 trials, accelerator stores in these always took 1 cycle to complete, but regular stores took a maximum of 59 cycles. Loads were more consistent in this test, with both hardware and software always taking 1 cycle. In software, this is because our test involved loading data that was recently prepared, so it was already in the L1 data cache and could always be accessed quickly.

	Memory Hierarchy	**Scratchpad**	**Scratchpad with Hazards**
load	1	1	8
store	1	1	2

Table 6.1: Cycles needed to perform simple memory access instructions.

By these results, Scratchpad Accelerator accesses would seem to always have a lower latency than memory hierarchy accesses. However, the way these individual instructions are resolved in the Rocket Chip's CPU pipeline is actually dependent on the instructions surrounding it. Sometimes instructions introduce pipeline *hazards* which add bubbles to the pipeline, reducing its overall throughput. Some pairings of instructions always introduce hazards because of the data dependencies or hardware resources shared between them. To understand the potential for these hazards to arise and affect the latency of a Scratchpad Accelerator call, we ran an additional experiment to time successive Scratchpad Accelerator accesses. We ran 100 trials for each, testing the latency of two Scratchpad Accelerator accesses in a row, with their source and destination registers remaining the same. The final column of Table 6.1 shows the results: although two Scratchpad Accelerator stores in a row have the expected latency of 2 cycles, two Scratchpad Accelerator loads take a minimum of 8, not 2, cycles.

To understand why hazards arise for these loads we referred to the Rocket Chip's Core codebase and Davide Pala's RoCC research [19]. RoCC instructions have several major timing limitations when traveling through the pipeline. This is mostly due to the fact that RoCC instructions are only handled in the writeback stage, rather than the execution or memory stages. Because of this, data

hazards are particularly serious for RoCC instructions. If an instruction is in the decode stage, and it uses a register that a RoCC instruction in a later stage will write to, the pipeline will stall. This stops any instructions from moving past the decode stage until the RoCC instruction is completely out of the pipeline. For a hazard between two consecutive instructions, this means the stall lasts 3 cycles. This makes sense for instructions that need to use registers in the execution stage, but for two RoCC instructions in a row, this is an unnecessary stall: they don't need input data until the writeback stage. Another limitation is the communication channel between the pipeline and an accelerator. Commands are routed to the appropriate accelerator during the writeback stage, but this adds a cycle of delay before the accelerator receives the instruction. After that, the accelerator must do its work, transfer a response back to the core, and finally the response will be written to a register file. The stall will continue until the register file has been changed. For most other instruction types, these hazards are less costly. First, most work happens in the interior of the pipeline, so the cost of changing RAM, which our accelerator must do before sending a response, is hidden. Second, results from one computation can immediately be *forwarded* to downstream instructions, letting them execute before register file changes happen. The design of the pipeline does not enable any of these features for RoCC responses, so when a hazard arises because of a RoCC instruction, the delay is costly. In Table 6.2 we explain the state of the pipeline during the processing of our 8–cycle pair of Scratchpad Accelerator load instructions.

IF	ID	EX	MEM	WB	Cycle	Hazard Reason	Accelerator
C	R	R	C	x	—	—	—
C	R	O	R	C	0	Use–after–RoCC–Write	—
C	R	O	O	R	1	Use–after–RoCC–Write	—
C	R	O	O	O	2	Use–after–RoCC–Write	Capturing Input
C	R	O	O	O	3	Wait for RoCC Response	Working
C	R	O	O	O	4	Wait for RoCC Response	Response Fired
C	R	O	O	O	5	Wait for RegFile Write	—
x	C	R	O	O	6	—	—
x	x	C	R	O	7	—	—
x	x	x	C	R	**8**	—	Capturing Input
x	x	x	x	C	9	—	Working

Table 6.2: State of the pipeline during two successive RoCC instructions, explaining why hazards appear. C is a cycle-read instruction, which captures the current cycle during the WB stage. R is a RoCC instruction, which also gets passed to the RoCC accelerator during the WB stage. O is a bubble, and x represents some unrelated instruction.

This latency tradeoff makes sense for reducing the complexity of accelerator development. It simplifies the core pipeline and reducing the need for RoCC accelerator designers to deal with pipeline state. However, it limits the viability of accelerators designed to deal with frequent, short–latency instructions, like ours. Because our accelerator replaces memory calls, it is very likely data hazards will arise, because useful data within the scratchpad will be retrieved before it is used in a program. This makes acceleration less likely for programs using frequent accelerated loads. Software loads still must deal with cache misses, but the consistent communication cost for our accelerator

could outweigh this over time.

6.2 Applications

With instruction latencies determined, we wanted to measure how acceleration affected applications. First, we looked at several common algorithms with limited data access locality. We compared regular implementations of these algorithms with implementations that held their data structures in the scratchpad. If access patterns and density are enough to cause L1 cache conflicts, then program latency will increase for the software trials. If data loads cause hazards, however, program latency will increase for hardware trials as well.

