UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de Formació Interdisciplinària Superior

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Grau en Matemàtiques
Grau en Enginyeria Informàtica

TFG - Treball de Fi de Grau

Hardware support for fine-grained parallelism and simultaneous multithreading in RISC-V

Author - Pau Recort Bascuas
Advisor - Arvind
Advisor - Miquel Moretó Planas

May 2023

Abstract

Modern processors must exploit all available levels of parallelism in order to achieve maximum performance and hardware resource usage. This work focuses on multithreaded architectures, which exploit thread-level parallelism by hosting multiple instruction streams in a single core with shared resources. Multi-threaded architectures, however, inherently require plenty of thread-level parallelism to execute efficiently. We propose using the P-RISC execution model in order to extract parallelism from the code and saturate the hardware resources. The P-RISC execution model adds custom instructions and hardware support that allow the creation and synchronization of parallel tasks with very small overhead. In this work, we design and implement a multithreaded RISC-V core capable of executing the P-RISC model and added instructions. We implement a split-phase memory system and extend it to accommodate a caching system that reduces the number of actual split-phase accesses. We implement a custom arbiter module with a speculation-level-based priority policy that maximizes the valid instructions scheduled into the execution units. Our implementation can scale up to hosting 12 threads per core, has been tested in both simulation and on FPGA, and can be synthesized at 1.1GHz using GlobalFoundries 22nm technology. It achieves 1.4× to 10× speedups with respect to the sequential equivalent on FPGA.

Keywords: Microarchitecture, Execution model, P-RISC, Simultaneous multithreading, Split-phase memory, Bluespec SystemVerilog, RISC-V.

AMS Subject Classification: 68W10 68W35

Resum

El processadors moderns han d'explotar tots els nivells de paral·lelisme disponibles per a aconseguir el màxim rendiment i ús dels recursos hardware. Aquest projecte es centra en les arquitectures multifil, que executen múltiples fils de codi simultàniament per a saturar recursos hardware compartits. Les arquitectures multifil, però, requereixen de forma natural d'una alta quantitat de paral·lelisme per a executar eficientment. Utilitzarem el model d'execució P-RISC per a extreure paral·lelisme del codi i saturar els recursos d'execució hardware. El model P-RISC afegeix instruccions pròpies i el suport hardware necessaris per a permetre la creació i sincronització de tasques paral·leles amb un overhead molt baix. En aquest projecte, dissenyem i implementem un nucli multifil RISC-V capaç d'executar el model i les instruccions afegides per P-RISC. Implementem un sistema de memòria amb accés en dues fases i l'estenem amb un sistema de cache que redueix el nombre d'accessos en dues fases. Implementem un àrbitre que maximitza el nombre d'instruccions vàlides enviades a les unitats d'execució mitjançant una política de prioritat basada en el nivell d'especulació. La nostra implementació es pot escalar fins a executar 12 fils fer nucli, ha sigut provada en simulació i FPGA, i es pot sintetitzar a 1.1GHz amb tecnologia GlobalFoundries de 22nm. Aconseguim increments en rendiment de 1.4× fins a 10× en comparació a l'equivalent seqüencial en FPGA.

Paraules clau: Microarquitectura, models d'execució, P-RISC, multifil simultani, memòria en dues fases, Bluespec SystemVerilog, RISC-V.

Resumen

Los procesadores modernos deben explotar todos los niveles de paralelismo disponibles para conseguir el máximo rendimiento i uso de los recursos hardware. Es proyecto se centra en las arquitecturas multihilo, que ejecutan múltiplos hilos de código simultáneamente para saturar los recursos hardware compartidos. Las arquitecturas multihilo, sin embargo, necesitan de maneral natural de una alta cantidad de paralelismo para ejecutar eficientemente. Utilizaremos el modelo de ejecución P-RISC para extraer paralelismo del código i saturar los recursos de ejecución hardware. El modelo P-RISC añade instrucciones propias y el soporte hardware necesarios para permitir la creación y sincronización de tareas paralelas con un overhead muy bajo. En este proyecto, diseñamos e implementamos un núcleo multihilo RISC-V capaz de ejecutar el modelo y las instrucciones añadidas por P-RISC. Implementamos un sistema de memoria con acceso en dos fases y lo extendemos para añadir un sistema de cache que reduce el número de accesos en dos fases. Implementamos un árbitro que maximiza el número de instrucciones válidas que se envían a las unidades de ejecución mediante una política de prioridad basada en el nivel de especulación. Nuestra implementación se puede escalar hasta ejecutar 12 hilos por núcleo, ha sido provada en simulación y FPGA, y puede ser sintetizada a 1.1GHz con tecnologia GlobalFoundries de 22nm. Conseguimos incrementos en el rendimiento de 1.4× hasta 10× con respecto al equivalente secuencial en FPGA.

Palabras clave: Microarquitectura, modelos de ejecución, P-RISC, multihilo simultaneo, memoria en dos fases, Bluespec SystemVerilog, RISC-V.

Acknowledgements

I would like to express my most sincere gratitude to Prof Arvind for hosting me in his group at CSAIL-MIT, for giving me the opportunity to conduct research on a topic that has been of the utmost interest to me, and for keeping collaborating with me after my return to Barcelona. I would also like to thank my colleagues at Arvind's group Thomas Bourgeat, Ana Arduengo, Xuhao Chen, Tianhao Huang, Jiazheng Liu, and Chanwoo Chung for helping me with my project and making the most out of my stay. I hope to work with them again in the future.

I would also like to thank Miquel Moretó and Max Doblas at BSC-CNS for the constant interest that they have shown in my project, for helping me countless times, and for providing feedback on my work. My collaboration with them has helped me grow my interest in research.

This project would have not been possible without the help of CFIS both during my stay and previously during my bachelor's degrees. Special thanks to Fundació Mir-Puig, the Barcelona Supercomputing Center, and Generalitat de Catalunya for the financial support.

Contents

Abstract i

Resum iii

Resumen v

Acknowledgements vii

1 Introduction 1

2 Related work 3

3 P-RISC Execution model 5
- 3.1 Abstract execution model . . . 5
- 3.2 Architectural execution model . . . 6
- 3.3 Algorithms in P-RISC . . . 7
  - 3.3.1 Mergesort . . . 7
  - 3.3.2 Matrix multiplication . . . 8

4 Core microarchitecture and design 11
- 4.1 Split-phase memory subsystem . . . 11
- 4.2 Narrowing pipeline . . . 14
- 4.3 L1I - Instruction fetch and decode . . . 17
- 4.4 Arbiter - Instruction issue . . . 19
  - 4.4.1 Naive design . . . 20
  - 4.4.2 Sorting network design . . . 21
  - 4.4.3 Speculation level and priority . . . 22
  - 4.4.4 Multi-cycle design . . . 23
- 4.5 L1D - Memory access instructions . . . 24
- 4.6 Harts and physical mapping of the architectural state . . . 26
- 4.7 Control-flow, redirections, and epochs . . . 27
- 4.8 NTTX - Stream fork and scheduling . . . 28
- 4.9 Full core overview . . . 30

5 System microarchitecture and design 33
- 5.1 L2S - Second level cache . . . 33
- 5.2 MTQ - Global pool of continuations . . . 34
- 5.3 Multi-core systems . . . 36

6 Testbench, synthesis tools and hardware verification 39
6.1 Bluespec SystemVerilog . . . 39
6.2 Testbench environments . . . 40
6.2.1 RTL simulation . . . 42
6.2.2 FPGA synthesis . . . 44
6.2.3 ASIC synthesis . . . 45
6.3 Testbench profiles . . . 48
6.3.1 Tandem verification . . . 48
6.3.2 Cycle-accurate pipeline visualization . . . 50
6.3.3 Event counters . . . 52
6.3.4 Memory subsystem testing . . . 53

7 Software stack and benchmarks 57
7.1 C ABI and ISA extension . . . 57
7.2 C API and libraries . . . 62
7.3 Software benchmarks . . . 64
7.3.1 Mandelbrot . . . 65
7.3.2 Matrix multiply . . . 65
7.3.3 SAXPY . . . 66
7.3.4 mergesort . . . 66
7.3.5 quicksort . . . 67
7.3.6 heapsort . . . 68

8 Evaluation 71
8.1 Timing and area in ASIC . . . 71
8.1.1 Arbiter . . . 71
8.1.2 Full core . . . 74
8.2 Software benchmarks . . . 75
8.2.1 Hardware configuration . . . 75
8.2.2 IPC, speedup, and instruction distribution graphs . . . 76
8.2.3 Mandelbrot . . . 78
8.2.4 Matrix multiply . . . 79
8.2.5 SAXPY . . . 80
8.2.6 Mergesort . . . 81
8.2.7 Quicksort . . . 82
8.2.8 Heapsort . . . 83
8.3 Summary . . . 84

9 Conclusions and future work 85
9.1 Conclusions . . . 85
9.2 Future work . . . 86

Bibliography 87

Chapter 1

Introduction

During the last few decades computer architecture research has focused on extracting parallelism at any possible level. At the same time, the improvements in manufacturing processes provide architects with more real state to add new hardware resources to their designs. The broader challenge has been for many years finding a more efficient way of using such resources. One option is improving the microarchitecture of high-performance cores so that they can extract more *Instruction Level Parallelism* (ILP) and provide an overall faster execution of sequential code. The other option is to implement machines that can execute multiple sources of code in parallel in what is known as *Thread Level Parallelism* (TLP). Parallel machines somehow distribute the resources between the different instruction streams.

Parallel processors partition the hardware resources either statically or dynamically. Multi-core processors include several cores. Each core is capable of executing an independent thread. The resources are said to be partitioned statically among threads, as each thread is able to execute using exactly the resources present in the core that hosts them. This can yield inefficient use of resources as different tasks might put lots of pressure on a few specific functional units and leave others unused. Furthermore, long latency operations, like main memory accesses, cause stalls on the requesting thread. If that happens, many resources of that core remain unused.

The answer to this inefficiency is multithreading. In this type of architecture, several threads are hosted in the same core and share the execution resources. The different threads can interleave at different granularity. The finest and more efficient granularity is *simultaneous multithreading* (SMT). Under SMT, all present threads simultaneously issue instructions to and compete for shared resources. This architecture is not necessarily incompatible with a multi-core approach. Most current SMT processors include several SMT cores. Therefore, resources are statically partitioned between cores but dynamically partitioned within each core.

One of the main difficulties of using such machines is extracting enough TLP from the program in order to saturate the resources. Most processors try to extract both TLP and ILP. The techniques used to maximize ILP - such as OoO execution, prefetching, and speculation - can be very resource intensive and therefore limit the number of resources allocated to executing higher levels of TLP. In this work we propose using the P-RISC parallel execution model introduced in [NA89] in order to maximize the TLP. This will reduce the need to extract single-threaded ILP.

The P-RISC execution model extends the base ISA of the system with Fork and Join instructions. Such instructions are used to fork an instruction stream into several independent ones that can be executed in parallel, synchronize them, and merge the results. Most parallel execution models are defined by software and require large overheads. This means they provide TLP efficiently only if the granularity is coarse enough. P-RISC provides primitives that have very little overhead and allows us to extract very fine-grained parallelism from many algorithms. P-RISC also proposes a microarchitecture with a split-phase memory system that prevents the resources from getting blocked by threads waiting on slow in-flight memory requests.

We believe that the P-RISC execution model together with an SMT machine provides a very efficient

execution platform. In order to prove that, our goal is to build a processor that specifically targets TLP. Our core does not provide virtually any level of single-thread ILP and relies on the TLP produced by the P-RISC execution model. We base our design on a very simple in-order pipeline. Instead of allocating resources to extracting ILP, we focus on executing efficiently with a high throughput when plenty of TLP is provided. This problem includes providing a microarchitecture that can halt streams that are waiting on a long latency memory access without blocking any of the execution resources of the core.

The contributions that our work adds to previous SMT and P-RISC research are as follows.

- A RISC-V SMT core that is focused on efficiently executing TLP. It does not allocate any extra resources to improving single-threaded ILP. Almost all resources are dynamically partitioned.
- The core uses a cached split-phase memory system that allows us to stall threads while they wait on long latency memory accesses without blocking any execution resources. This idea from the original P-RISC microarchitecture is extended with a first-level cache that is used as a filter that reduces the pressure put on the split-phase memory system.
- The instructions produced by the different threads are issued into the backend by an arbiter module with a priority policy based on speculation level. This policy minimizes the amount of backend throughput wasted on wrong-path instructions with minimal hardware overhead.
- We implement the Fork and Join instructions proposed by P-RISC in order to obtain fine-grained parallelism with very little overhead. This allows us to obtain plenty of parallelism in many algorithms and obtain an overall high throughput and resource usage.
- We prove that the core implementation provided can scale up to configurations with 12 harts.
- An extension to the Spike RISC-V ISA simulator that can verify P-RISC hardware implementations with Fork and Join instructions.
- A basic C API that includes fork and join primitives with minimal overhead.
- A series of benchmarks implemented using the API and achieving a $1.4\times$ to $10\times$ speedup with respect to the sequential equivalent in our hardware.

The remaining sections of this thesis are organized as follows. Chapter 2 summarizes previous research into SMT computing, the commercial success of such processors, P-RISC, and related models. Section 3 gives further explanations about the P-RISC execution model. Chapter 4 discusses the design and implementation of the SMT core, which is the main hardware device implemented in this work. Chapter 5 specifies the hardware components that the processors should include in order to efficiently make use of the SMT cores and the P-RISC model. Chapter 6 gives a comprehensive overview of the testbench and tools that were used to develop and evaluate the hardware. Chapter 7 discusses the basic software stack provided and how it was used to implement several parallel benchmarks. Section 8 evaluates the results both in terms of the quality of the synthesis of the core and the performance obtained when executing the aforementioned benchmarks. Chapter 9 concludes the thesis and outlines possible further lines of research.

Chapter 2

Related work

Simultaneous multithreading was extensively studied during the 1990s [Lo+97; TEL95]. The improvements in fabrication techniques enabled the design of processors that were more resource intensive. After achieving high single-threaded performance in the previous decades thanks to out-of-order pipelines, caching systems, and many other microarchitectural techniques, research started shifting towards single-chip TLP computation. The immediate option was chip multiprocessors, where a few independent cores could be manufactured in the same die. The main criticism toward chip multiprocessors is the inefficiency of statically partitioning the resources. If the instruction streams rely mostly on different resources - e.g. one executes mostly floating-point arithmetic and another executes mostly integer arithmetic - or one of them stalls due to a long latency operation, some of the resources remain highly unused.

SMT architectures propose a dynamic partition of resources among threads. Several instruction streams are hosted in the same core. Some of the resources are physically or logically separated in order to provide the appearance of execution in isolation - in terms of correctness but not necessarily in terms of performance. However, many of the expensive resources are shared among threads. This could include the L1I and L1D caches and memory pipelines as well as the functional units or rename logic and physical register files. Extensive research has been conducted on the topic of resource usage efficiency under parallel programs in SMT [Lo+97; TEL95]. In [Tul+96] they focus o the practical problem of fetching and issuing instructions from simultaneous streams in order to add SMT capabilities to an existing core with minimal changes to the microarchitecture. In [TB01] they focus on the problem of hiding long latency operations in SMT and freeing the resources that were used by the issuing thread while the request is being resolved. We can also find research focused on the problem of integrating vector units into EV8 SMT cores in Tarantula [Esp+02]. This symbiosis makes sense as vector operations are usually not as latency-sensitive as their scalar counterparts and also can introduce long latency themselves, which SMT architectures are good at hiding.

Simultaneous multithreading has been quite successful in the commercial space as well. Starting in the early 2000s, several companies have offered general-purpose processors with some degree of SMT capabilities for the workstation and server markets [Mar+02; But+11; KST04]. Intel introduced their commercial implementation of multithreading technology in the Pentium IV generation for their Xeon line. This microarchitecture only provided 2-way multithreading and several of the resources were still statically partitioned among threads [Mar+02; TT03]. In a similar time frame, IBM introduced SMT capabilities to their server processor line with the POWER5 [KST04]. Later they refined their microarchitecture until achieving 8-way multithreading in their POWER8 machines[Sin+15]. Later on, AMD also introduced processors with multithreading capabilities to their commercial line starting with the Bulldozer architecture [But+11]. The Alpha EV8 reached the late stages of development with a 4-way multithreading architecture [Eme+99]. Intel and AMD still commercialize processors with 2-way multithreading to date in the desktop and laptop space.

Although SMT has seen commercial success in different lines of products, we observe a few differences

between them and the earlier academic proposals. The most obvious one is the difference in SMT width. Most of the academic work concludes that cores should host in the neighborhood of 8 simultaneous streams. On the commercial side, however, except for the latter IBM proposals, processors usually host only two threads per core. Another noticeable difference is the partition of resources. The academic work indicates that most resources should be partitioned dynamically in order to achieve maximum performance and efficiency [Lo+97]. Having a static partition in certain resources makes for a more stable and predictable behavior - which is more relevant in commercial products - and also reduces the number of access ports required in many components. Even if it limits the maximum performance under optimal conditions, it guarantees better performance in worst-case conditions [TT03]. Most commercial solutions also add single-thread execution modes that suspend one of the contexts and allocate all the resources to the remaining one. Finally, it must be noticed that although they are also expected to be used in multi-core configurations, processors do not necessarily expose to the operating system the difference between physically independent cores and independent contexts of the same core.