We first tested *predecessor binary search*, where an array of sorted numbers is probed to find the closest value under a given target. Since our scratchpad is limited to 1 kilobyte, we chose to test this on an array of each integer from 1 to 256. Immediately after array initialization, we timed the cycles needed to search for one value, either 255 or 128. The searched arrays were either stored in the scratchpad or the regular memory hierarchy through the heap. Finally, experiments were compiled at optimization level 0 or 2. Table 6.3 displays the results for each of these variations. Binary search only involves loading values from the array, so this experiment mostly compares scratchpad and L1 cache loads. Here, cache loads are uniformly faster. This divergence in load times is entirely due to the expensive data hazards that scratchpad loads can cause and the favorable placement of the heap–based array in the memory hierarchy. We also note that variance is frequently much higher in the timing of the memory hierarchy experiments. The unpredictability of the regular memory hierarchy can cause a wide range of performances across time for the same program.

	Unoptimized				**Optimization Level 2**			
Target	255		128		255		128	
Trial	Scratchpad	MH	Scratchpad	MH	Scratchpad	MH	Scratchpad	MH
1	908	664	708	548	289	212	237	187
2	908	693	716	526	283	212	245	190
3	908	663	722	526	273	209	245	184
4	909	665	707	528	264	211	246	229
5	905	663	705	528	269	209	250	185
6	908	666	717	529	276	212	245	196
7	896	706	714	538	273	209	245	184
8	896	662	723	491	273	213	244	182
9	912	670	709	525	272	215	247	187
10	899	665	703	540	259	213	242	190
Median	908	665	712	528	273	212	245	187
Speedup		0.732		0.742		0.777		0.763
Variance	29.9	206	44.4	203	65.9	3.65	10.2	171

Table 6.3: Cycles needed to search a 256–int array of the sorted numbers 1 through 256, for the value 128 or 255. We compare storing the array in the scratchpad to using the memory hierarchy, denoted here as MH.

We next measured the performance of *quicksort*, because it involves both loads and stores from the array that it sorts. Again, we created an array of 256 integers directly before timing the performance of the function. We tested one array with the integers 256 to 1, placed in reverse–sorted order. Another array contained a random assortment of integers. Both tests were run unoptimized, and Table 6.4 displays the timing results of 10 trials. Again, there is slowdown using the acceelerator, although it is less severe than with binary search. Fewer hazarding Scratchpad Accelerator instructions likely help with the accelerator's performance here, but there are still many loads susceptible to data hazards. Variance here varies across experiments, with three of the four resulting in the same magnitude of variance, while the randomized cache experiment proved very stable.

	Reversed		**Randomized**	
Trial	Scratchpad	Memory Hierarchy	Scratchpad	Memory Hierarchy
1	83212	71125	106599	91837
2	83178	71208	106580	91792
3	83213	71246	117967	91934
4	94541	71173	117632	91655
5	83238	82533	106455	91769
6	93770	71244	116303	91829
7	93365	71257	106585	91712
8	83160	71221	106520	91791
9	83238	81926	106547	91864
10	83172	82139	106616	91762
Median	83226	71245	106592	91792
Speedup		0.85605		0.86115
Variance	24072000	25378000	24394400	5527.9

Table 6.4: Cycles needed to run quicksort on a 256–element array of integers. We compare the performance when the array is in the scratchpad to when it is stored in the regular memory hierarchy.

Key–Value Stores

We wanted to test one other common application because of the special nature of the structure of our accelerator. The Scratchpad Accelerator's associativity enables parallel lookups of data within a set based on its tag value. In the last chapter, we anticipated that developers could use these tags as keys for a key–value store. Then, by addressing a certain store and key, users could insert and retrieve data more quickly through this set–level parallelism. Although many caches are associative, only the Scratchpad Accelerator allows users to do this because they have control over the exact storage locations of data within the scratchpad. There are a couple of constraints for testing and using this feature. Sets are naturally small, only eight lines wide on our Scratchpad Accelerator configuration. So our experiment compares a set–based lookup against other small data structures.

Because fast lookups are a key benefit of hash tables, we took the small data structures that

power hash tables as our comparisons. Chained hash tables use linked lists to store items that hash to the same bucket, so we first ran this experiment on a linked list. On insertions it would need to allocate a new node for a key–value integer pair, prepending it to the list's head, and on lookups the list would be traversed to find the right key–value pair. Since these insertions require memory allocation and the scratchpad's do not, we also ran an experiment on a modified list that grows in a pre–allocated array. Key–value pairs are stored together, and the list must be searched sequentially on lookups. Additionally, the address of the new head of the list is known on an insertion, so the pair can be placed immediately. Finally, we compared this to insertions and lookups in a single set of the scratchpad, where addresses must be correctly composed of the bucket to search and the key to access.

	Insert			**Retrieve**		
Trial	Scratchpad	Array List	Linked List	Scratchpad	Array List	Linked List
1	152	226	13080	212	473	370
2	150	177	12569	211	465	370
3	150	229	12202	198	463	364
4	154	176	12326	205	468	362
5	152	179	12465	214	472	363
6	150	183	12229	211	466	361
7	152	186	13008	213	462	370
8	153	180	12033	211	471	357
9	154	179	12832	211	472	338
10	150	176	12826	203	471	368
Median	152	180	12517	211	469.5	364
HW Speedup		**1.18**	**82.3**		**2.23**	**1.72**
Variance	2.41	378	119510	23.9	14.8	83.4

Table 6.5: Cycles needed to fill a key–value store of size 8, either with dynamic array–based accesses or scratchpad "bucket" accesses.