A computing parading that is closely related to SMT and has great commercial success is *Single Instruction Multiple Thread* (SIMT). This correlation is especially true if vector instructions are added to the SMT core. Under the SIMT model, and like in vector units, a single computational kernel is executed over separate sets of data. However, the control of instruction paths over each data lane is separated. That is, each data segment is logically treated by an independent instruction stream and can tolerate data-dependent changes in the instruction path. Although functional correctness is mostly guaranteed, it should be noted that good performance and efficient use of resources is obtained only if the control-flow divergence among lanes is limited. SMT is better suited for general-purpose computing and offers a computing parading with much more thread independence, even if it cannot quite offer the efficiency of SIMT in well-suited problems. One of the most noticeable examples was the commercial introduction of CUDA by Nvidia in the Tesla generation and continuation in the latter generations [Lin+08]. This domain-specific language brought general-purpose computation to GPUs. More recently, research has been conducted on the problem of porting the RISC-V ISA family into an open SIMT model [Col17].

In recent years research has also been done in the ordered parallelism field. This paradigm tries to exploit as much parallelism as possible at a task level. It defines the set of parallel tasks as a set of transactions that must be perceived to execute in order. This concept borrows many ideas from transnational memory systems [Moo+06]. The architecture design in [Jef+16] executes whole tasks speculatively as soon as they are discovered, without knowing a priory if their dependencies have been met. If later it is discovered that some dependencies were not met, it rolls back the results from that task and executes it again. A processor microarchitecture implementation was later proposed in [AS20]. This implementation uses the same basic principles but has the major restriction that every object is owned and can only be modified by a specific core - although it can be read by any of them. Tasks must be scheduled into execution units according to their write-set. Despite that limitation, they showed promising performance results.

In this work, we build an SMT core highly focused on efficient TLP. We implement a system that relies on the P-RISC execution model to extract fine-grained parallelism and then efficiently executes the tasks generated in SMT. The previous research done on the topic and exposed here mostly focuses on increasing ILP by simultaneously executing TLP. For the most part, they base their designs on powerful processors that already exploit ILP in single-thread execution and they try to maximize such parallelism by adding TLP. Our core does not exploit any ILP under single-thread execution. Although that is an approach that has been successful in many environments, we try to minimize the hardware resources allocated to improving single-thread ILP - OoO execution, prefetching, branch prediction, etc.- and focus on efficient TLP. Still, we obtain competitive performance under parallel loads, proving the efficiency of the SMT architecture when combined with the fine-grained parallelism of P-RISC. From the original P-RISC research, we mainly borrow the Fork and Join instructions and the split-phase memory system ideas. However, we port the Fork and Join instructions into a modern RISC-V ISA, substitute the recirculating token queue with an SMT architecture, and extend the split-phase memory system to consider caching hierarchies.

Chapter 3

P-RISC Execution model

In this chapter, we focus on discussing the P-RISC parallel execution model. This is the model that we use in our system in order to obtain fine-grained parallelism. We believe that such a model is very well suited for SMT execution. It must be noticed however that P-RISC is not necessarily tied to our specific hardware implementation nor to SMT in general. In the following section, we start by providing an abstract generalization of the execution model. This is a theoretical model that could be translated into other specific architectures or ISAs. After that, we define the precise implementation of the model that we use in our machine. Finally, we provide a couple of problems that exemplify the parallelization techniques used in this model.

3.1 Abstract execution model

The P-RISC execution model is based on an object that, at an abstract level, is much smaller than a task or a thread. The fundamental element is a *continuation.* A continuation is an object that defines a small action or computation that must be performed [NA89]. This includes some element, typically a program counter, that indicates what action must be performed. It also includes a certain hardware state, a data collection that might be used when performing this action. The combination of both pieces of data is what we refer to as the architectural state of that continuation. The execution of a continuation might generate one or more new continuation tokens. The continuation tokens start with an architectural state that is roughly a copy of that of the continuation that generated them.

Figure 3.1 shows a logical diagram of a P-RISC system. It consists of a pool containing continuation tokens and a collection of processing elements (PE). The continuations included in the pool define the work that must be done and the PEs are machines capable of executing such work. Each PE repeatedly consumes and executes one element from the pool and inserts back the newly generated continuation tokens, if any. The logical system also includes a global memory. This is a shared memory space where all the continuations can store data and communicate with each other. The continuations are small computations and are not necessarily pinned into any PE.

We consider three types of actions defined by a continuation:

1. **Linear actions.** Executing a linear action produces exactly one continuation token. The state of the new continuation is a copy of that of the consumed continuation, with some minor changes. For example, the program counter is incremented in arithmetic instructions or modified in control flow instructions. Arithmetic instructions modify the register set of the generated continuation.

2. **Fork actions.** Fork actions create two continuations. Each receives a copy of the architectural state of the consumed continuation. The PC might be different on each continuation.

3. **Join actions.** A join action generates either one continuation, with the same architectural state, or no continuation at all. It can be used to merge and synchronize continuations.

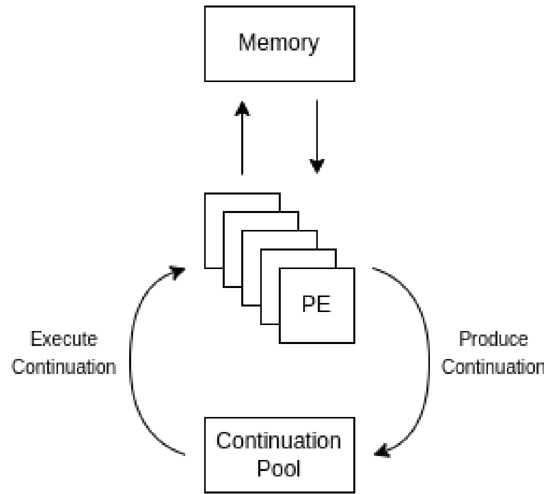


Figure 3.1: P-RISC abstract execution model.

We also consider the notion of a *stream*. A stream is a set of continuations that succeed each other in execution order. Implementing a sequential algorithm in P-RISC is equivalent to defining a linear stream of actions corresponding to the instructions executed by the algorithm implementation. A stream can be forked into two diverging ones with a fork action. Two streams can be merged using a pair of join actions. A stream is similar to the notion of a thread in many operating systems. However, we expect a lot of streams to be forked and joined during the execution of any given program with very little overhead. Thus, it should be considered as something more dynamic and lightweight. Also, streams do not have an explicit parent/child or ownership relation. Once forked, two streams become independent.

In this implementation, we consider continuations that require small actions, roughly corresponding to individual instructions from the targeted ISA. The aforementioned hardware state corresponds to one instance of the register sets defined by the ISA. Specifically, the action of continuation is indicated by a program counter pointing to an instruction of the program. Executing a linear action corresponds to the execution of an instruction from the standard targeted ISA. Fork and join actions are implemented as new instructions extending the ISA. We base the design on the RISC-V32 ISA because of the modularity and simplicity that it provides. P-RISC systems could however be implemented targeting any other ISA.

3.2 Architectural execution model

The model proposed in the previous section is abstract. We must now define a more specific architectural model. The continuations and linear streams mentioned in the abstract model are actually tasks implemented in a specific ISA. Executing one of these continuations is equivalent to executing a piece of sequential code. However, to our code, we add Fork and Join instructions. Fork instructions fork the calling continuation into two. When a Fork instruction is executed, a new continuation is generated. This continuation receives a copy of the current architectural state and a modified program counter in order to jump to a different code path. The calling continuation keeps executing the current instruction path.

Join instructions can be used in order to merge and synchronize two continuations. Join instructions are atomic memory instructions that toggle the value in the target address with the side effect of maybe terminating that particular stream depending on the value read. When two streams of instructions must merge, they both execute Join instructions targeting the same memory address. The value read from memory indicates which one was the first one to reach the rendezvous point. The first continuation

to execute the Join terminates. We call this an unsuccessful execution of a Join instruction. The second continuation that reaches the synchronization point does not terminate and keeps executing the current instruction path. We consider that a successful execution of a Join instruction. After both continuations have executed the Join instruction, they have effectively merged without any previous active waiting.

Our implementation is based on a RISC-V ISA. This is an obvious choice for two main reasons. First, it is open source and it is being adopted by many companies and research institutions [SL]. Second, RISC-V is a family of modular ISAs designed to be extended and adapted to different use cases. We add custom instructions that execute Fork and Join and enable the use of the P-RISC model. Executing standard RISC-V code becomes equivalent to executing a linear continuation. We expect algorithms to be mapped into parallel instruction streams. Therefore, we must implement a system that efficiently executes many simultaneous streams. We do that with an efficient SMT core. This core is able to execute many simultaneous continuations. The system also includes a token queue. Fork instructions push a new continuation into the token queue. When a hart becomes available in the core, the first continuation of the queue is loaded into it.

The Fork and Join instructions are added to the base RISC-V ISA. In order to actually code algorithms in a pragmatic way, we must provide primitives in a high-level language. We write all our test code in C. Alongside the hardware implementation, this work provides a basic C API with methods that wrap the Fork and Join instructions. We consider these to be our C primitives. They are internally implemented in assembly code but are compatible with the C calling convention. On top of that, we define block and reduce primitives. Those primitives split a task directly into many parallel streams and reduce the results at the end. That behavior can also be achieved with the original fork and join primitives. However, these identify common patterns and parallelization structures and provide a primitive that is easier to use on them.

3.3 Algorithms in P-RISC

In this section, we show a few examples of how one can intuitively map algorithms on the P-RISC model. All of these algorithms can potentially be implemented using other parallel paradigms. However, our model should provide very low overhead, allowing for very fine-grained tasks. In this section, we only show a few abstract examples. A specific and comprehensive list of implementations is included in a later section.

3.3.1 Mergesort

Let us consider a standard recursive implementation of the mergesort algorithm. This algorithm is based on a very simple divide-and-conquer strategy. The array being sorted gets split into two and each side gets sorted independently. Both partial solutions are then merged. This is usually done recursively. Thus, a binary tree of recursive calls is created, as shown in figure 3.2a. The depth of this tree is logarithmic with respect to the array size. It is very straightforward to realize that recursive calls on the same level of the recursion tree are actually working on independent sets of data. Therefore, it is very simple to divide this work into subtasks.

Suppose we want to divide the execution of this algorithm into four parallel tasks. We can actually take the calls in the second level and enclose them into independent tasks as shown in figure 3.2b. One can do this at different levels of the tree, proportioning different degrees of parallelism. Implementing this using a P-RISC machine is very simple. Figure 3.2c shows the control-flow graph obtained. We use fork primitives on the first levels of the tree recursively to split the search into parallel streams. Once the appropriate depth is reached - determining the number of parallel tasks, the implementation executes a standard sequential mergesort implementation. After that, join primitives synchronize the tasks, and the results are merged. This is a very natural strategy for this algorithm that actually executes in parallel most of the computation.

3.3.2 Matrix multiplication

Consider now a simple matrix multiplication algorithm as shown in listing 3.1[1]. This simple algorithm is composed of three nested loops. We will call them loops I, J, and K respectively. It is a well-known fact that the order of these loops may be exchanged without altering the final result. Also, due to the spatial locality in data caches, some orderings yield to better hit rates and performance. This algorithm has a certain geometric structure that makes it trivial to parallelize.

One can take any of the nested loops and split the iterations into independent tasks that can be executed in parallel. However, not all of them yield simple and efficient parallel implementations. Notice that matrices A and B are only being read, whereas matrix C is being not only read from but also written to. This means that two distinct iterations from the inner loop that use the same i and j and different k will try to read-modify-write the same position in C. This can create a data race. Therefore, splitting the iterations from loop K into distinct tasks requires very fine-grained synchronization. This might add too much overhead. Still, splitting loops I and/or J into independent tasks is always safe, as different i and/or j guarantee no conflicting accesses to matrix C.

This example showcases a common pattern in different algorithms where the final result is an array - or any sort of data structure - of several dimensions. If that is the case, a simple parallelization strategy is to divide the result array into blocks and assign them to independent tasks. Assuming that there are no data races, the parallelization strategy should be trivial and a P-RISC model will provide a very efficient way to generate and execute the corresponding tasks.

Let us assume a square matrix of size 32. We simply split the result matrix into four blocks of size 16 and execute independent matrix multiply operations in them. Figure 3.3 shows the control-flow graph obtained in this algorithm. This can be implemented manually using fork and join primitives to create and synchronize tasks. However, since this is a common pattern, we can use a standardized API call. Listing 3.2 shows a very simple implementation using a generic *block* P-RISC API in C. This API internally uses the join and fork primitives which are discussed in detail in a later chapter.

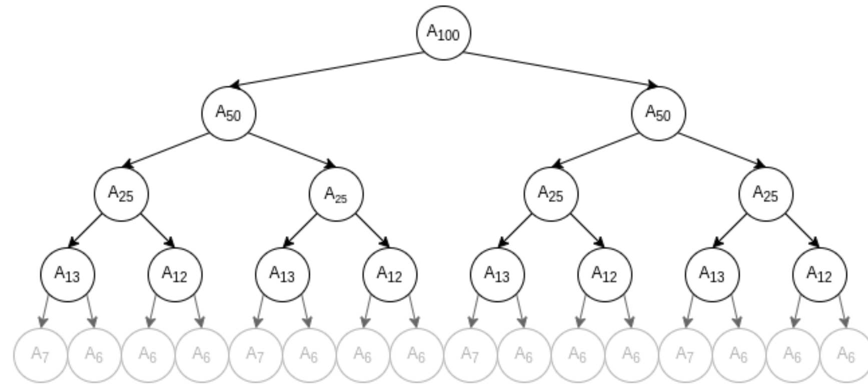
```
void mmt() {
  for (int i = 0; i < MSZ; ++i) // I
    for (int j = 0; j < MSZ; ++j) // J
      for (int k = 0; k < MSZ; ++k) // K
        C[i][j]+=A[i][k]*B[j][k];
}
```

Listing 3.1: Simple sequential matrix multiplication code. The matrices are assumed to be squared and of size MSZ. Matrix B is assumed to be transposed.

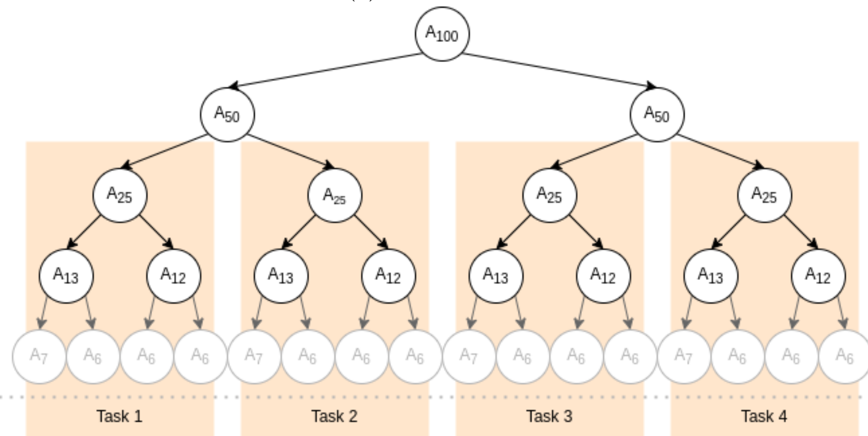
```
void mmt16(int I, int J) {
  for (int i = I; i < I+16; ++i) // I
    for (int j = J; j < J+16; ++j) // J
      for (int k = 0; k < MSZ ; ++k) // K
        C[i][j]+=A[i][k]*B[j][k];
}

void mmt() {
  block2D(0,MSZ,16,0,MSZ,16,&mmt16);
}
```

Listing 3.2: Simple parallel matrix multiplication code. The matrices are assumed to be squared and of size MSZ. Matrix B is assumed to be transposed. The result matrix is divided into square blocks of size 16 and then the computations are performed in parallel in each block. MSZ is assumed to be a multiple of 16. This can always be achieved by adding padding.

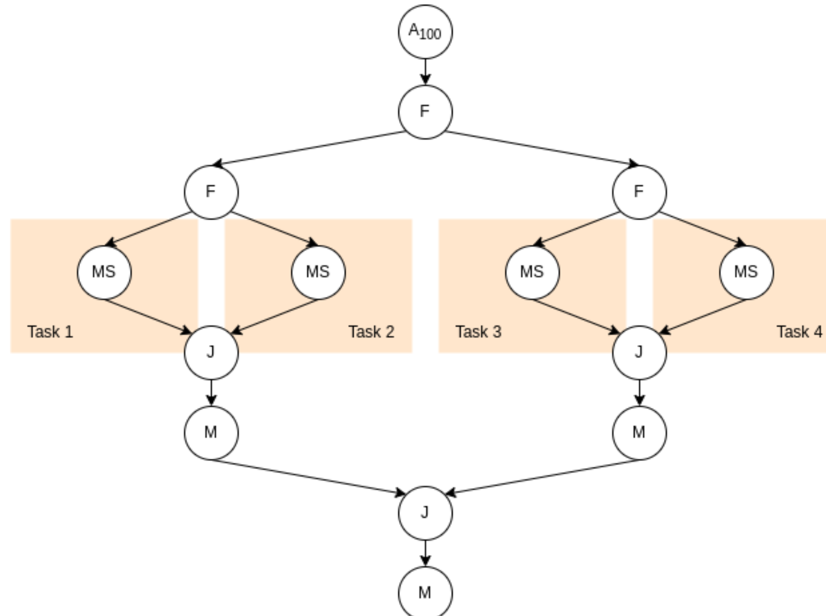
[1] We are actually assuming that matrix B is transposed for better cache locality.



(a) Recursion tree



(b) Parallel tasks over recursion tree



(c) Control-flow graph

Figure 3.2: Mergesort algorithm. Sub-indices in the nodes of sub-figures 3.2a and 3.2b represent the size of the array being sorted by that function call. Nodes in sub-figures 3.2c represent the following actions: Fork (F), Sequential mergesort (MS), Join (J), and Sequential merge/reduce (M).

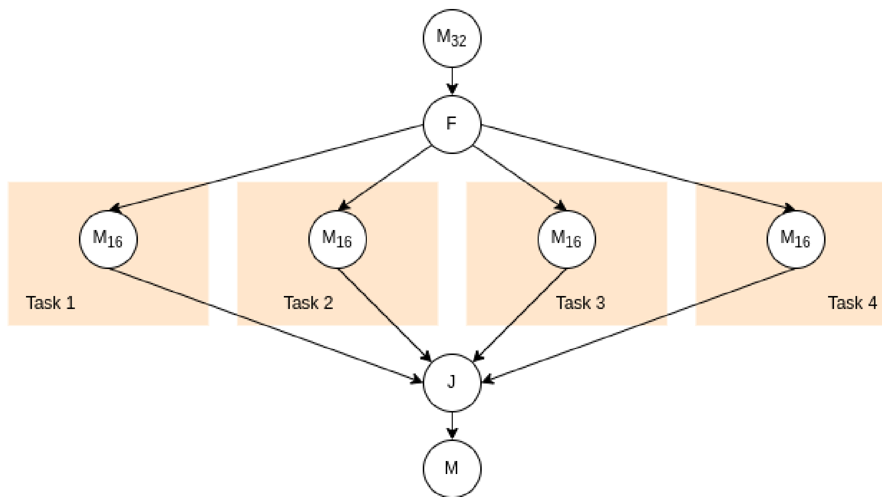


Figure 3.3: Matrix multiplication control-flow graph. Nodes represent the following actions: Fork (F), Sequential matrix multiplication of size 16 (M_{16}), and Join (J).

Chapter 4

Core microarchitecture and design

This chapter discusses in detail the microarchitecture of our SMT core, which is the main hardware component developed in this work. The four main goals of this core are to simultaneously host several continuations, treat memory accesses in split-phase, minimize the amount of resources wasted on wrong-path instructions, and execute Fork and Join instructions that create and terminate continuations. The idea behind these goals is to build a core that can easily discover the TLP exposed by the P-RISC model and efficiently execute it using the SMT capabilities and the split-phase memory system. We do not allocate any expensive hardware resources to improving single-threaded ILP. This implies executing in order, without branch prediction, and without instruction and data prefetching. Under a single-threaded load, the core appears to be a very simple in-order pipeline. The core designed here offers high throughput under TLP, even if single-threaded performance is poor.

At a high level, this core implements a narrowing in-order pipeline. At fetch stage, each fetch unit belongs to a hart hosting a different instruction stream[1]. Afterward, an arbiter module schedules instructions into the shared backend and execution units. The backend is composed of a single memory lane and a configurable amount of arithmetic lanes. When a stream sends a request into the split-phase memory system, the corresponding hart is locked and prevented from further executing instructions until the memory access is resolved. In the meantime, the rest of the harts keep fetching and executing instructions from other continuations in other to keep the pipeline fed. Finally, we need a mechanism that forks and joins continuations and schedules them into the frontend of the core.

The sections included in this chapter are as follows. First, we present the split-phase memory system, where the first level of caching acts as a filter, reducing the number of actual split-phase memory accesses, and the second level of caching increases the performance of the split-phase memory system itself. Second, we discuss the design of the actual execution pipeline. We then discuss in detail a few of the most relevant components of the core. This includes the L1I and the instruction fetching process, the instruction arbiter module and stage, and the L1D and execution of memory instructions. We describe then the way we map the architectural state of the continuations being hosted in the core to physical harts in a way that meets the requirements of the pipeline. Related to that are the control-flow mechanics of the pipeline and the NTTX module, which is in charge of forking and scheduling continuations. Finally, we present a detailed overview figure of the whole core.

4.1 Split-phase memory subsystem

One of the key features of our architecture is the split-phase memory system. Without a doubt, one of the main challenges of current processors is hiding the main memory access latency - which can be in the order of hundreds of CPU cycles. The basic principle that we use to overcome this problem

[1] We further discuss the usage of the term *hart* in a later section. By hart we roughly refer to the physical allocation within the core that hosts the architectural state of a given instruction stream. The number of harts in a core is equivalent to the number of streams that the core can host and simultaneously execute.

is split-phase accesses. The core idea here is to clearly split the memory accesses among request and response. When a stream of instructions must load a value from memory, it sends a request to the memory system. The core should then keep executing other work while the request is fulfilled. When the response arrives, that stream continues executing as usual. Furthermore, several streams should be able to stall and wait for their on-flight memory accesses while the rest keep executing with all the backend resources available to them.

Split-phase memory access is an important point that was already identified in the original P-RISC paper [NA89]. In fact, one of the uses of the P-RISC architecture highlighted in that paper is the ability to offload memory accesses into parallel instruction streams to prevent misses from halting the main computation kernel. In the original paper, the authors considered the idea of forking tiny tasks that only included a single or very few memory accesses. This way, the task executing the offloaded memory accesses would stall and wait for their completion while the original caller could keep executing other computations. This required a program implementation that considered at a very fine grain the memory accesses and how they were scheduled.

We believe that this idea is still key for our approach, but we do not believe that programs should define such tiny tasks. This is due to three reasons. First of all, such fine-grained decisions about hardware resource usage should be handled by the hardware itself and not by the software; as the software might not know enough about them and their usage at runtime. Second of all, forking and joining new tasks has a certain overhead associated. Even if we only consider the Fork and Join instructions themselves, the process of allocating and de-allocating a new hardware context has latency associated with it and uses a fairly large amount of hardware resources. This implies both that the calling stream observes an overall latency overhead and also that it uses hardware resources that could otherwise be allocated to other tasks. Third, this could only be achieved by either having very low-level compiler support or defining code by hand that wraps memory accesses.

What we propose is an SMT architecture that handles at a hardware level very fine-grained scheduling of the available tasks. Our SMT core hosts multiple simultaneous streams of instructions. Whenever one of them has to access main memory, it becomes locked and waits to get the response in a split-phase process. The hardware resources needed to do such an operation are minimal, as it only requires a place to store the current architectural state of the stalled stream. In the meantime, all other hardware resources are used by other continuations.

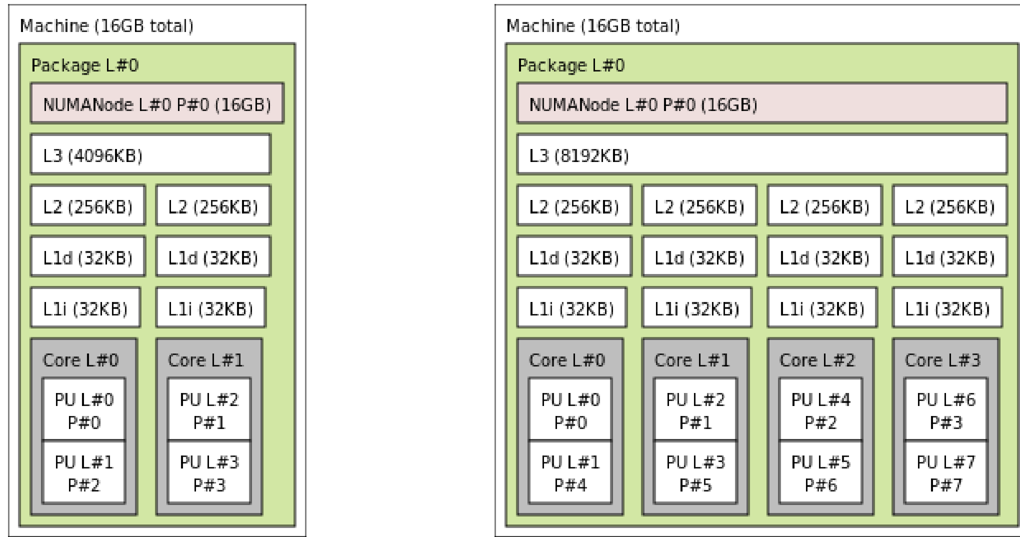
It is not reasonable however to assume that this idea is enough to hide the penalty if all memory operations were to access directly main memory. Consider a pipelined main memory system where the expected latency is n cycles. Every time an instruction stream sends a request to memory, it stalls for n cycles. Consider a system with m streams, each capable of waiting on an in-flight request. The total throughput of the core is limited to executing m memory operations - and a number of arithmetic operations - every n cycles. For a modern ASIC device, it is expected to observe a 100-cycle to 300-cycle latency to main memory. For programs that are mostly composed of arithmetic operations, having a few streams could be enough to hide the latency. However, for programs that have higher memory demands, considering the high memory latency, the number of necessary streams to hide it could become prohibitive.

A system that can host 100 simultaneous streams, for instance, would be required to host 12.8KB or 25.6KB of architectural state for RV32 or RV64 base integer ISAs and even more for CSR and floating-point extensions. This is possible if hosted in SRAM or BRAM instead of flip-flop-based structures. In fact, commercial GPUs have been hosting hundreds of computing SIMT threads - albeit with a very different architecture and computing model - since the NVIDIA Tesla generation [Lin+08]. Still, those machines need complex caching systems to obtain good memory throughput. P-RISC is no different. Even if our hardware could host hundreds of streams - and the algorithms could efficiently scale to this level of parallelism - the main memory would not have enough bandwidth to keep up with all the requests in memory-bounded applications. We do not directly use main memory as our split-phase memory system.

A caching system helps us solve both problems for P-RISC SMT, as in most architectures. If a caching system is added, the average latency per memory access is drastically reduced and, thus,

the number of streams required to hide it, keep the pipeline fed, and the hardware resources used. Furthermore, only accesses that miss move into the next level of the memory hierarchy. Thus, reducing the throughput required at the next level. This becomes even more relevant in multi-core systems, where the pressure against the memory system is even higher.

Most modern SoCs use a caching system composed of several levels. For instance, in figure 4.1 we show the cache hierarchy of two Intel x86-based systems - the computers that have been used to write this thesis. We can observe that both have independent first-level instruction and data caches for every core. The second level cache is larger and shared between instructions and data. The third and final level of cache is shared among instructions and data of all cores. In [MHS14] they study in depth the caching system and its impact on the memory subsystem performance of an older generation of Intel and AMD x86 systems.



(a) Topology of an Intel Core i7-7500U system.

(b) Topology of an Intel Core i7-6700K system.

Figure 4.1: Topology of two Intel x86 based systems as reported using the tool `lstopo` on Linux.

The caching system in our architecture has the same basic goals. At first sight, it might seem that having a caching system makes our split-phase memory system too complicated or difficult to implement. However, this is not actually the case. We think of our caching system as two distinct parts consisting of the first level of caching for both instructions and data and the second - and/or above - shared level of caching. Let us start by discussing the first caching level.

This first level of caching is fast, with only a 2-cycle latency, and tightly integrated into the pipeline. If a request produces a hit, it does not alter the regular flow of the pipeline, and the cache acts as any other execution unit. If a request produces a miss, there is also a response within 2 cycles indicating such an event. In that case, that particular stream is stalled and the request is forwarded to the memory hierarchy. The accesses that hit in L1 are not actually split-phase and do not disrupt the usual pipeline flow. Only the first-level misses stall the stream and send a request into the split-phase memory system. This way, the first level of cache does not try to increase the performance of the split-phase memory system but rather reduces the pressure that the core applies to it and the number of total stream locks and unlocks.

This caching model has two main advantages. From the core's point of view, having the first level of cache integrated into the pipeline reduces drastically the number of stream locks and unlocks related to memory accesses. Thus, reducing the overhead implied by such operations. From the split-phase memory system's point of view, it drastically reduces the number of outgoing requests and thus the pressure put into it. Again, this could prove especially relevant in multi-core systems.

Outside of the core, we add an optional second-level cache. This cache - and any other level that might be added above - is part of the actual split-phase memory system. The need for a second level depends on the environment and the observed main memory latency and throughput. Our caching system must manage responses out of order. Otherwise, one miss effectively blocks all the following requests, whether hit or miss, and the split-phase memory becomes serialized. This would increase the stall times of all streams waiting on on-flight requests.

In out-of-order memory hierarchies, requests must be tagged so that the core can identify the responses when they are received. There are many architectures that use a tagged memory subsystem. The goal is usually to be able to speculatively handle several memory requests at once. Our goal is rather to isolate the memory misses among stream instructions. We propose tagging the memory access simply by their hart identifier. Under hit, this is not necessary, as the L1I and L1D are integrated as functional units and do not alter the usual flow of the pipeline nor stall the requesting stream. Under miss, the request to the next memory level is tagged by the hart identifier.

We only consider having one instruction miss and one data miss in-flight per every hart. At any given time, a stream might have several memory instructions in flight in the memory pipeline, however. They are executed in a pipelined fashion as long as they cause a hit. If one of them misses, the younger ones are ignored and that particular stream is locked until the older request is resolved. This way, the tags that the L1 caches emit to the next level are small, only 2 or 3 bits indicating the hart that that access belongs to. In the second level cache, the tags are extended with one more bit indicating whether the client of the access was L1I or L1D.

The architecture described here implements a split-phase memory system that is actually very similar to the one described in the original P-RISC paper but with two added hardware features. First, the L1 caches act as a filter, reducing the number of memory accesses that are actually issued into the split-phase system. When they hit, they behave just as a regular arithmetic functional unit integrated into the pipeline. When they miss, in a way, they raise a hardware-managed exception that sends a request into the split-phase memory hierarchy. Second, the L2 cache acts as part of the split-phase memory hierarchy by reducing the average latency access and increasing the local throughput while maintaining the core functionality.

When compared to a typical memory hierarchy for a RISC-V device, our hierarchy has a very different interpretation and usage. However, the actual implementation of the caches is not that different. We should be able to substitute the second-level cache with any standardized memory hierarchy that can support the total number of in-flight requests produced by P-RISC SMT. This makes our architecture easier to justify in a production environment, as we should be able to reuse existing memory hierarchies. This means that a P-RISC core that implements a first level of cache with a standard coherence protocol could be integrated into a larger system with relative ease.

4.2 Narrowing pipeline

Each processor pipeline targets an ideal throughput. Simple in-order pipelines usually target an IPC of 1. Wider or OoO pipelines might target larger IPC, depending on their width. In P-RISC we target superscalar throughput on every core. This means that, under ideal circumstances, we would like the pipeline to process several instructions in every stage and every cycle. However, several factors can stall the pipeline and reduce performance. Our core uses a simple in-order narrowing pipeline that is proportioned to match the effective throughput of every stage. The frontend is composed of several fetch and decode units, one for each stream of instructions being hosted. The backend is composed of one memory lane and a configurable amount of arithmetic lanes.

On a simple processor pipeline, we may observe stalls due to:

1. Instruction cache miss.
2. Data hazards through registers.
3. Data cache miss.

4. Exceptions and branch misprediction.

Two well-known techniques for mitigating stalls are speculation and OoO execution. Speculation techniques try to minimize the number of possible stalls. This includes, amongst others, data prefetch and branch prediction. Data prefetch devices predict what memory lines will be needed in the near future and fetch them beforehand. This hides memory latency to the processor and might also help reduce noise on the memory hierarchy [VL00]. Branch predictors predict the direction of a branch early in the pipeline in order to both correct the instruction path being executed and prefetch the newly needed instruction cache lines. Although at a significant hardware cost, TAGE has been considered for the last few years to be the state-of-the-art branch prediction technique [Sez11].

Out-of-order execution hides most of the remaining stalls - either observed due to miss-speculation or the data dependencies among instructions through the register file. This typically requires renaming registers on instructions and having an oversize register file. Also, it requires a complex scoreboard and logic to roll back on exceptions or branch mispredicts. These techniques can vastly improve performance when compared to a simple in-order core. However, they require a lot of hardware overhead. We try to achieve high performance without those techniques and their hardware overheads.

Let us consider a 5-stage pipeline as shown in figure 4.2a. It does not include any sort of branch prediction or memory prefetch. This simple pipeline always fetches “PC+4”, unless a control-flow instruction is committed. In that case, a redirect request is sent to the fetch state. It uses an epoch system to avoid the committing of wrong-path instructions. Ideally, it would execute with an IPC of 1.

Different causes stall the pipeline at different stages:

1. Instruction cache miss: Fetch stage.
2. Data hazard: Decode stage.
3. Data cache miss: Memory/Commit stage.
4. Exceptions and branch misprediction: Fetch stage[2].