Table 6.5 shows the timing results over ten trials for each of these methods. We compared the times needed to insert and retrieve eight ints from these structures, in programs compiled with optimization level 2. In all cases, the Scratchpad Accelerator shows speedup. Insertion into linked lists takes a long time because of memory allocation costs. Yet even for the immediately–accessed array list, insertion is slower than for the Scratchpad Accelerator, because multiple accesses need to be made to store the key and value. Retrievals for the array list and linked list again take longer than for the Scratchpad Accelerator, because in each case, the lists must be traversed to search for a key. Since keys are used immediately after retrieval in these tests, the scratchpad accesses are likely still suffering from data hazards on loads, sacrificing some speedup. Variance here is significantly smaller for scratchpad insertions than other insertion types, but for retrievals, array lists have the smallest variance. Since array lists are contiguous but linked lists aren't, it makes sense that linked lists would have more opportunities for cache misses and thus a greater variance. Retrieved values for each experiment are stored to a local array in the regular memory hierarchy, so the greater variance in scratchpad retrievals over insertions is likely due to

this additional memory access.

This performance gain is on a very small scale, with a list of only eight key–value pairs. How does the Scratchpad Accelerator perform when used as the base of an actual hash table? We run a similar experiment to measure this, comparing a Scratchpad Accelerator hash table with the C++ standard library's `unordered_map` implementation, as well as our own software implementation. The scratchpad hash table uses a simple hashing function to map keys to buckets, which in this case are scratchpad sets. The bucket is then accessed, either trying to place a key during insertion or find a key during retrieval. If this is unsuccessful, the next bucket is accessed, and so on. Thus the scratchpad hash table uses linear probing at the scale of buckets, but groups key–value pairs for faster lookups within a bucket. A bucket–based hash table is also used for `unordered_map`, but it uses chaining to resolve collisions in these buckets, so they may start smaller or grow larger than eight elements. Finally, our custom hash table implementation uses open addressing with the same hashing function as the scratchpad's table. The custom table stores all pairs in an array, and if a key hashes to a pair which is accessed unsuccessfully, the array is probed linearly for a successful access.

	Insert			**Retrieve**		
	Scratchpad	unordered_map	Custom	Scratchpad	unordered_map	Custom
Average	8993	42351	17417	10860	10744	18580
Median	8996	41226	17364	10862	10730	18581
HW Speedup		**4.583**	**1.930**		0.98784	**1.7106**
Variance	21.56	13369000	4653.2	183.43	3148.44	443.73
Average	7912	41872	19188	8745	11365	20654
Median	7913	41642	19175	8744	11367	20656
HW Speedup		**5.262**	**2.423**		**1.300**	**2.362**
Variance	105.0	242190	512.10	350.5	2151.4	304.10
Average	7972	40832	17589	8932	10454	18849
Median	7970	40685	17562	8920	10481	18848
HW Speedup		**5.105**	**2.204**		**1.175**	**2.113**
Variance	76.10	259320	2378.1	868.4	5844.0	557.14

Table 6.6: Cycles and speedup for inserting and retrieving values from three different hash table implementations. The first uses Scratchpad Accelerator associativity, the second uses the C++ `unordered_map` library, and the third is a custom-built linear probing hash table implementation in C. 10 trials over three different randomly-generated datasets were recorded, accessing 256 integers in a table of size 256.

To run these experiments, we first ensured that each hash table was prepared to handle the amount of data we would give it, so that memory allocation and load factor did not factor into performance results. Our Scratchpad Accelerator is configured with 128 lines of storage, so for each hash table implementation, we required them to fit 128 key–value pairs. Our custom hash table was built upon an array of 128 pairs, and we reserved 128 `unordered_map` pairs ahead of time. We measured the cycles needed for insertions and retrievals: one test to measure the time needed to fully fill the scratchpad, and another to measure the time needed to retrieve each element in the

same order as they were placed. We ran each experiment ten times, discarding the maximum time to avoid large outliers caused by context switches. We repeated this process for three different sets of random integer keys. These experiments were all compiled at optimization level 2. Table 6.6 shows the results.

Except for one `unordered_map` retrieval experiment, the Scratchpad Accelerator's hash table ran faster than its counterparts. The Scratchpad Accelerator was pretty consistent in speedup between insertions and retrievals when compared to the custom hash table implementation. It was much less balanced when compared to `unordered_map`, with much more speedup on insertions and much less on retrievals. `Unordered_map` is trading insertion complexity for retrieval ease, whereas the custom hash table keeps both simple. The scratchpad retrieval experiments also involve a lot of hash table accesses, so data hazards on scratchpad loads could be decreasing speedup there. Insertions don't involve Scratchpad Accelerator loads, so performance there is not impacted by the RoCC response system.

6.3 Memory Shadowing

One of our goals for the Scratchpad Acceleratorwas to enable safe memory shadowing for security, safety, or program analysis. Our final experiments measure the feasibility of the scratchpad for these applications.

Region Isolation

Our first experiment adapts a *SHA256* cryptography algorithm, storing its primary arrays in the scratchpad. The round constants and message schedule are each stored in the scratchpad, but in different regions. This allows us to test region isolation, verifying that accesses to one region do not affect accesses to another. We created two variations of the program to demonstrate this. The first shadows the original data structures, using their local–stack–based addresses to store and load the arrays from the scratchpad. Since the arrays are stored contiguously in the local stack, their addresses cannot collide in our scratchpad in ways that would overfill any of its sets. Because we store each array in a separate region, however, their addresses should certainly not be colliding, because addresses should only map within the current region. We test this by making the addresses of the two arrays overlap, so that accesses will conflict if region isolation is not set up properly.

After compiling both programs at optimization level 0, we compared their hashes of several test messages. This verified that both programs operated correctly and consistently. Data stored in different Scratchpad Accelerator regions, even if accessed using the same addresses, does not affect the operation or communication to other regions. This means there is no easy way for processes that are restricted to accessing different regions of the scratchpad to view the data held by other processes.

Stack Shadowing

With isolation verified, our final experiment tested whether the scratchpad could act as a full shadow memory for a C program by supporting its call stack. We started with a C function that implements the *Euclidean Algorithm* for calculating the greatest common divisor between two integers. After compiling this into assembly, we replaced every memory access in the program with a corresponding Scratchpad Accelerator access. Because this included the function's prologue and epilogue, it meant that the recursive function's entire call frame would be placed into the scratchpad rather than the normal memory hierarchy. This means that from the first call to this function, the call stack would grow from a point in the scratchpad, rather than the regular memory hierarchy. On a return from that function, the address of the top of the stack in the memory hierarchy would be restored to the stack pointer. Then outer functions would access the call stack using regular memory calls, restoring normal functionality.

Trial	Shadowed	Software
1	852	527
2	840	542
3	843	539
4	848	532
5	846	534
6	850	544
7	849	539
8	846	536
9	843	532
10	846	533
Median	846	535
Speedup		0.632
Variance	11.8	24.3

Table 6.7: Cycles needed to run a Euclidean Algorithm program for finding the greatest common divisor of 10 and 2. The software version of the program is compared with a completely–scratchpad hosted shadow version.

Migrating the call stack proved successful, and we were able to measure its impact on runtime performance in Table 6.7 for finding the GCD of 10 and 2. Like with previous experiments, this program often involves data hazards following scratchpad loads, and this hurt its performance. However, the Euclidean algorithm was able to operate successfully without using the regular memory hierarchy at all. This shows the feasibility of moving sensitive programs to the scratchpad, where their data is kept entirely out of the regular memory hierarchy until it is ready to be committed. Because our scratchpad is configured with 1 kilobyte of space, we could only perform this transfer for programs with small call stacks. With a larger scratchpad, of 16 or 32 kilobytes, a much greater array of programs could be moved. Another solution would be to only place key functions in the scratchpad, deciding at compile–time which functions need to have data secured at this higher level.

6.4 Summary

Overall, these experiments affirm that the design of the Scratchpad Accelerator enables useful memory models for developers. Scratchpad accesses are consistent, and their uniform access times provide a stability that some applications may find useful. The device is secure and flexible, to the point that it can support a program's call–stack or multiple shadowed memory locations without issue. However, our experiments also show that our implementation of the accelerator within the RoCC system does not enable fast enough communication for acceleration to occur in most applications. Instead, the heightened acceleration provided by our exposed associative memory structure allowed key–value store applications to speed up. In this implementation of the Scratchpad Accelerator, the goals of supporting key–value storage, data isolation, and memory shadowing found the most success. With modifications to the RoCC communication pipeline, the runtime performance of these applications would become more desirable in comparison with the regular system it seeks to supplement.

Chapter 7

Conclusion

The Scratchpad Accelerator described in this work provides an important first step to exploring the general–purpose utility of scratchpads as a supplement to the regular memory hierarchy in RISC systems. Over the process of development and evaluation for this accelerator, we noted several intriguing directions to take this work for future research.

7.1 Future Work

There are both pragmatic and exploratory options for investigating the impact of the Scratchpad Accelerator further.

Pipeline Integration

The most pressing question we have for the Scratchpad Accelerator is whether its performance for most data structures would become more competitive if the accelerator used a different communication framework with the core. As noted in Chapter 6, data hazards during RoCC instruction handling cause delays of up to six extra cycles for an instruction that is implemented to take one cycle, a drastic limitation to accelerator performance that could be eliminated. Loads are a frequent operation for our accelerator, and the memory loads they replace can take a minimum of one cycle to run, so this severely limits the chance that our accelerator can accelerate an application. A program would need a very high number of cache misses, or use the highly–accelerated key–value store supported by our accelerator, to find performance gains here. Modifying the accelerator's communication framework to eliminate this extra delay on hazards would take a non–trivial amount of modification to the Rocket Chip core's design. However, we have modified other regions of the RoCC system in prior work, and have already identified the modules causing this additional delay. To truly compare a scratchpad memory alternative to the L1 cache, this modification is necessary. Without it, data accesses to the two memory systems are hard to compare, and the performance of load operations reflects more about the RoCC communication channel than the design of the scratchpad.