Consider now a superscalar pipeline with multiple lanes that execute instructions from several independent harts, as seen in figure 4.2b. This targets superscalar performance, but stalls limit the throughput. At any given moment, some of the fetch units could miss. This means that only a few of the lanes push instructions into decode stage. Some decoders remain unused. At decode stage some lanes stall due to a data dependency, so even fewer of those make it to execute stage. Then, several functional units remain unused.

The total throughput is determined by the number of valid instructions that reach commit stage each cycle, which is significantly less than the initial width of the pipeline. We can solve this inefficiency by implementing a pipeline that narrows on every stage that could cause a stall, as seen in figure 4.2c. If the width of every stage is properly proportioned, we should be able to efficiently make use of all of the available resources. The targeted throughput is the width of the commit stage.

A narrowing pipeline also allows us to balance the number of functional units of each type that our pipeline uses. For example, the first level of data caching can be very complex and resource intensive. We build a backend that has several lanes. However, only one of those has access to the single L1D and is able to execute memory instructions. The rest of the lanes only execute arithmetic instructions. The number of arithmetic lanes is parameterized and proportional to the number of expected arithmetic instructions. Figure 4.2c shows a design example.

Although we try to achieve superscalar behavior, this implementation is limited when compared to an OoO pipeline of similar width. First of all, we need to keep the instructions in order on a per-hart basis. If every level of the narrowing pipeline chooses instructions from the previous stage using

[2]Unless otherwise optimized, a branch misprediction is detected at commit stage. This renders pre-existing instructions in the pipeline invalid. The redirect process is re-initiated at fetch stage. If one ignores the wrong-path instructions, the pipeline has effectively stalled at fetch stage until it has been properly redirected. This is, however, an implicit stall, as it remains unsuspected of until the instruction reaches commit stage.

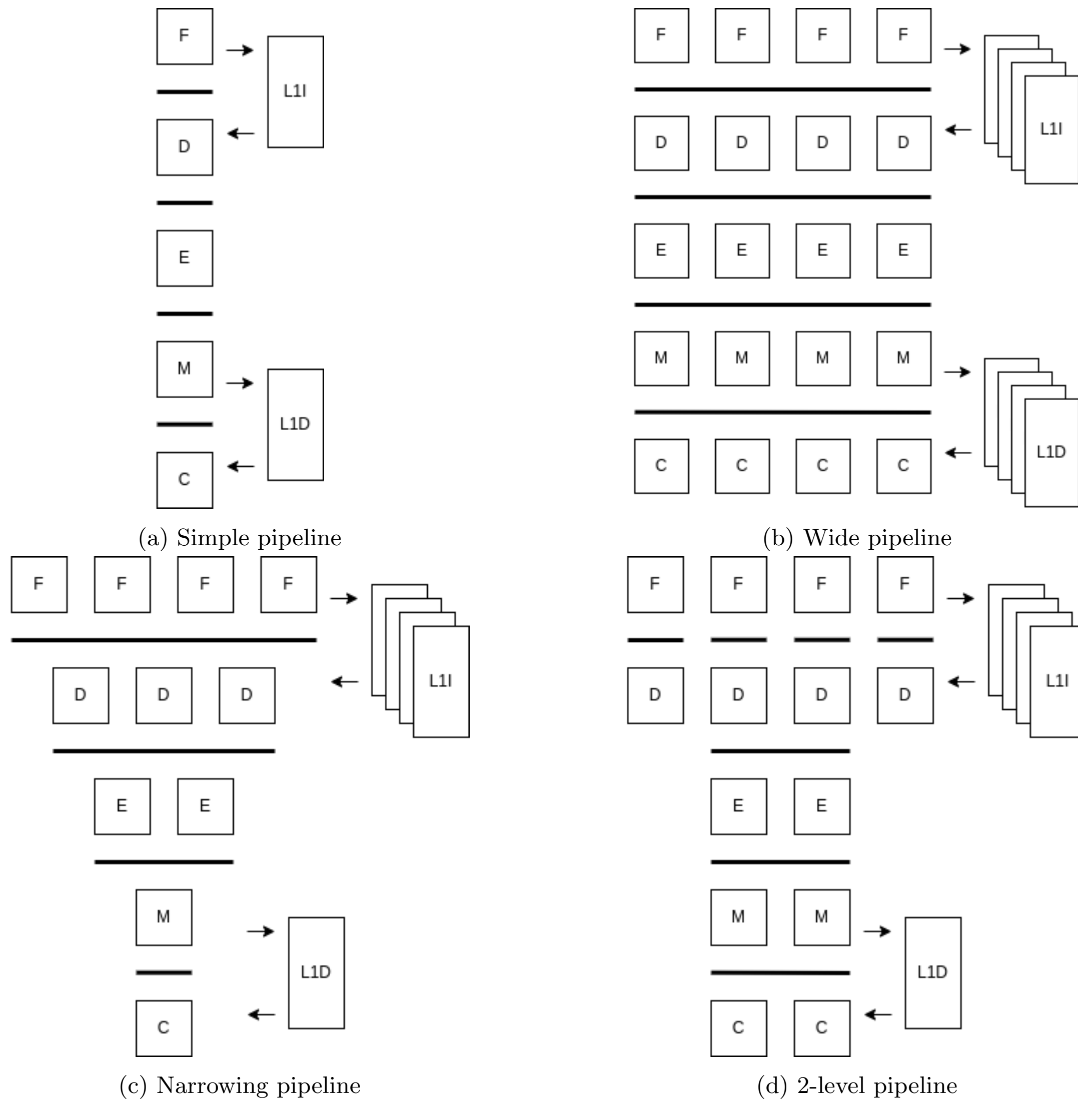


Figure 4.2: Simple processor pipelines with different widths. 4.2b, 4.2c, and 4.2d have four independent sources of instructions. In 4.2d, only the right-most execution lane has access to the L1D. The stages are: Fetch (F), Decode (D), Execute (E), Memory (M), Commit (C).

arbitrary criteria, instructions could become out of order after the second level of arbitration. However, we want to design a backend with lanes that host different functional units to balance the resources. This means that we need an arbiter with a fully associative behavior: Sending an instruction from any hart into the appropriate backend lane. Our pipeline is divided into two sections of different widths, as shown in figure 4.2d. Having only one level of arbitration that is fully associative does not cause a reordering problem. Still, this design allows us to hide from the backend many stalls produced on the frontend.

The frontend of the pipeline hosts multiple harts, each with a short pipeline that fetches and decodes instructions independently of each other. After that, an arbiter selects a few of the decoded instructions and schedules them into the appropriate lane of the backend. The arbiter makes sure that each wave of instructions only contains one from each hart. However, at any given moment, one hart could have instructions in flight at different stages of different lanes. If we treat them as independent pipelines and one of them stalls for a few cycles, instructions could become out of order. Therefore, we should not think of the backend as multiple independent pipelines but rather as a single pipeline with multiple lanes. This is more restrictive as a stall at any given point of the pipeline stalls all the lanes. Stalls are not to be expected however under normal execution.

Notice that the main difference between figure 4.2c and figure 4.2d is actually the reduction in width at decode stage. This is meant to compensate for the stalls due to instruction fetch misses. In our SMT core, each hart needs a separate register file even if the decode stage is narrower. In RISC-V, decoders are fairly small, and almost negligible when compared to the size of the register file or the scoreboard. Narrowing the decode stage would reduce the amount of decoder combinational logic but not the number of stateful elements, which are much larger.

In our synthesis experiments, we have observed that register files approximately take $6000\mu m^2$ of area, while decoders only use approximately $90\mu m^2$. We have experimented with the idea of sharing a single decoder every two harts but decided it was not worth it. The reduction in the overall area was negligible, as extra multiplexing logic was required, and a slight reduction in performance could be noticed in most tests. Although for our architecture this change was not worth it, it could be for other more complex ISAs that require larger decoder logic.

A noticeable example is the x86 ISA used in many modern commercial processors. The instruction set there is much more complex and many solutions actually map instructions into micro-ops. High-performance x86 systems even use a micro-operation cache that bypasses the decoder. Efficient decoding and micro-op caching for this ISA is an area of research that is still active [KK20]. In this case - or others with complex instruction sets - it could be worth it to reduce the number of decoding units.

4.3 L1I - Instruction fetch and decode

We address now the process of fetching instructions. Each hart in the core has a different PC. Fetching and executing the instruction indicated by any of the PCs is a correct and useful operation. At an abstract level, the frontend from this core is a unit that can arbitrarily fetch and decode an instruction indicated by any of these PCs and forward it to the backend of the pipeline. However, we need to avoid saturating the backend with instructions belonging to a single hart as the high throughput obtained by this system precisely comes from being able to execute many independent instructions concurrently. We need a system that fetches instructions from different harts independently with high throughput and stalls one of them precisely when the corresponding stream stalls.

Most programs benefit from the reuse of the same code - execution of loops and calling the same methods multiple times. Therefore, an instruction cache makes sense, just as in most processors. Also, if we assume that the different streams belong to the same program, they could be using a similar set of functions. This means that we should be able to observe the reuse of code in every stream but also among different streams. This way, it makes sense to share an instruction cache among harts to maximize reuse. This approach uses vastly fewer resources than having several replicated instruction caches for every hart.

Several harts might need to fetch instructions on the same cycle. This means that this system must be able to fetch many instructions per cycle independently, ideally, up to one per hart every cycle. Using a shared instruction cache, however, imposes a bandwidth bottleneck. We solve this problem by considering a fetch system with two levels of cache. First, each hart has a tiny L0I cache that it can use to fetch instructions independently from the rest. If it misses, the L0I requests the corresponding line into the L1I. The L1I is similar to a regular instruction cache, hosting several lines and allowing the reuse of instructions not only within the same hart but also among several of them. We can observe such a system in figure 4.3.

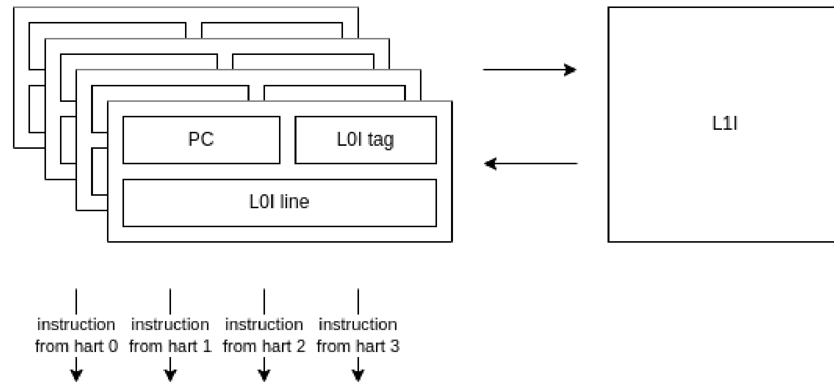


Figure 4.3: Instruction fetch system. Each hart can independently fetch instructions from the corresponding L0I system. Upon an L0I miss, the line is requested to the shared L1I.

Consider we are using a standard cache line size of 512 bits. RV32 instructions are 32 bits each - maybe 16 too when using the compressed instruction extension. Thus, each cache line can hold up to 16 instructions. If a hart is executing linear code, without control flow instructions, it will be able to fetch and decode 16 instructions from each line before having to request the following one. Control flow instructions can either jump to a different line and shorten the usability of such line or execute very small loops that fit in a single line and obtain the opposite result. Still, we can assume that once we fetch a line from L1I, we will be able to feed the corresponding hart for at least a few cycles.

This justifies implementing a very simple L0I that hosts only a single line of cache per each hart. Roughly every 16 instruction fetch, the L0I of each hart misses and requests the new line to L1I. This system drastically reduces the pressure put on the L1I and provides fetching independence between harts with very little hardware overhead. This means that a simple L1I that can serve one line per cycle should be enough to feed several harts - we expect to use 4 to 8 harts per core. Still, if this were not to be enough, a banked L1I could be considered, serving more than one request per cycle under hit. Also, an L1I with a single-cycle latency is not strictly necessary.

The L1I that P-RISC SMT includes is a 2-cycle instruction cache. It is be implemented on BRAM for FPGA targets and on SRAM for ASIC targets. Cache size and associativity are parameterized between a directly mapped cache and a set-associative cache. Of course, the degree of associativity has an impact on the physical layout and timing of the device. The L1I is internally implemented using a module called `BareInstCache`. This module stores the actual data, tags, and metadata. The L1I only instantiates a single `BareInstCache` module for direct mapping configurations and several of them for set-associative configurations.

The direct mapping index for a direct mapped or set associative cache has a great effect on the hit rate. The simplest index possible is obtained by simply selecting a predefined subset of the address bits. This does not always avoid conflicts in common patterns. More sophisticated hash functions can be used to compute indexes. xor-based hash functions are a very interesting family of hashing functions. The idea behind this family of hashing functions is to compute each bit of the index as the xor reduction of a certain subset of address bits.

The xor family of caches has two clear benefits for cache indexing. First of all, they are efficient in terms of circuitry and timing. xor reduction of several bits can be done in a tree structure. This will only require a linear amount of gates and add a logarithmic amount of delay. Second of all, they can be tailored to avoid conflicts in specific memory access patterns.

In [VD05] they define the address set and the index set as vector spaces when operated with bit-wise xor functions. In that framework, hash functions from address to index can be seen and expressed as linear maps of maximum range from one space to the other. Then, the kernel space of that map defines the linear subspace of addresses causing collisions. Furthermore, they propose a method by which such linear maps can be tailored to avoid conflicts in the required memory access patterns. For our L1I we use a simple stride xor hash function. Figure 4.4 shows the matrix corresponding to such mapping when expressed under the canonical base of the address and index vector spaces.

row								
25	.	1	.	.	.	.	.	.
24	1	.	.	.	.	.	.	.
23	.	.	.	.	.	.	.	1
22	.	.	.	.	.	.	1	.
21	.	.	.	.	.	1	.	.
20	.	.	.	.	1	.	.	.
19	.	.	.	1	.	.	.	.
18	.	.	1	.	.	.	.	.
17	.	1	.	.	.	.	.	.
16	1	.	.	.	.	.	.	.
15	.	.	.	.	.	.	.	1
14	.	.	.	.	.	.	1	.
13	.	.	.	.	.	1	.	.
12	.	.	.	.	1	.	.	.
11	.	.	.	1	.	.	.	.
10	.	.	1	.	.	.	.	.
09	.	1	.	.	.	.	.	.
08	1	.	.	.	.	.	.	.
07	.	.	.	.	.	.	.	1
06	.	.	.	.	.	.	1	.
05	.	.	.	.	.	1	.	.
04	.	.	.	.	1	.	.	.
03	.	.	.	1	.	.	.	.
02	.	.	1	.	.	.	.	.
01	.	1	.	.	.	.	.	.
00	1	.	.	.	.	.	.	.

Figure 4.4: Matrix of a stride xor-reduction hash function. The address vector space is assumed to be of dimension 26 - 32-bit ISA address minus a 6-bit offset for 512-bit lines - and the index vector space to be of dimension 8 - for 16KB of directly addressed memory with 512-bit lines.

The L1I always works with a granularity of a full cache line (512 bits). Access to individual instructions is managed by the L0I. When it misses, the L1I requests the line to the next level of the memory hierarchy. As mentioned previously, the L1I does only support one miss in flight per hart. The requests that the L1I issues to the next level are tagged with the identifier of the requesting hart. In response, the line is simply forwarded to the corresponding L0I.

The misses on L1I cause the corresponding stream of instructions to stall. The L1I only accepts one simultaneous request from each L0I. On misses, it does not resolve that access until it gets the response from the next level of caching. Therefore, the corresponding hart is not able to fetch instructions and continue execution. Also, it is not able/needs to issue any other request to L1I. When the miss is resolved and passed through to the L0I, the hart is implicitly unlocked.

4.4 Arbiter - Instruction issue

In this section, we discuss the design and implementation of the arbiter module. This module selects what instructions available on the frontend are forwarded to the backend every cycle. A naive imple-

mentation of this module works for very small configurations of the core - a very narrow frontend and only one or two lanes on the backend. However, that design does not scale well to more realistic configurations. This becomes apparent especially if we add any non-trivial scheduling policy. The module that we design here is much more scalable than the naive implementation both in terms of area and timing and implements a more refined scheduling policy.

In this implementation, we consider a backend that has two types of lanes - arithmetic and memory. However, this design should scale efficiently to host more distinct types of backend lanes. Also, we want fully associative behavior. This means that we want the arbiter to be able to issue an instruction coming from any hart in the frontend to any of the appropriate lanes in the backend. We define this problem as the selection problem. Let us start by defining a simple abstraction of the I/O requirements of this module.

On the input side, we find a set of independent queues, containing the instructions available from each hart. At any given moment, some of these might be empty. Otherwise, the available instructions are either arithmetic or memory. We define the input vector as an array containing instructions belonging to the head of each one of these queues. If a queue is empty, that entry is tagged *Invalid*. If it is not empty, the corresponding entry contains the first instruction available and is tagged either *Arithmetic* or *Memory* depending on the type.

On the output side, the arbiter issues an array containing the set of instructions to be forwarded to the arithmetic and memory lanes, respectively, using the output queue. Additionally, the instructions selected must be dequeued from the input queues at the end of the cycle. To that end, the vector entries also tag the instructions with their hart identifier.

4.4.1 Naive design

The naive solution to the selection problem uses combinational BSV syntax in order to express with a high-level language the desired behavior. Listing 4.1 shows a possible implementation for such a simple design. In a smaller configuration, this design works. However, it does not scale well at all. The first problem is related to the BSC compiler. BSC performs strong static analysis in order to check the integrity of the rules and methods and try to simplify the implementation as much as possible. In this design, this becomes a problem. The static analysis becomes way too complex and crashes on larger configurations[3].

This problem can be mitigated using the BSC `(*noinline*)` attribute. This attribute requests BSC to generate a separate Verilog module for the function, simplifying the static analysis and forwarding the design to downstream tools [Blu10]. If that is done, compilation and synthesis can be successful - unless the configuration is very large. However, the physical design obtained is poor: It is slow, too large, and takes a long time to synthesize.

[3]The BSC static analysis process actually runs out of memory on the host machine due to the complexity of the design. In particular, it exceeds 16GB of memory available on the machine. For reference, in other projects of similar size it is expected that 1GB to 2GB should be enough.

```
(*noinline*) function InstSelect select(Vector#(FrontWidth,Maybe#(ExecToken)) inst);

    Vector#(BackWidth,Maybe#(ExecToken)) forward  = replicate(tagged Invalid);
    Vector#(FrontWidth,Bool)             takenM   = replicate(False);
    Vector#(FrontWidth,Bool)             takenA   = replicate(False);
    Vector#(FrontWidth,Bool)             taken    = replicate(False);

    // Arithmetic
    for (Integer i = 0; i < valueOf(FrontWidth); i=i+1) begin
        for (Integer j = 1; j < valueOf(BackWidth); j=j+1) begin
            if(isValid(inst[i]) && !takenA[i] && !isValid(forward[j])
                                && isArithInst(fromMaybe(?,inst[i]))) begin
                forward[j] = inst[i];
                takenA  [i] = True;
            end
        end
    end

    // Memory
    for (Integer i = 0; i < valueOf(FrontWidth); i=i+1) begin
        if(isValid(inst[i]) && !isValid(forward[0])
                            && isMemInst(fromMaybe(?,inst[i]))) begin
            forward[0] = inst[i];
            takenM  [i] = True;
        end
    end

    for (Integer i = 0; i < valueOf(FrontWidth); i=i+1)
        taken[i] = takenM[i] || takenA[i];

    return InstSelect{ taken: taken, forward: forward};

endfunction
```

Listing 4.1: Naive implementation of the core combinational logic of the arbiter module.

4.4.2 Sorting network design

The design proposed in this section to solve the selection problem is based on two key ideas:

1. Selecting memory and arithmetic instructions are two independent matters.
2. Selecting valid instructions is equivalent to sorting the input vector.

Consider the I/O abstraction previously defined. The arbiter must select a set of arithmetic instructions and a set of memory instructions. The type of each instruction strictly restricts the set to which it can be added. Thus, one can ignore the instructions from the wrong type when trying to define one of these sets and the two problems become independent. Thus, the selection problem splits into two that can be solved in parallel.

Each one of these problems can be solved by sorting the input vector according to validity - within the type of instruction being considered - and selecting the lower positions of this vector. If the number of available valid instructions is larger than the number of lanes, all the lower positions of the output array contain valid instructions. Otherwise, some lanes remain vacant. Still, all available instructions are scheduled in that case. Therefore, this design always maximizes the number of instructions being forwarded to the backend and, thus, the potential throughput.

In figure 4.5 we show a schematic of the solution proposed to the selection problem. First, two copies of the input vector are considered and the validity of each entry is masked according to the instruction type. Thus, we obtain two arrays containing only valid instructions of a certain type. The valid positions of these vectors do not overlap. Then, these vectors must be sorted. We can do that

simultaneously with two sorting networks, which we call *Arbiter sorting network* (ASN). Then, the lower positions of such vectors are selected and concatenated to form the output vector. Finally, the selected instructions must be dequeued from the input queues. This can be done by decoding the identifier tags on each entry. We know that those tags do not collide, thus, it should be a simple task.

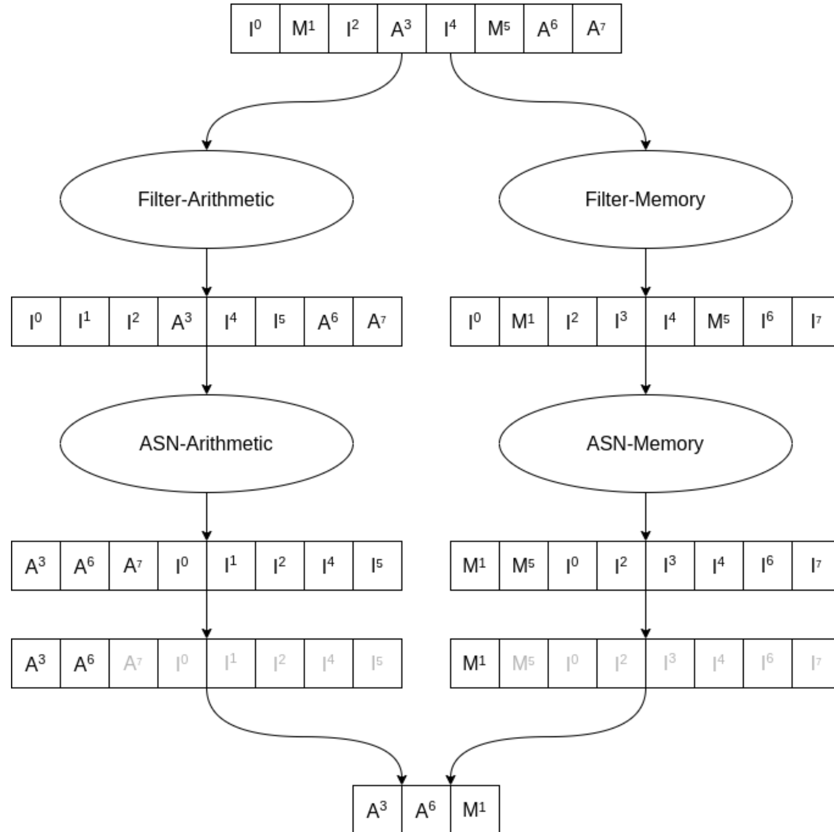


Figure 4.5: Schematic of an efficient implementation of the arbiter stage logic. Invalid, arithmetic, and memory instructions are denoted, respectively by *I*, *A*, and *M*. Super-indices denote the owner hart for those instructions. The particular instance shown here considers a configuration with 8 harts at the frontend, and 2 arithmetic lanes, and 1 memory lane at the backend.

The more complex circuit in this design is the ASNs - the rest can be efficiently specified using BSV high-level syntax. However, it must be taken into account that the actual comparison on each layer is very simple - only one validity bit if no priorities are added. There are many well-known sorting network designs that could be used here. Any reasonable implementation of the ASN yields a design that is much better than the naive one both in area and timing, as shown later in this text. Furthermore, adding priorities to this design is trivial. Entries of the input array can be also tagged with the desired priority scheme and this is used as part of the comparison in the ASNs. Therefore, the ASNs shift to the left all valid instructions and sorts them by priority.

4.4.3 Speculation level and priority

The basic selection problem can be extended and solved easily with the ASN and the design proposed in order to implement a more refined scheduling policy. In our machine, some instructions may cause a change in the instruction path when commit. This could either be caused explicitly by a control-flow instruction or implicitly by a memory instruction that misses and needs to stall the corresponding

stream of instructions. Notice that these only redirect the corresponding stream of instructions. The rest of the streams are independent and keep executing as usual. Recall that we do not use branch prediction logic and therefore a few wrong-path instructions might have already been fetched and decoded - and even forwarded into the backend - after executing any control-flow instruction. When the arbiter chooses which instructions to issue, we want to maximize the choice of right-path instructions among the different streams available. The technique proposed here is not meant to guarantee that all instructions issued into the backend are safe but rather to increase resource usage efficiency and overall performance. We therefore still require an epoch system in order to ensure the correctness at commit stage.

The main reason behind the necessity of a more refined scheduling policy is to avoid polluting the backend lanes with invalid instructions. The throughput offered by the backend is limited, therefore we want to maximize the usage with right-path instructions. One possibility to avoid this problem is to lock any stream that issues a speculative instruction until it commits. This prevents the backend from getting filed with wrong-path instructions. However, this actually yields poor performance - especially in lightly-threaded applications - as streams are constantly locked. The idea we propose is to use a speculation index for every stream of instructions that indicates how likely are the instructions from that thread to be wrong-path.

We assign a speculation-level counter to every stream of instructions. Every time the arbiter issues an instruction that could cause a redirect, the corresponding counter is increased. The amount increased depends on how likely is that instruction type to cause a redirect. Then, it is decreased by one unit every cycle. A stream that has just issued several dangerous instructions has a very high speculation level. This means that the lower the speculation level of a stream, the lower the chances are for younger instructions to be wrong-path. This counter roughly predicts how many dangerous instructions does each thread have in flight. We could implement an exact metric by providing feedback on the instructions being committed and redirects being issued by the backend. We decided to use the predictive method in order to save hardware resources. The predictive method does not alter the interface provided by the arbiter nor requires modifying any other module and only adds a minimal amount of stateful elements into the module. Still, it increases performance significantly in some programs.

This metric can be easily incorporated into the arbiter design. This way, the arbiter prioritizes instructions with a lower speculation level. Instead of sorting just by the validity of instructions, it sorts using combined criteria:

1. Any valid instruction is ordered lower than any invalid instruction.
2. Two invalid instructions are equal in order.
3. Two valid instructions are ordered according to speculation level.

In practice, we have seen that a speculation index of 3 bits has enough precision. This is due to the fact that the latency from when a dangerous instruction is issued to when it is committed and the redirect is received is strictly lower than 8 cycles. Thus, the maximum time for the speculation level of a stream to saturate to zero is matched by the duration of the speculation. Longer and more complex pipelines might benefit from adding more precision.

The comparison required at every level of the ASN is a 4-bit less-than - concatenating the validity bit and the 3 speculation level bits. This was not achievable by the naive implementation in a reasonable synthesis. Insertion sort or bitonic sort ASNs manage this level of precision in reasonable configurations. For larger configurations, the bitonic ASN pulls ahead and offers a faster critical path while taking 3 to 4 times less area. As we show in our results, adding the speculation level sorting noticeably increases performance.

4.4.4 Multi-cycle design

Taking into account the timing restrictions that the arbiter is subject to, it would be natural to consider a pipelined arbiter design. In fact, during the development of this project, the arbiter has many times

become the bottleneck in terms of timing for both FPGA and ASIC synthesis and also has taken a significant proportion of the LUTs/cell area. It should be noticed that the ASN is not only deciding what instructions to issue but also moving the actual data into the forwarding queues. Making a pipelined arbiter, however, is not as simple as pipelining other units.

Pipelining a sorting network is usually a straightforward task. Most sorting networks - including insertion and bitonic - can be split into layers and then sets of layers can be assigned to different stages. In [MTA12], they study the efficient implementation of sorting networks on FPGAs achieving very competitive designs. However, in this particular application, there is a feedback loop that makes matters more complicated.

After selection, the instructions selected on a given cycle must be dequeued from the input queues. This cannot be done until the selection process has finished. Therefore a new selection process cannot be started in a pipelined fashion. A multi-cycle arbiter would not be able to de-queue instructions until the end of the process. Therefore, no new instructions can be input and considered earlier. This means that, effectively, this would be a multi-cycle unit but not a pipelined unit. Still, this design could make sense.

Let us consider an arbiter of width w divided into k stages. This arbiter can only make a selection of w elements every k cycles. Thus, it cannot offer a w IPC. However, one could make an arbiter of width wk and k stages, achieving a maximum theoretical throughput of w IPC. The time complexity of the bitonic sort is $log^2 n$, where n is the width of the array being sorted. Thus, the theoretical delay on such arbiter would be $log^2(wk)/k$ when divided into k stages. This factor tends to zero when k is increased, lowering the delay.

Still, this solution is not perfect due to four reasons:

1. The space complexity of such a network is $nlog^2 n$. If divided into k stages, this expression becomes $wklog^2(wk)$. This clearly increases rapidly on k, therefore increasing the size of the total combinational logic required.

2. Pipelining a design also adds extra cost in terms of stateful elements.

3. Pipelining a design adds extra *set* and *acquire* delay between stages.

4. In the ASN context, having a multi-cycle design implies that new instructions available are only be considered every k cycles. Thus, ignoring new valid instructions for a few cycles and reducing the effective throughput of the system.

In this project, we have not tried to implement a multi-cycle arbiter as it was not the main topic being addressed and the benefits are not yet clear. However, as shown here, we believe it could make sense depending on the fabric and width of the pipeline.

4.5 L1D - Memory access instructions

In this section, we discuss the process of executing memory instructions in the backend of the core. Memory instructions in P-RISC include both load and store operations and join operations. From a memory point of view, join operations essentially execute an atomic read-then-modify operation; similar to the standard RISC-V AMO instructions. These instructions are usually executed at the first-level data cache. Although the number of arithmetic lanes is parameterized, we include a single memory lane, having access to a single L1D. The arbiter module schedules all memory instructions coming from the different harts into the memory lane. The maximum theoretical throughput of this system is limited to executing one memory instruction per cycle. Under hit, the L1D is tightly integrated into the pipeline and can be seen as any other functional unit. The L1D acts as a filter, only the accesses that miss in this cache access the split-phase memory system.

The L1D, as the L1I, is pipelined and has a two-cycle latency. The mapping, also as the L1I, is direct. As already discussed, xor-based hash functions offer a good trade-off between performance and

implementation simplicity and timing when used as indexing functions. In the L1D we also use a stride xor hash function for indexing. Figure 4.6 shows the matrix obtained for the hash function used here as represented in the bases proposed by [VD05]. It would actually be the same one as found in the L1I if the sizes were equal. The cache is implemented in the `BareDataCache` module instantiated by the `L1D` module.

row									
25	.	.	.	.	.	.	.	1	.
24	.	.	.	.	.	.	1	.	.
23	.	.	.	.	.	1	.	.	.
22	.	.	.	.	1	.	.	.	.
21	.	.	.	1	.	.	.	.	.
20	.	.	1	.	.	.	.	.	.
19	.	1	.	.	.	.	.	.	.
18	1	.	.	.	.	.	.	.	.
17	.	.	.	.	.	.	.	.	1
16	.	.	.	.	.	.	.	1	.
15	.	.	.	.	.	.	1	.	.
14	.	.	.	.	.	1	.	.	.
13	.	.	.	.	1	.	.	.	.
12	.	.	.	1	.	.	.	.	.
11	.	.	1	.	.	.	.	.	.
10	.	1	.	.	.	.	.	.	.
09	1	.	.	.	.	.	.	.	.
08	.	.	.	.	.	.	.	.	1
07	.	.	.	.	.	.	.	1	.
06	.	.	.	.	.	.	1	.	.
05	.	.	.	.	.	1	.	.	.
04	.	.	.	.	1	.	.	.	.
03	.	.	.	1	.	.	.	.	.
02	.	.	1	.	.	.	.	.	.
01	.	1	.	.	.	.	.	.	.
00	1	.	.	.	.	.	.	.	.

Figure 4.6: Matrix of a stride xor-reduction hash function. The address vector space is assumed to be of dimension 26 - 32-bit ISA address minus a 6-bit offset for 512-bit lines - and the index vector space to be of dimension 9 - for 32KB of directly addressed memory with 512-bit lines.

The L1D must be able to support as many misses as harts in the core. We map every new miss into a mshr while the request is being handled by the memory hierarchy. While a miss is on-flight, a different hart might try to access the same line of memory. In that case, the request is added to the already allocated mshr, and the memory line is not requested again to the memory hierarchy. When the response is received, all pending accesses into that line are resolved. We need the same number of mshr as harts in the core in order to support that many misses. The simplest way of allocating the mshrs is by directly indexing them by the hart identifier. When a stream from a hart generates a miss that is unique, it is always allocated into the mshr belonging to that hart. The misses that are not unique, are added into the mshr belonging to the hart that produced the oldest miss into that line. Accumulating misses to the same line not only reduces the pressure put into the next level of cache but also eliminates a possible line aliasing problem, making consistency simpler.

Although the maximum theoretical throughput is one request per cycle, this level of performance could not hold due to two factors. These are back-to-back instructions that write into the same line and misses that effectively put two requests into the `BareDataCache`. This is a problem that can significantly harm performance. Because the backend is a single, wide, pipeline - rather than individual pipelines; the whole execution is stalled when the L1D stalls. In any case, such an event is rare and we do not observe many stalls.

In order to execute memory instructions we need to first read the current values of the cache in order to decide if it is a hit or a miss. Loads simply return the value read when they hit. Store instructions must also write. We cannot issue the write until the access has been confirmed to be a hit. This means that store instructions actually issue two requests. The second request is issued two cycles after the

original one and, at that point, the hit confirmation can already be sent to the commit stage of the core. In order to support this behavior we use a two-port BRAM/SRAM. One port is used for reads and the other for writes.