Security Verification

Beyond changes to the accelerator's placement, we would like to subject the Scratchpad Accelerator to further security testing by implementing it in a fully secured environment. While the hardware infrastructure is already in place, we did not set up OS support for Scratchpad Accelerator securing during context switches, so we were not able to use the securely–configured Scratchpad Accelerator in a running system. With proper communication of the PID through the OS, it would be informative to subject the Scratchpad Accelerator to a rigorous security analysis, identifying what information may leak across applications. There are fewer opportunities for data leakage in the scratchpad as opposed to the larger memory system because of the number of resources involved in each. However, our design may have unforeseen security flaws or impacts. Since we've already verified that the basics of data protection and isolation work in the Scratchpad Accelerator, this research would delve deeper to detect potential covert channels.

Alternative Memory Models

We think the most intriguing research arising from this work will come from exploring the use of the Scratchpad Accelerator from a software developer's perspective. This accelerator is exciting because it is so general–purpose. It offers multiple ways for developers to conceptualize memory use, and this is a key factor in software design. Chapters 5 and 6 investigate some of these memory models, like using the scratchpad to store arrays, shadow memory, and support key–value stores. However, the scratchpad's flexible addressing and stripe–based storage system can enable even more ways of accessing the scratchpad for data storage. The memory models that arise may be more intuitive to developers or provide better benefits to data structures than the regular models we have grown used to. Additionally, because the scratchpad's metadata is freed from supporting cache coherence, it has a lot of room to be modified to support novel uses of hardware memory. We explored one of these uses when implementing key–value store buckets atop the scratchpad's sets. Constructing other data structures atop the scratchpad could benefit them similarly.

Further Fine–Grained Memory Control

Novel instructions could help extend the Scratchpad Accelerator's computational capabilities as well. As explained in Chapter 4, our ISA has plenty of room for expansion. The `opcode` bits of Scratchpad Accelerator instructions can easily fit more types without extension, but we additionally have room to add more opcode bits in the future. Future iterations of the Scratchpad Accelerator could capitalize on this to enable better support for popular software idioms or basic data structure operations. To start, adding informational instructions could help with data structure design. Being able to check how full a stripe of the scratchpad is could help with balancing it as it supports data structures. Other instructions could expose more of the powerful structure underlying the scratchpad's memory, opening them to programmer use. Being able to recover the tags stored in a scratchpad line could help accelerate further types of key–value store operations. Finally, we could add hardware support for running safe and useful data transformations within the scratchpad itself.

In running our experiments, certain assembly patterns for memory accesses frequently reappeared after compilation. These compiler idioms often arose in local stack frame management and data structure access, two domains that the Scratchpad Accelerator supports. By adding fast, single–cycle instructions that collapse popular idioms into hardware operations, the Scratchpad Accelerator could provide direct hardware support for compilers and their writers.

7.2 Final Remarks

Today, our computers are increasingly augmented with special–purpose accelerators and coprocessors for the applications we rely on most. Their circuitry prioritizes certain computational styles because we use them frequently and they aren't served perfectly by the regular CPU. General–purpose computers should start providing the same level of flexibility in supporting the data used in software. After all, data accesses define a great part of the code, functioning, and performance of a program. The protected, associative RISC scratchpad that we propose gives software an important mastery over its memory use when needed. Developers can use memory according to their priorities, emphasizing the performance, security, and access methods of key data structures. In its evaluation, we verified that the Scratchpad Accelerator provides uniform memory access and software control over data's location. This increases the predictability and stability of a program's performance. The Scratchpad Accelerator also provides data protection, keeping data secured in its designated scratchpad region. The invisibility of the scratchpad to the regular memory hierarchy provides specialized benefits: code can run without using caches for data storage, monitoring the state of the regular memory hierarchy without affecting it. Although we found that the Scratchpad Accelerator's performance suffers during loads, it still aids software in other ways. The associative structure of the scratchpad helps to accelerate lookups, opening the door to intriguing data structure optimizations. Likewise, multiple methods of accessing data in the Scratchpad Accelerator serve to provide intuitive support for many memory models. With the control provided by the Scratchpad Accelerator, memory use can be customized to fit a program's needs.

The World Today: May 23, 2022

What meaning does a work have without context? The last few years have been tumultuous indeed, hitting us with unwelcome surprises while long–lasting trends carried on with their own business, seemingly oblivious. Luckily, we've also seen impressive innovation and delightful moments, even in the midst of a global pandemic. This glimpse of life, technology, and culture in 2022 helps anchor my thesis to the fleeting moment that reared it.

This time, like all times, is a very good one, if we but know what to do with it.

— Ralph Waldo Emerson

Cost of a Gallon of Milk $4.29 at Stop and Shop, Williamstown

Cost of a Gallon of Oat Milk $9.98 at Stop and Shop, Williamstown

COVID Cases at Williams 859 student positives since August 2020

World Population 7,982,973,977

US Population 334,585,863

Food on Spring Street 8 restaurants, 5 cafes/dessert shops

Cost per Gigabyte of a SSD 12 cents

Best Recent Movie Unbiasedly, *Everything Everywhere All at Once*

GIF Pronunciation gif

Williams Tuition Full cost, including room and board, is $74,660 for 2021-2022

Williamstown Weather Average low of 14°F in January and average high of 82°F in July

Size of the Williams Climbing Wall 2000 square feet

Bibliography

[1] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. The Rocket Chip Generator. Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 4 2016.

[2] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012* (June 2012), pp. 1212–1221.

[3] Burbage, M. A hardware engine for generating number-theoretic sequences. In *Grace Hopper Celebration (GHC 20)* (Virtual, 2020).

[4] Burbage, M. Hardware acceleration for search algorithms. In *ACM Student Research Competition: Grand Finals* (2021).

[5] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2011), FPGA '11, Association for Computing Machinery, p. 33–36.

[6] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1999), PLDI '99, Association for Computing Machinery, p. 1–12.

[7] Chipyard. Documentation: Memory Hierarchy. `https://chipyard.readthedocs.io/en/dev/Customization/Memory-Hierarchy.html#the-l1-caches`.

[8] Cyll, T. Cache-Conscious Dynamic Memory Allocation. Honors Thesis, Williams College, May 2004.

[9] Digilent. *Nexys A7 FPGA Board Reference Manual*. Pullman, WA, USA, 2019.

[10] Efnusheva, D., Cholakoska, A., and Tentov, A. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Information Technology and Computer Science* (04 2017), 151.

[11] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Elsevier Inc., Waltham, MA, USA, 2012.

[12] Hölzle, U., Chambers, C., and Ungar, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 1991), ECOOP '91, Springer-Verlag, p. 21–38.

[13] Kang, M. SOAR: a Self-Optimizing Adaptive SoC on FPGAs. Honors Thesis, Williams College, May 2020.

[14] Kannan, A., Shrivastava, A., Pabalkar, A., and Lee, J.-e. A software solution for dynamic stack management on scratch pad memory. In *2009 Asia and South Pacific Design Automation Conference* (2009), pp. 612–617.

[15] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).

[16] Liptay, J. S. Structural aspects of the System/360 Model 85, II: The cache. *IBM Systems Journal 7*, 1 (1968), 15–21.

[17] lowRISC. Documentation. `https://www.lowrisc.org/docs/`.

[18] Mahapatra, N. R., and Venkatrao, B. The processor-memory bottleneck: Problems and solutions. *XRDS 5*, 3es (Apr. 1999), 2–es.

[19] Pala, D. Design and Programming of a Coprocessor for a RISC-V Architecture. Master's Thesis, Politecnico di Torino, December 2017.

[20] Savage, J., and Jones, T. M. HALO: Post-link heap-layout optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (New York, NY, USA, 2020), CGO 2020, Association for Computing Machinery, p. 94–106.

[21] Singh, A., Dave, S., Zardoshti, P., Brotzman, R., Zhang, C., Guo, X., Shrivastava, A., Tan, G., and Spear, M. SPX64: A scratchpad memory for general-purpose microprocessors. *ACM Trans. Archit. Code Optim. 18*, 1 (Dec. 2021).

[22] Syrowik, B. A., Fort, B., and Brown, S. D. Use of CPU performance counters for accelerator selection in HLS-generated CPU-accelerator systems. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (New York, NY, USA, 2018), HEART 2018, Association for Computing Machinery.

[23] Vogt, M., Hempel, G., Castrillon, J., and Hochberger, C. GCC-plugin for automated accelerator generation and integration on hybrid FPGA-SoCs. In *Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP 2015)* (2015).

[24] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, 5 2014.

[25] Weaver, D., Germond, T., and International, S. *The SPARC Architecture Manual: Version 9*. PTR Prentice Hall, 1994.

[26] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), pp. 428–441.

[27] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C., and Torrellas, J. Correction: InvisiSpec: Making speculative execution invisible in the cache hierarchy. Tech. rep., University of Illinois at Urbana-Champaign, 2019.

[28] Zacharopoulos, G., Ferretti, L., Ansaloni, G., Di Guglielmo, G., Carloni, L., and Pozzi, L. Compiler-assisted selection of hardware acceleration candidates from application source code. In *2019 IEEE 37th International Conference on Computer Design (ICCD)* (2019), pp. 129–137.

Appendix A

Glossary of Terms

ASIC or an application–specific integrated circuit, is a hardware device that has been designed for a single use. This allows it to be better for that application than a general–purpose computer would be. When we say a circuit is "in silicon" it is typically an ASIC device.

BRAM or block RAM, is an FPGA component used to store large amounts of data efficiently within the FPGA itself. For us, large memory constructs in Chisel and Verilog are inferred to use BRAM when the Verilog design gets processed by Vivado. On–chip caches thus likely use BRAM in FPGAs when they would use SRAM in silicon implementations.

Chisel is an open-source domain-specific language for designing hardware. It is embedded in Scala, allowing hardware designers to use high-level abstractions for describing circuit generators. These circuit generators are all parameterized, letting developers reuse and adapt hardware components easily. Elaboration creates Verilog circuit designs out of these descriptions, allowing designs to be simulated, mapped to FPGAs, or placed in silicon.

CISC or complex instruction set computer, is a contrasting computer architecture to RISC. CISC instructions can perform a lot of computation following a single instruction. This means code can be written using few instructions, but some of them may be wide and have varying latencies.