A two-port memory should allow us to achieve one request per cycle of throughput. However, requests accessing the same line cannot be executed back to back. Otherwise, the youngest access might not observe the write done by the oldest access. When back-to-back requests are issued to the same line, some stalls might be caused. Misses also cause some stalls. When a miss is resolved by the memory hierarchy, the newly fetched line must be stored in the cache. This generates yet another access and overall use some of the available throughput of the cache. The two factors discussed here slightly reduce the throughput under access collision.

In order to maintain or increase the theoretical cache throughput we could consider using a multi-banked cache. In our design, that can be done in two ways. First, we might implement the L1D using a collection of independent `BareDataCache` modules banked by address. This hides the stalls produced by resolving misses as long as those misses are not mapped into the same bank as the younger requests being currently handled by the L1D. Second, we could target a higher throughput by expanding the backend into multiple memory lanes with multiple L1D units. In that case, each memory lane would address a subset of memory addresses. The arbiter module should pre-compute a fraction of the address of memory requests in order to decide to which lane it shall be issued.

It should be noted that the consistency model enforced by our current implementation is very strong. The fact that hits are pipelined and we only consider one outstanding miss per stream guarantees that the level of consistency observed by the user is sequential. Under hit, all requests are issued and treated by the L1D in order. Under miss, younger requests belonging to the same hart are ignored until the older access is resolved again. Therefore, the total set of requests as sorted by their commit point - whether after the original access hits or when the miss is resolved - defines a global ordering that is observed by all streams of instructions.

The sequential consistency model is very strong and guarantees the safest behavior to the user. However, most high-performance frameworks guarantee weaker consistency models. Weaker models typically allow the hardware to re-order accesses to increase performance. In fact, the base RISC-V ISAs use the RISC-V Weak Memory Ordering (RVWMO) model, and the Ztso extension forces the microarchitecture to implement A Total Store Order (TSO) consistency model [WA19b]. Hardware implementations must ensure *at least* RVWMO or TSO in order to be considered RISC-V or RISC-V with Ztso extension. Implementations can always guarantee stronger memory models and will be considered correct.

The sequential consistency guaranteed by the core developed here boils down to the fact that we only support one outstanding data miss per hart. Although we theorize that this should be enough to yield good performance when executing many streams, it is probably not enough when executing only a few in a kernel that is strongly memory-bounded. One interesting area for future research would be to consider supporting several outstanding misses per stream of instructions. This modification might also be tied to having several memory lanes. For instance, we could consider hosting one outstanding miss per hart per memory lane. In this work, however, we have not implemented such a feature and we maintain the one miss per hart policy.

4.6 Harts and physical mapping of the architectural state

Each core in our P-RISC system is able to manage several streams in parallel using a narrowing pipeline. For this purpose, we use the term *hart*, borrowing and extending its meaning from the RISC-V terminology. A hart corresponds to a set of hardware resources to which the state of a given continuation or stream can be mapped. The number of harts that a core includes is equivalent to the number of simultaneous streams that this core can execute. Our implementation separates the architectural state for the given harts but can execute simultaneously instructions from any of them in the shared functional units. However, this is an implementation decision that remains transparent to

the user. From a software correctness point of view, each stream mapped into a hart executes in total isolation.

The harts implemented must be able to host the architectural state of the streams being hosted. This is composed mainly of the program counter and the register set. We will now discuss the physical resources used to actually host such an architectural state. Recall that our backend is a wide superscalar pipeline. A typical configuration has a single memory lane and a few arithmetic lanes. At commit stage, the pipeline needs to write back the results of several instructions which are guaranteed to belong to different harts and modify disjoint sets of architectural state. Therefore, we need a physical implementation that allows for simultaneous writing to the program counter and register set of several harts.

We must decide whether to use a shared structure that hosts the architectural state of several harts and limits the number of simultaneous writes or to use independent structures at a slight resource overhead. The program counter is small and is modified on every instruction fetch - either incremented by 4 on most instructions or modified somehow in control flow instructions. This means that every cycle each hart might need to update theirs. Thus, it makes sense to simply implement them as actual independent registers, one per hart. The register set is much larger, however - 1024 bits per hart in RV32I. There are two possible approaches. We could implement either a single, shared, large register file or several independent, small, register files.

If we want to use a single register file, once one instruction has been fetched and decoded, the backend stages should be able to execute it in a way that is agnostic to the hart that produced the instruction. A very simple renaming process can implement such an abstraction. Let us assume that each hart has a unique identifier. This is a small value - 2 or 3 bits for an implementation with 4 or 8 harts, for example. After decoding an instruction, one can generate extended virtual register indexes by concatenating the identifier of the corresponding hart and the register indexes indicated by the instruction. Then, we use the extended indexes in the following stages of the pipeline and address a shared register file. One could also add a renaming process in an implementation with OoO execution in a relatively simple way by renaming the extended virtual indexes into physical indexes. In any case, this abstraction would allow us to implement a design where a single physical register set is needed and shared among harts.

Every instruction executed can read up to two operands and store up to one result in the register set. Thus, it could make sense to implement it using a single register file as described above. However, if we are targeting superscalar performance, we need to be able to read and write several values per cycle. This can put significant pressure on the register file hosting these sets. Still, the register sets from different harts are independent and they never interfere with each other. Thus, the parallel decode stages never try to fetch more than two operands from the same set and the wide commit stage never tries to write more than a single register to each set. Therefore, implementing an independent register file for each hart also makes sense and is a much simpler solution.

In our implementation, we are targeting superscalar performance in an in-order pipeline. Thus, the second option seems more appropriate. This way, the frontend stages can independently fetch operands from the respective sets as usual. At commit stage, we know that several instructions might try to write results to the register sets. However, because instructions are kept in order, they will never try to use the same set. Still, this design adds a level of indirection that other designs do not have: Any of the backend lanes can execute instructions from any hart and the results must be re-routed at commit stage to the appropriate register file in a fully associative way.

4.7 Control-flow, redirections, and epochs

The control-flow system in most pipelines has the basic goal of avoiding the execution of wrong-path instructions. This would mainly include redirecting the pipeline after the execution of control-flow

instructions[4]. In this work, we implement a simple epoch system for that. Recall that in our pipeline each fetch unit is fetching instructions belonging to different continuations. Therefore, the control-flow system manages a different epoch for each hart. It also has the goal of locking and unlocking the harts that are waiting on in-flight memory access in the split-phase system. We do that by adding a "lock" to the frontend of each hart that is toggled by locking and unlocking redirects.

The first level data cache should be a functional unit with only two cycles of latency, tied to the normal execution of the pipeline. However, a miss disrupts the regular flow of instructions. The corresponding memory instruction cannot be committed on the expected cycle. Still, the L1D is expected to always send a response within two cycles, either resolving the access or informing of the miss. In that case, the instruction cannot commit, other in-flight instructions from that same hart must be invalidated with a redirect and the hart locked. After the access is resolved, the L1D sends a *late response*. Then, the requesting hart is unlocked and continues execution.

This control flow scheme introduces two differences when compared to a standard epoch system. The first difference is that redirects can lock and unlock harts. When a data cache miss is detected at commit stage, the corresponding thread must be not only redirected to discard the in-flight instructions but also locked, to prevent it from issuing them again until the access is resolved. We propose this lock is implemented right before the arbiter. This way the corresponding hart can start fetching and decoding the next instructions again, but they will not be issued into the backend until they are safe.

The second difference is that two different stages can send redirect requests. We can think of a pipeline that adds a later *late commit* stage, as shown in figure 4.7a. Memory instructions that miss in the first level get to that stage and send an unlock petition. The late commit stage is not an actual physical stage of the pipeline, the L1D actually holds the corresponding token until the access is resolved and can be committed. However, for control-flow purposes, we can think of it as an actual stage. This means that the epoch needs to be a 2-bit value, as instructions can cause up to two redirects during execution.

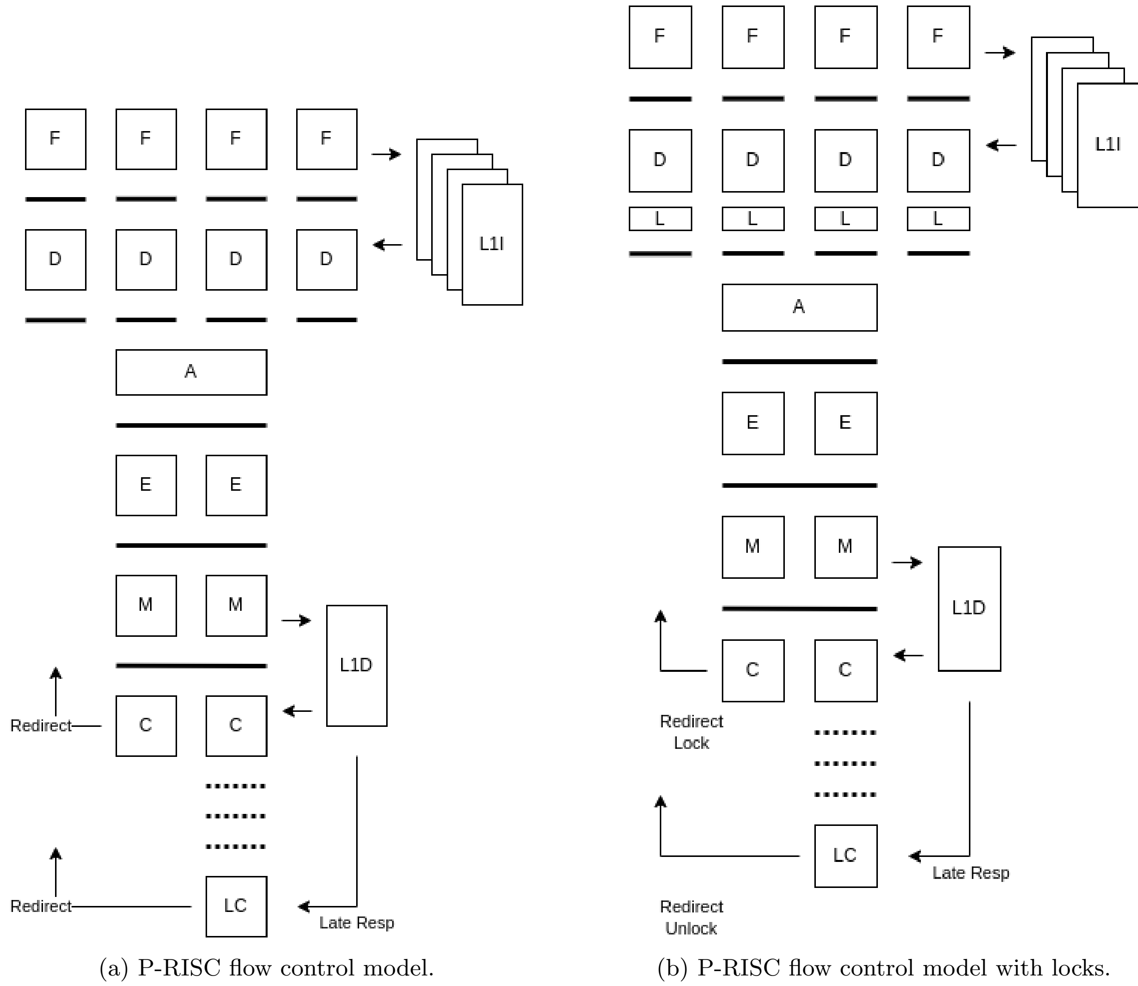
Finally, it is desirable to extend the mechanism so that it can evict arbitrary streams from the core. This can be used to implement hardware-based scheduling policies. We implement that using *ghost* instructions. When an evict request is received, the first component to react is the corresponding fetch unit. This unit forwards a special instruction containing the program counter from the next expected instruction but tagged as a *ghost*. The actual instruction is neither fetched nor executed in the pipeline. Rather, this is used to flush the pipeline from all ongoing instructions belonging to this particular hart.

When the ghost instruction reaches commit stage, all older instructions have been committed and there are no younger ones in flight. Thus, we can now safely generate a continuation token containing the program counter indicated by the ghost instruction and reading the current values from the register file. Then, a *ghost redirect* is sent to the frontend indicating that the stream has been evicted and a new one can be scheduled. It must be noticed that, despite not being a real instruction, control-flow mechanics also apply to ghost instructions. For instance, if a mispredicted branch is in flight when a ghost is generated, the ghost also must be invalidated and redirected, as it would otherwise generate an incorrect continuation. A new ghost will be generated.

4.8 NTTX - Stream fork and scheduling

The NTTX is the main module in charge of forking and joining streams of instructions. Here we discuss exactly how this module is used in order to fork new streams of instructions and manage them. Recall that the semantics of the Fork and Join instructions are as follows. When a stream executes a Fork instruction, a new continuation token is generated. This token represents a continuation that receives a copy of the register set from the original continuation and a modified program counter. The Join

[4]Exceptions are also an important consideration. From a control-flow point of view, exceptions behave for the most part as control-flow instructions, redirecting the execution of code to another path. In our simple core, we do not implement exceptions. However, if they were present, they would be mostly treated as described here for control-flow instructions.



(a) P-RISC flow control model.

(b) P-RISC flow control model with locks.

Figure 4.7: P-RISC pipelines with epoch and locking systems.

instructions try to test-and-set a position in memory. Depending on the value read, the stream either terminates - unsuccessful join - or continues executing the next instruction - successful join.

Fork and Join instructions are executed using the memory lane. This is due to the fact that the NTTX can handle a single request per cycle. Channeling all the Fork and Join instructions through the same lane makes scheduling access to this module trivial. Furthermore, the Join instruction is essentially an atomic memory instruction. We have observed that in most programs the amount of extra pressure that this puts in the memory lane is negligible.

Let us start by discussing the Fork instructions. The execution of these instructions starts by computing the jump address for the new continuation. This is encoded and computed in a similar manner to that of regular JAL and JALR RISC-V instructions. When this instruction reaches commit stage, a request is sent to the NTTX with the jump target address. Now, the NTTX must read a copy from the register file values and generate the token. This token is stored in a small queue within the NTTX. When a hart in the core becomes available, a token from this queue will be scheduled into it - if there are any pending to be executed.

The process of copying the register file could take a few cycles. Thus, we must prevent the pipeline from modifying it until the copy has been transferred to the NTTX. We do that by using the epoch system already present in the pipeline. When a fork instruction is committed, a locking redirect is sent. This redirect is actually trivial and redirects the hart to "pc+4"; which was already the next instruction being fetched. The significance of this redirect is therefore not the redirect itself but rather the fact that it locks the corresponding hart and prevents the register file from being modified. Once the NTTX has finished with the transfer, it will send a redirect that simply unlocks the hart again. The hart will keep hosting and executing the original continuation then. For arbiter scheduling purposes, fork instructions are marked as speculative, just as control-flow instructions, in order to minimize the impact of the redirect.

Join instructions are essentially atomic memory operations with the potential side effect of terminating the continuation. They are executed in the memory lane using the L1D, as all of the memory operations. However, at commit stage, they are treated differently. If the Join was unsuccessful, the stream must be terminated. We do that by simply marking the corresponding hart as available. If new continuations are pending to be executed in the NTTX, one of them will be scheduled into that hart. When the Join instruction commits, some instructions following it might be already in-flight. A simple epoch change purges them before scheduling the new stream. Notice that Join instructions might miss in L1D and enter the split-phase memory system. In that case, the commit of a Join instruction will be performed at late commit stage.

We have mentioned here the *small* token queue present in the NTTX module. This queue is a fast hardware queue implemented using flip-flops. We need such a fast queue in order to make agile scheduling of the new continuations that have been forked by the NTTX. However, recall that the continuation tokens are fairly large pieces of information. This largely limits the number of tokens that we can reasonably store in such a queue. We need a larger structure to store the overflow of the NTTX. This is what we call the MTQ. This queue is external to the core - maybe shared among several cores - and implemented using BRAM/SRAM. The MTQ is further discussed in the next chapter.

4.9 Full core overview

Figure 4.8 presents an overview of the full core developed in this work. The frontend of the core is composed of the fetch and decode stages of the pipeline. Those are composed of several independent pipelines, each fetching and decoding instructions from independent streams; and the first-level instruction cache. The arbiter implements the next stage of the pipeline, issuing instructions into the backend. The backend is composed of three execution stages and includes the first-level data cache. It also includes the *Late Commit* special stage that commits memory instructions that miss in the L1D and enter the split-phase memory system. Finally, the NTTX module is shown at the bottom of the figure. The L1I and L1D are connected to the memory hierarchy and the NTTX connects to the MTQ.