CPU or central processing unit, is the part of a computer that executes a program's instructions. Many CPUs process instructions according to a pipeline in order to increase efficiency, passing instructions through several stages so that each stage works on an instruction at the same time. On RISC machines, these stages might involve fetching a new instruction from memory, decoding the instruction to see what should happen, executing the computation demanded by the instruction, accessing memory if needed, and writing back any results to registers.

CSR or control and status register, is a special register used inside RISC–V CPUs to store important microarchitecture state. They might include information about timing, hardware traps, or protection levels.

DRAM or dynamic random access memory, is a type of RAM that stores bits on capacitors, and is often used to support a computer's main memory. Another common type of RAM is SRAM, which stores bits in feedback loops. DRAM is slower but less expensive than SRAM, so it is better suited for storing large amounts of data.

EX or execute, is the third stage in a classic RISC pipeline. Instructions that involve computations, like arithmetic operations, do their work in this stage.

FPGA or field programmable gate array, is a hardware device that can be programmed to behave like different pieces of hardware. This lets it support reconfigurable computing, allowing for testing hardware circuits during their development or supporting a hardware device that evolves over its use. Designs are limited by the number of components in a given FPGA, which include Flip–Flops, LUTs, routing units, and specialized hardware like ALUs and BRAMs.

HDL or hardware description language, is used to design circuits through describing the timing and layout of a hardware design. The most common HDLs for digital circuit design are Verilog and VHDL. Other popular HDL approaches include domain–specific functional languages like Chisel. HDL designs are useful because they can be simulated or used to produce hardware like ASICs or FPGA configurations.

HLS or high–level synthesis, is a process that can take an abstract, high–level design of a piece of hardware (typically in a language like C) and produce a detailed HDL description of the corresponding circuit. This helps raise the abstraction level for hardware designs.

ID or instruction decode, is the second stage in a classic RISC pipeline. Now instructions are decomposed into important information, like the addresses of the registers that they will operate on. Source register values get fetched here, but if the pipeline is not ready to execute this instruction yet, it will stall here instead, stopping IF and ID from obtaining new instructions.

IF or instruction fetch, is the first stage in a classic RISC pipeline. Here, the next instruction is fetched from memory for processing in the pipeline.

IP or intellectual property, is a (possibly, patented or proprietary) piece of technology made available for general reuse. IP cores are common building blocks for HDL designs that represent reusable pieces of hardware, similar to a library used in software development.

ISA or instruction set architecture, defines how a computer should be instructed by providing an abstract model of programming. ISAs provide the set of instructions a computer understands along with important architectural features like how memory is modeled.

L1 or level 1, is a type of CPU cache that lies closest to the processor when the computer has multiple caches. In this type of memory hierarchy, larger caches can store more data, but also must lie further from the processor, so data retrieval takes longer. The L1 cache is thus the smallest and fastest cache. Then the processor will try retrieving data from it first before falling back to slower cache levels. For efficiency, many modern L1 caches are additionally divided into an instruction cache (for running code) and data cache (for everything else).

lowRISC is a RISC–V SoC built at Cambridge. It adapts the Rocket Chip's designs for implementation on the Nexys A7 FPGA development board.

LUT or lookup table, stores precalculated values in an array so that runtime calculations can be changed into fast array lookups. Hardware versions of these are used extensively in FPGAs to enable their reconfigurability. Each LUT in an FPGA can be programmed to act like a Boolean logic function, making them a key building block for logical circuits.

MEM or memory, is the fourth stage in a classic RISC pipeline. Here, instructions that involve memory access communicate with the memory hierarchy, since accesses take at least one cycle to complete.

PID or process identifier, is a unique number for a running process which helps a computer's OS to manage processes and their protections.

RAM or random access memory, is a type of storage where data reads and writes take about the same amount of time regardless of access order and data location.

RISC or reduced instruction set computer, uses an ISA with constant–length instructions that each perform simple operations. This is in contrast to CISC architectures which may use fewer, but longer, more complicated instructions.

RISC–V pronounced "risk–five," is an open standard fifth generation ISA developed at UC Berkeley. Perhaps unsurprisingly, it is a RISC ISA.

RoCC or Rocket custom coprocessor interface, provides a way to extend Berkeley's Rocket Chip SoC with custom accelerators. The interface allows accelerators to receive designated custom instructions from the core, access the memory hierarchy, and interrupt the core, among other connections. Instructions can have two source registers and may take a register of response, but computations can also be more decoupled. Additionally, one Rocket core can have multiple RoCC accelerators.

Rocket Chip is an open-source SoC designed at UC Berkeley. It is written in Chisel so that it can be elaborated into Verilog circuitry and placed in silicon, simulated, or mapped to an FPGA. The design is parameterized, so the same design can be used to produce variations on the Rocket Chip with different numbers of cores, component sizes, and varying extensions.

SoC or system on a chip, implements an entire computer on one chip.

SRAM or static random access memory, is a type of RAM often used for memory storage within CPUs, like caches. SRAM is faster but more expensive than DRAM.

WB or writeback, is the last stage of the classic RISC pipeline. Results from the EX and MEM stages are finally written into the register file in this stage, completing an instruction's operation. In the Rocket Chip, this stage is when a custom instruction gets shared with the RoCC interface for communication with accelerators.