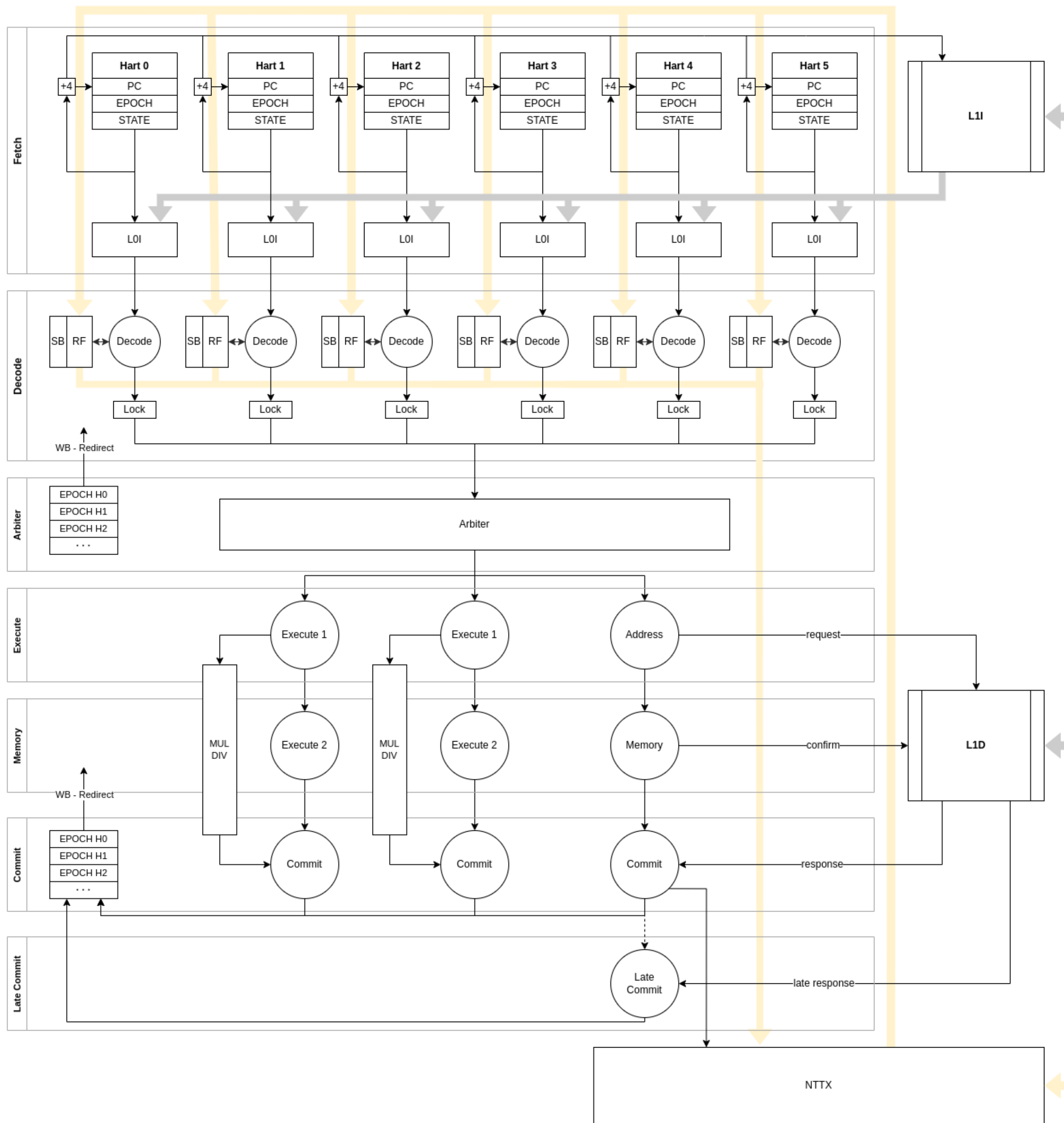


Figure 4.8: Overview of the full P-RISC core. Grey lines denote cache line buses and yellow lines denote continuation token buses. The core is shown in a configuration with 6 harts, 1 memory lane, and 2 arithmetic lanes. The external connections go from the L1I and L1D into the memory system and from the NTTX into the MTQ.

Chapter 5

System microarchitecture and design

In the previous section, we discussed in detail the different components and microarchitecture of the P-RISC core itself. That also included the first level of caches and a small queue in the NTTX to temperately store the tokens generated by fork instructions. In this section, we discuss the components that are part of the hardware system but sit outside of the core. In fact, some of them could be shared by several cores in a multi-core configuration. It should be noted that many of the ideas implemented here have not been actually implemented or only in a basic capacity. The goal of this section is not to propose a refined system implementation, but rather to identify the hardware requirements at a system level and explain how some of them have been solved in this work.

5.1 L2S - Second level cache

In the previous sections, we discussed the possible need for a second level of caching. Depending on the environment it might be needed to increase the performance of the split-phase memory system. In the evaluation sections, we discuss the performance impact of this on different benchmarks and environments. Here we explain the internal implementation of our L2S, although the instance of this device in the system is optional.

The second level of caching is shared among instructions and data and always works at a granularity of a 512-bit cache line. On one side, the clients for the L2S are the L1I and the L1D. They can both support one outgoing request per hart. Thus, the L2S should be able to handle at least two outgoing requests per hart in the core. On the other side, the L2S requests into the main memory controller. The memory controller in our testbench is a parameterized module that abstracts the main memory of the system and instantiates different implementations depending on the environment. This component is discussed in detail in the testbench section. The interface that it exports to the L2S is never modified tough.

We want to support the split-phase memory model. Thus, our cache must be able to handle requests out of order. This could mean two different things. First, the L2S might send the responses in a different order than the corresponding requests were received. This is done so that one miss does not also stall consequent hits. Second, the L2S might be able to accept main memory responses out of order. Our design can do both using the existing hart-based tags.

The requests received from the first level of caches are tagged by the hart identifier that issued that particular access. On a hit, the L2S sends a response with the same tag. On a miss, that request is placed in a mshr and issued to the next level. The tag of the request forwarded to the next level is actually extended by one bit indicating the original client of the request. This avoids collisions between data and instruction requests coming from the same hart.

In terms of performance, our L2S can manage one request per cycle and has a latency of 5 cycles. The latency observed by the core is actually 7 cycles, as the send and receive queues from the first level of caches add an extra cycle of latency in each direction. As a rough point of comparison, we have observed the DDR4 controller in our FPGA evaluation board to be able to handle comfortably one request every 10 core cycles with a latency of 17 to 18 core cycles.

The L2S is an associative cache. The size and degree of associativity of the cache can be configured. For our tests, we settled into 128KB of total memory split into 4 banks of 32KB each. In the evaluation section, we run the benchmarks both with the L2S in place and with direct access to main memory from the first-level caches. We started with a set-associative design. As we did with the first-level caches, the index within each bank is based on a stride xor hash function. In [VD05] they also argue the benefits o skewed-associative designs against set-associative designs. Therefore, we decided to modify our original design in order to implement a simple skewed-associative scheme. In our design, we chose to use the same stride function in each bank but offset it by one unit in the original address. In figure 5.1 we show the matrices obtained for each bank for a 4-way associative configuration.

The advantage of a skewed-associative cache against a set-associative is the ability to increase *interbank dispersion*. In a set-associative scheme, a set of blocks that map into the same index for one bank also map into the same index for all banks. Using different hash functions for the different banks should lower the chances of collision in all banks in the most common patterns. This increases the ability of different blocks to be spread out through the cache. This property is called interbank dispersion [Sez93]. In early testing, this modification showed a slight increase in hit rate for our test programs.

5.2 MTQ - Global pool of continuations

The ability of the P-RISC execution model to generate parallelism is based on continuations. Every time an algorithm can benefit from parallelism, it must fork into independent streams of instructions. At a hardware level, this generates a continuation token. This token symbolizes a piece of work that must be executed. A continuation token is something that we should be able to “load” into a core for it to be executed. Thus, it must host the architectural state of the stream of instructions that it is attached to.

The P-RISC core designed in this work includes a small token queue in the NTTX. Fork instructions push into that queue. Whenever a hart becomes available, it will pull a continuation from the queue and start executing the corresponding stream of instructions. This makes sense for a small program. However, larger programs might fork into many streams of instructions. A hardware queue might not be able to host all of them efficiently. It does not make sense to store many continuations in a core if it does not have the bandwidth to execute them. We need a structure that is external to the core and can hold the overflow of continuation tokens generated by large programs.

We add into our system the *Main Token Queue* (MTQ). This structure takes the spill from the internal queue of the core. The core will fetch again the continuations whenever it is ready to host them. This external structure is justified by two main reasons. First, the hardware queue in the core has limited capacity, and increasing it would have a noticeable increase in area. An external unit adds flexibility in terms of implementation and will be more easily adapted to the environment. Second, an external structure can be used as a means of load balancing in a multi-core system. Consider the situation where one core of the system is executing a stream that discovers a lot of parallelism and generates a lot of tokens. The first few tokens will be hosted by the same core and start executing immediately. If too many tokens are generated, they will spill out into the MTQ. Then, the MTQ should be able to migrate them into a different core that still has resources available.

As we use a RISC-V ISA as our base, the architectural state is composed of a 32-element register set and a program counter[WA19b]. In RV32 and RV64, the register set is composed of 32 entries totaling 1024 and 2048 bits respectively. This can be stored in exactly 2 and 4 cache lines respectively. This makes it a very convenient piece of information to store using standard memory devices. We still need

row									
25	.	.	.	.	.	.	.	1	.
24	.	.	.	.	.	.	1	.	.
23	.	.	.	.	.	1	.	.	.
22	.	.	.	.	1	.	.	.	.
21	.	.	.	1	.	.	.	.	.
20	.	.	1	.	.	.	.	.	.
19	.	1	.	.	.	.	.	.	.
18	1	.	.	.	.	.	.	.	.
17	.	.	.	.	.	.	.	.	1
16	.	.	.	.	.	.	.	1	.
15	.	.	.	.	.	.	1	.	.
14	.	.	.	.	.	1	.	.	.
13	.	.	.	.	1	.	.	.	.
12	.	.	.	1	.	.	.	.	.
11	.	.	1	.	.	.	.	.	.
10	.	1	.	.	.	.	.	.	.
09	1	.	.	.	.	.	.	.	.
08	.	.	.	.	.	.	.	.	1
07	.	.	.	.	.	.	.	1	.
06	.	.	.	.	.	.	1	.	.
05	.	.	.	.	.	1	.	.	.
04	.	.	.	.	1	.	.	.	.
03	.	.	.	1	.	.	.	.	.
02	.	.	1	.	.	.	.	.	.
01	.	1	.	.	.	.	.	.	.
00	1	.	.	.	.	.	.	.	.

(a) Matrix of a stride xor-reduction hash function with an offset of 0.

row									
25	.	.	.	.	.	.	1	.	.
24	.	.	.	.	.	1	.	.	.
23	.	.	.	.	1	.	.	.	.
22	.	.	.	1	.	.	.	.	.
21	.	.	1	.	.	.	.	.	.
20	.	1	.	.	.	.	.	.	.
19	1	.	.	.	.	.	.	.	.
18	.	.	.	.	.	.	.	.	1
17	.	.	.	.	.	.	.	1	.
16	.	.	.	.	.	.	1	.	.
15	.	.	.	.	.	1	.	.	.
14	.	.	.	.	1	.	.	.	.
13	.	.	.	1	.	.	.	.	.
12	.	.	1	.	.	.	.	.	.
11	.	1	.	.	.	.	.	.	.
10	1	.	.	.	.	.	.	.	.
09	.	.	.	.	.	.	.	.	1
08	.	.	.	.	.	.	.	1	.
07	.	.	.	.	.	.	1	.	.
06	.	.	.	.	.	1	.	.	.
05	.	.	.	.	1	.	.	.	.
04	.	.	.	1	.	.	.	.	.
03	.	.	1	.	.	.	.	.	.
02	.	1	.	.	.	.	.	.	.
01	1	.	.	.	.	.	.	.	.
00	.	.	.	.	.	.	.	1	.

(b) Matrix of a stride xor-reduction hash function with an offset of 1.

row									
25	.	.	.	.	.	1	.	.	.
24	.	.	.	.	1	.	.	.	.
23	.	.	.	1	.	.	.	.	.
22	.	.	1	.	.	.	.	.	.
21	.	1	.	.	.	.	.	.	.
20	1	.	.	.	.	.	.	.	.
19	.	.	.	.	.	.	.	.	1
18	.	.	.	.	.	.	.	1	.
17	.	.	.	.	.	.	1	.	.
16	.	.	.	.	.	1	.	.	.
15	.	.	.	.	1	.	.	.	.
14	.	.	.	1	.	.	.	.	.
13	.	.	1	.	.	.	.	.	.
12	.	1	.	.	.	.	.	.	.
11	1	.	.	.	.	.	.	.	.
10	.	.	.	.	.	.	.	.	1
09	.	.	.	.	.	.	.	1	.
08	.	.	.	.	.	.	1	.	.
07	.	.	.	.	.	1	.	.	.
06	.	.	.	.	1	.	.	.	.
05	.	.	.	1	.	.	.	.	.
04	.	.	1	.	.	.	.	.	.
03	.	1	.	.	.	.	.	.	.
02	1	.	.	.	.	.	.	.	.
01	.	.	.	.	.	.	.	1	.
00	.	.	.	.	.	.	1	.	.

(c) Matrix of a stride xor-reduction hash function with an offset of 2.

row									
25	.	.	.	.	1	.	.	.	.
24	.	.	.	1	.	.	.	.	.
23	.	.	1	.	.	.	.	.	.
22	.	1	.	.	.	.	.	.	.
21	1	.	.	.	.	.	.	.	.
20	.	.	.	.	.	.	.	.	1
19	.	.	.	.	.	.	.	1	.
18	.	.	.	.	.	.	1	.	.
17	.	.	.	.	.	1	.	.	.
16	.	.	.	.	1	.	.	.	.
15	.	.	.	1	.	.	.	.	.
14	.	.	1	.	.	.	.	.	.
13	.	1	.	.	.	.	.	.	.
12	1	.	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	.	1
10	.	.	.	.	.	.	.	1	.
09	.	.	.	.	.	.	1	.	.
08	.	.	.	.	.	1	.	.	.
07	.	.	.	.	1	.	.	.	.
06	.	.	.	1	.	.	.	.	.
05	.	.	1	.	.	.	.	.	.
04	.	1	.	.	.	.	.	.	.
03	1	.	.	.	.	.	.	.	.
02	.	.	.	.	.	.	.	1	.
01	.	.	.	.	.	.	1	.	.
00	.	.	.	.	.	1	.	.	.

(d) Matrix of a stride xor-reduction hash function with an offset of 3.

Figure 5.1: Matrices of a stride xor-reduction hash function with different offsets. The address vector space is assumed to be of dimension 26 - 32-bit ISA address minus a 6-bit offset for 512-bit lines - and the index vector space is assumed to be of dimension 9 - for 32KB of directly addressed memory with 512-bit lines.

to store the `pc` though, which might seem inconvenient. However, all RISC-V ISAs have a property that plays to our advantage here. As per ISA definition, the register zero does not actually store a value, it always reads zero. This means that in the continuation tokens, we might use the first entry of the register file to piggyback the value of the `pc`. Thus, a continuation token can hold all the architectural state of RV32 or RV64 ISAs in exactly 2 or 4 cache lines.

A more extensive P-RISC implementation might include RISC-V extensions that add architectural state. The floating point extension adds a floating point register set and the `fcsr` register [WA19b]. The CSR extension adds the CSR registers - albeit it should be discussed how CSRs are integrated into such a system, as each one might be tied into a stream, a physical hart, a core, or the whole system. These extensions, therefore, increase the amount of state that must be mapped into the continuation tokens and increase their size. A particular implementation might also add information that is not visible to the user to implement scheduling policies or other load-balancing policies.

The system that we implemented in this work is single-core and aimed at a nimbler environment. As we have limited execution bandwidth we expect algorithms to not generate a huge amount of continuations under a reasonable implementation. We believe that it makes sense to implement the MTQ of this particular system as a BRAM/SRAM-mapped queue. Under the FPGA execution environment, the MTQ is mapped into a BRAM queue. In an ASIC target, this should translate naturally into an SRAM-based queue. These storage elements are slightly slower than the hardware queues included in the core; but can efficiently store a much larger amount - in the order of hundreds - of continuation tokens.

We believe that in higher-end systems, however, the MTQ should be mapped into main memory. This is due to two reasons. First, in large multi-core systems - especially the ones implementing extensions that add architectural state - one could expect a larger amount of continuation tokens that might not be reasonable to store in BRAM/SRAM-based solutions. Furthermore, the MTQ should be mapped into an MMIO or reserved memory region. This means that the software EEI could actually define the size of the MTQ and even make it dynamic. Second, if the MTQ is mapped into a memory region that is accessible by software, this information could be used in more refined load balancing schemes where the amount of parallelism exploded and how it is allocated is balanced dynamically by observing the MTQ and inferring the current load of the system. All in all, an MTQ mapped into main memory could define a system that is more flexible and easier to adapt to different software environments.

The caveat of mapping the MTQ into main memory is the decrease in performance with respect to BRAM/SRAM-based solutions and the increased pressure into main memory. Both factors could reduce the rate at which continuations are created and scheduled and the memory bandwidth available to actual streams executing in the cores. We believe that this could be solved by caching part of the MTQ into BRAM/SRAM. This mitigates both problems. For instance, a simple solution could use an SRAM to host the head of the MTQ and store the tail in main memory. When not many continuations are generated, they are managed quickly by the SRAM system. When too many are generated and they spill into main memory, that SRAM device will still host a few continuations that can be accessed and scheduled quickly when a hart becomes available.

5.3 Multi-core systems

The execution model and architecture proposed in this work are focused on obtaining high throughout when the algorithm being executed can be extensively parallelized. It only makes sense therefore to consider a multi-core system. In this section, we discuss why multi-core P-RISC systems make sense and why they should be relatively easy to implement.