Appendix B

At–a–Glance Light Debugger

The At–a–Glance Light Debugger, designed in previous research, proved very helpful for developing the Scratchpad Accelerator. We modified lowRISC so that a RoCC accelerator could send signals through the RoCC system that would control the Nexys A7's 7–segment display [4]. To make display control painless, we created a Module (a Chisel hardware block) called `DisplayDriver`. This module handles the specifics of converting binary or hexadecimal data into readable values on the display. All that is needed is input: the display needs to be told how many of the display's digits should be used as digits, and how many should be broken into their component display segments to efficiently relay binary values. For the digits, the driver needs to be told what encoding to use for data: for us, we use hexadecimal values from 0-F. Finally, the accelerator sends the actual hexadecimal and binary data to `DisplayDriver`.

To help with debugging, the Scratchpad Accelerator collects important state data continuously and during notable events. Both are displayed, so if events happen infrequently enough, like when a program is run step–by–step during debugging, developers can read what is happening within the hardware as it runs. Tables B.1 and B.2 explain the Chisel wires captured by the display. In Table B.1, we explain the the data shared in 24 segments that capture single–bit state values. Wires prefixed with 'got_' are only captured when the accelerator is in the `work` stage, where an instruction has been decoded and is being handled. These help identify the state of the `Metadata` and `Data` blocks. Wires prefixed with 'has' display events that have happened since the last RoCC instruction was received. These help identify the effects of an instruction on the long–term state of the accelerator. Finally, the wires prefixed 'io.' are the accelerator's actual input/output wires which connect to the Rocket Core. These are updated continuously and show the state of the accelerator within the broader system. Figure B.2 shows how the display looks after an erring Scratchpad Accelerator instruction has been processed.

Binary data only uses three of the eight available display digits. The final five digits display larger values: each digit is capable of representing values from 0-F in hexadecimal. Table B.2 explains what these digits display and when their values are captured. This level of detail helps verify that the accelerator is interpreting and handling an instruction correctly.

The At–a–Glance Light Debugger was most helpful during the implementation and testing

	Digit		
Segment	1	2	3
A	got_allValid	hasErred	io.resp.ready
B	got_EmptyFound	hasResped	io.resp.valid
C	got_TagFound	hasSend2	io.interrupt
D	got_CanAccess	hasSend1	hasGoneToErr
E	got_Clear	hasSend0	got_StoreConditional
F	got_Write	hasSent2	got_Read
G	got_CanExpose	hasSent1	got_StillLoaded
DP	got_ClearRegion	hasSent0	got_LoadLink

Table B.1: Binary Scratchpad Accelerator signals captured by the debugger, used for debugging the hardware during development.

	Digit		
	4	5	6
Data	Error Code	State	Scratchpad Data Read
Wire Names	`newError.asUInt`	`state`, `work`	`datapad.io.readData(3,0)`
Capturing Event	On a scratchpad error	Every cycle	When sent in a response
	7		8
Data	Instruction Type		Scratchpad Data Read
Wire Names	`offsetMode`, `usingDest` `usingFirst`, `usingSecond`		`datapad.io.readData(3,0)`
Capturing Event	During decoding		Before sent in a response

Table B.2: Numeric values captured by the debugger and displayed in hexadecimal.

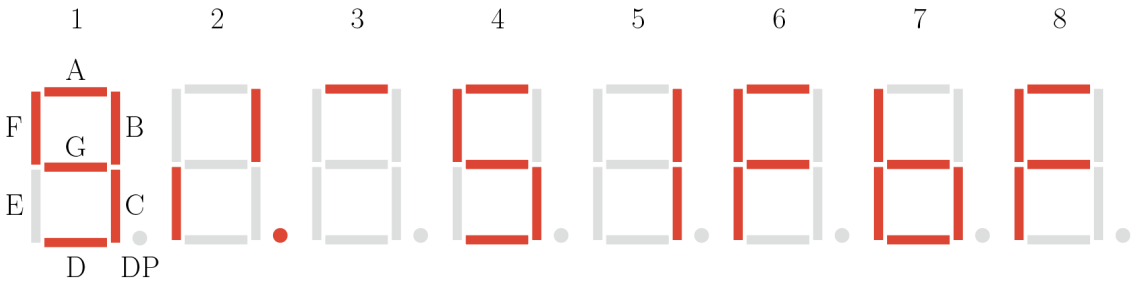

Figure B.1: Potential state of the display after a successful `Put` instruction.

Figure B.2: State of the display after an erring Scratchpad Accelerator instruction.

stages of Scratchpad Accelerator development. On secured accelerator configurations, the display should be turned off. However, during software development that targeted the Scratchpad Accelerator, we found the lights very useful for the testing of software itself. Beyond showing error types, the display shows useful information on the state of the accelerator that helps unmask why an error occurred. This was useful to us in developing accelerated code, because our OS does not handle error signals from the Scratchpad Accelerator. In developing low–level code for the accelerator, like libraries of scratchpad–based data structures, OS signal support, or custom high–performance programs, code complexity increases. Developing and testing low–level code can be very slow, but the selection of useful information provided by the debugging display helps ease that process for programs interfacing with the Scratchpad Accelerator.