The throughput that a given P-RISC core can achieve is limited by its width and the memory access paths. Although there are certainly ways to increase them, we believe that at some point a multi-core system should be considered where each core hosts a few streams that share resources. On one hand, the microarchitecture of the core designed here is very associative. The L1I is shared among fetch units.

The arbiter can schedule an instruction from any frontend hart into any backend lane. Any functional unit might be used by any stream of instructions. On the other hand, the internal microarchitecture of the core is relatively rigid and deterministic. It is important to keep in mind that the backend of our core should not be considered to have many independent pipelines but rather a single, wide, pipeline. We believe that these characteristics make it efficient when executing a few independent streams of instructions but they might limit scalability into very large configurations.

The limited performance scalability of cores is not a unique problem to P-RISC. For the last two decades, it has been understood that processing devices must be designed as multi-core solutions to provide competitive performance in relation to the area and power consumed. Furthermore, it is believed by many authors that heterogeneous systems are key to achieving efficient performance. Sources of heterogeneity in processors might be classified roughly into static and dynamic sources. Static sources are mainly due to actual design differences. Many current multi-core processors use two or more types of cores. Vector processing units, accelerators, and specialized cores are also becoming fundamental. Dynamic sources of heterogeneity can be introduced at runtime by voltage/frequency scaling or due to fabrication process variability and/or defects. In [FC08] they argue the relevance of heterogeneous systems and further discuss the possible sources of heterogeneity. In [Gup+] they address the increase in scheduling complexity due to heterogeneous systems.

We believe that a P-RISC execution model will inherently benefit from a heterogeneous multi-core system. Our execution model allows the programmer to efficiently split the algorithm into smaller tasks. It has already been established that this model enables a high level of parallelism and requires a parallel microarchitecture. One of the key ideas of task scheduling for heterogeneous systems is matching each core with the task that it is better suited for [FC08] [Gup+]. Having fine-grained parallelism should split complex algorithms into smaller tasks that could be better matched by different cores. This way, the parallelism obtained by a P-RISC model could also allow for more efficient scheduling in heterogeneous systems.

In this work, we have only designed and implemented one core and it has been tested only in single-core configurations. However, we believe it is clear that making such core - and others - ready to be configured in a multi-core environment is fundamental. We believe that adding P-RISC cores into a multi-core environment should be only slightly more challenging than adding standard RISC-V cores. The two main points have already been identified in this work.

The first challenge is to adapt the split-phase memory model of P-RISC into one of the standard memory coherence protocols. We believe that this should be a relatively simple task. As addressed in the previous chapter, the first level of caching is part of the core itself and acts as a filter that reduces the number of requests sent into the split-phase memory model. The requests forwarded from the first level of cache are tagged and it is expected to receive the responses out of order. Most high-performance systems, protocols, and memory hierarchies work with the same basic principle of tagging memory requests. This is something that maybe was not obvious at the time of writing the original paper on the P-RISC architecture [NA89]. However, we believe that it is reasonable to add a standard coherence protocol to the specialized first level of caching from this architecture and then use existing memory hierarchy caches and components.

The second challenge is to support the MTQ and scheduling of the instruction streams. The hardware support of the MTQ has been discussed in the previous section. Although it is an added degree of complexity, we believe that it should be attainable. Furthermore, having hardware-based support for task creation and scheduling could have very interesting implications for scheduling policy for multi-core and heterogeneous systems. Standardizing an MTQ interface could allow the software solutions to have access to a larger amount of information about the tasks available and their characteristics. This could be used to improve scheduling policies in a hybrid software-hardware solution.

Chapter 6

Testbench, synthesis tools and hardware verification

The system developed in this project has been implemented mainly in *Bluespec SystemVerilog* (BSV). This is a high-level hardware description language that can be synthesized into different targets for evaluation. The *Bluespec Compiler* (BSC) compiles BSV into standard Verilog. The Verilog design can then be compiled into RTL simulation, FPGA, and ASIC targets. In this section, we describe the integration of the design in different evaluation environments and profiles and how we use them to verify and test the design. By *evaluation environment* we refer to whether the device is tested in RTL simulation, FPGA, or ASIC. By *evaluation profile*, we refer to the methodology that we use to extract information from the device and how this information is evaluated.

6.1 Bluespec SystemVerilog

BSV is an object-oriented *Hardware Description Language* (HDL). Like with most HDLs, the design can be modular and parameterized. That is, the device being implemented can be divided recursively into modules that are then implemented individually. Then, smaller modules are instantiated as components of larger modules. Unlike most HDLs, the internal implementation of the modules is specified by parallel atomic rules, and the interfaces connecting different modules are defined by abstract methods - instead of low-level signals and combinational logic. Communication between modules is achieved by calling those methods from the containing module.

This methodology splits the design of hardware into two separate problems of functional correctness and performance. The correctness of the design is guaranteed by the designer for individual rules and methods. BSC compiles these rules into parallel Verilog code ensuring that the behavior specified by the rules is observed to be atomic. That is, different rules are only allowed to execute in parallel if they do not interfere with each other. Performance is determined in great part by the degree of parallelism achievable for the rules implemented. A design where the rules defined are correct but cause many conflicts among them is guaranteed to be safe by BSC but will likely have very poor performance.

We chose BSV as the main implementation language for several reasons mainly focused on the ease of development and design verification. The novel approach that BSV takes to hardware design makes the development of new devices agile. On one hand, the modular and object-oriented nature makes a very simple task from modifying the design and adding functionality. On the other hand, the atomicity of BSV constructs makes it simple to verify the correctness of the design at a high level. Although we did not do it for this work, this nature can also be used in order to formally verify the design against abstract models using mathematical reduction techniques.

The main drawback of this HDL is the opacity of the compiled design in terms of performance. BSC rule scheduling decisions and static analysis can be difficult to understand and decide on. In

consequence, achieving the expected performance can prove to be complicated. This problem has been identified and studied in [Bou+20]. Here, they develop Koika, a derivative language from BSV that is more expressive in terms of performance and rule scheduling specification; but should preserve the functional correctness and ease of formal verification provided by BSV. Still, we believed that the advantages of BSV already proved it to be a better alternative to classic HDLs for this project.

Downstream tools usually do not have direct support for BSV descriptions. However, the Verilog code produced by BSC is standard for the most part and can be further synthesized by most downstream tools into different targets for different evaluation environments for the same project. Only a few of the components need to be tailored to the synthesis target. We have used the following three evaluation environments:

1. RTL Simulation - Verilator
2. FPGA - Xilinx VCU108
3. ASIC - GlobalFoundries 22nm

On top of these targets, the design can be parameterized into three evaluation profiles for testing:

1. Spike tandem verification - RTL Simulation and FPGA.
2. Cycle-accurate pipeline visualization - Simulation only.
3. Event counters - RTL Simulation and FPGA.

In the following sections, the different environments and evaluation methods are described in detail.

6.2 Testbench environments

During the development of this project, two main environments have been used. First, an RTL simulation environment can provide a much shorter development cycle. This is due to the fact that synthesizing the design for this environment is much faster and it is also much easier to extract the information needed for testing and evaluation at any given moment. Second, an FPGA environment can be much faster, especially when testing large programs. Evaluating the execution of a program on an FPGA can be orders of magnitude faster than executing it in simulation. However, interfacing with the design and extracting information can be complicated, especially if one needs very detailed and accurate measures. At the later stages of development, we also decided to use an ASIC target in order to understand the quality of the design in terms of timing and area.

One of the main advantages of compiling BSV into Verilog is the ability to use existing downstream tools for different targets. In figure 6.1 we show the compile and synthesis flow of the design. Table 6.1 shows a summary of the specific releases used for every tool. The main design is specified in BSV and uses libraries from BSC. Those export BSV interfaces but can internally have direct Verilog implementations for better efficiency and compatibility with downstream tools. After obtaining the Verilog code, we can follow one of three paths:

1. For RTL simulation we have used Verilator to compile the design into C++ code. Then, GCC is used to compile the simulation and obtain an executable[1].
2. For FPGA synthesis we use Vivado to synthesize the bitstream for our VCU108 FPGA.
3. For ASIC synthesis, we use Genus Synthesis with a GlobalFoundries 22nm target.

In this process *Fpgamake* is used for the FPGA synthesis [Hicb]. We also use *Buildcache*, a utility that can cache the result of hash commands and reuse them in future compiles [Hica]. This makes the compiling process faster, as certain parts of the design are not be synthesized again unless they have been modified.

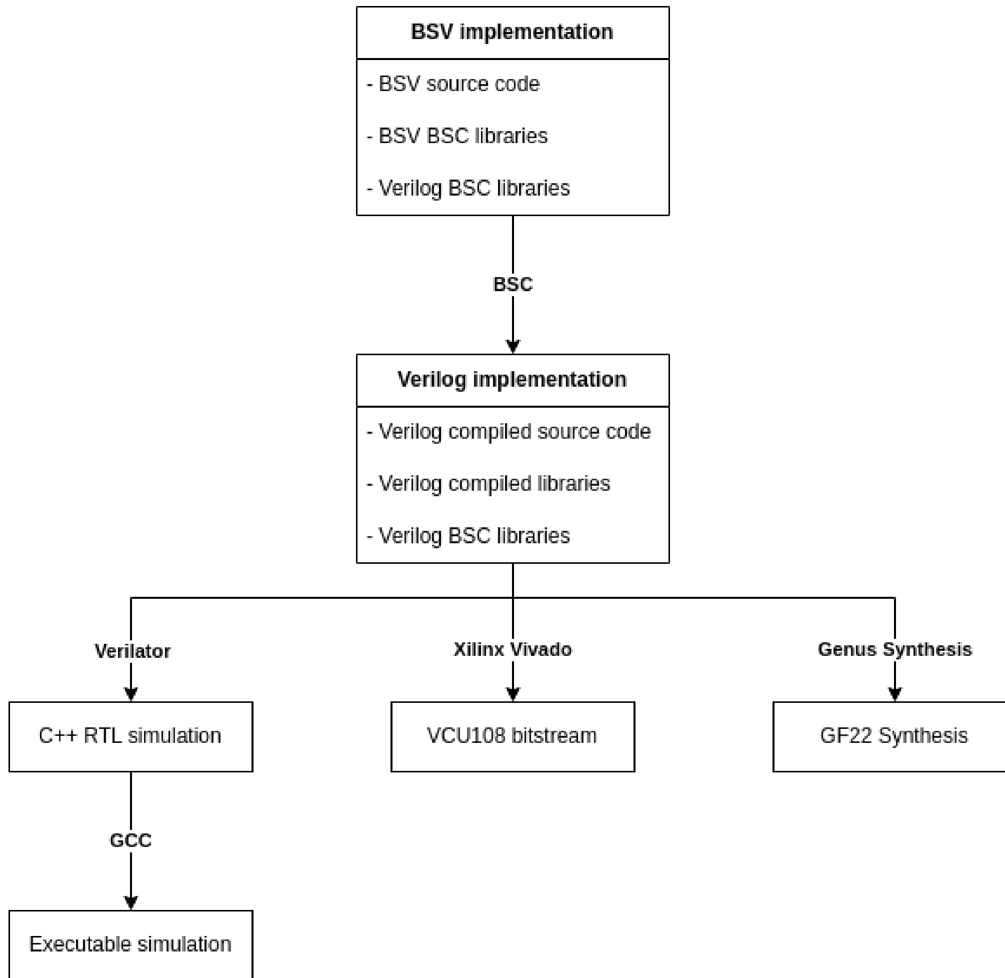


Figure 6.1: Compile and synthesis flow of the design.

Tool	Version - target	Host OS
Bluespec BSC	bsc-2023.01-ubuntu-20.04	Ubuntu 20.04.6 LTS
Verilator	Verilator 4.028 2020-02-06 rev v4.026-92-g890cecc1	Ubuntu 20.04.6 LTS
GNU GCC	(Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0	Ubuntu 20.04.6 LTS
Xilinx Vivado	Vivado v2019.1 (64-bit) - Virtex UltraScale VCU108	Ubuntu 18.04.6 LTS
Cadence Genus	21.10-p002_1 - GlobalFoundries 8 tracks 22nm	Rocky Linux release 8.7

Table 6.1: Comprehensive list of tools and versions used for compilation and synthesis.

We decided to use the interface *Connectal* to bridge the host server's main program and the target design. Connectal provides an interface that can be used on one end by the host server (C++) and at the other end by the target design (BSV). This interface can be used as a message queue to send information to and from the target design running on the FPGA through PCIe [KHA15]. We call these queues *To Host Queues* (THQ). The main advantage of this interface is the easy integration on the FPGA. However, it can also be used in a simulation environment. Thus, to simplify the design, we use it in both. In figure 6.2 we can see a schematic of the testbench environment.

The Connectal interfaces are specified in BSV as two interfaces, depending on the direction of the queue. The interfaces act as a FIFO to send information from/to the target device to/from the host server. When used in the *host* $\rightarrow$ *target* direction, Connectal declares a method in the main C++ program that can be called to send data. On the target side, we must implement a method in the top-level module to receive and use the data. When used in the *host* $\leftarrow$ *target* direction, a method becomes available in the top-level module of the device to collect the data. Then, a method must be defined in the C++ main program to receive and use the data. The actual forwarding of the data is implemented by Connectal and can be transferred and used in the simulation.

Figure 6.2a shows the environment when synthesized for an FPGA target and figure 6.2b when compiled for a simulation target. Most of the implementation both on the target and host side can be reused for both environments, as Connectal makes the interfacing agnostic to the platform. For ASIC synthesis we do not synthesize any of the interfacing components or tools. We only generate a synthesis of the actual device to evaluate the quality of the physical design obtained in terms of timing, area, and power.

The testbench, on the host server side, is composed of three main elements. First, the main host program, written in C++, has access to the target interface and bridges it with the rest of the components. Second, we developed a custom version of *Spike*, the standard RISC-V simulation tool. This is a program that can simulate the expected behavior of a generic RISC-V processor when running any given target program. We use it as a verification method. The *tandem verifier* on the main program compares and verifies the results during verification. The *interpreter* prints the executed instructions in a user-friendly way. Third, P-RISC-TEST contains a test library that can be used to evaluate the target design. These tests are compiled into the target ISA using a RISC-V compiler and run on the evaluated device. This also contains the definition and implementation of the API needed to use P-RISC capabilities in C programs. Software development and tests are further discussed in the following chapter.

When it comes to the three different evaluation environments, most of the design and RTL logic are generic and are inferred and synthesized by the tools. However, there are a few specific components that must be tailored to each target. Those are:

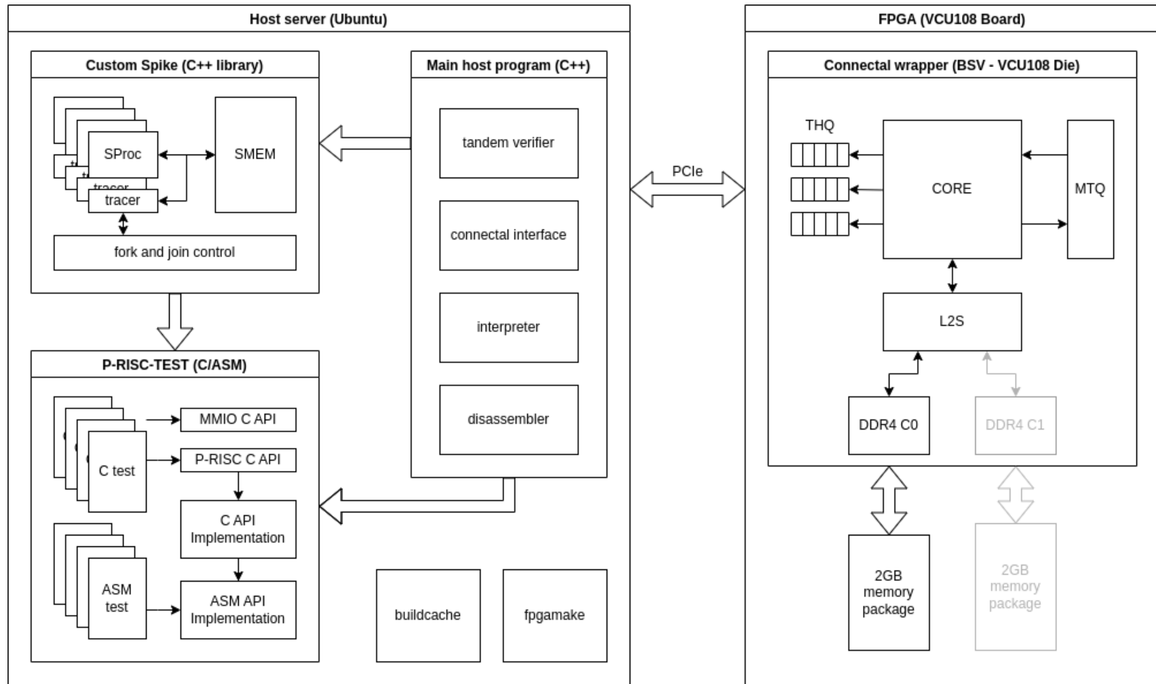
1. Connectal interfaces
2. DDR4 main memory interfaces
3. BRAM and/or SRAM on caches
4. Mul and Div functional units

In the following sections, we explain the specifics of the different targets and their requirements, including the tailored components.

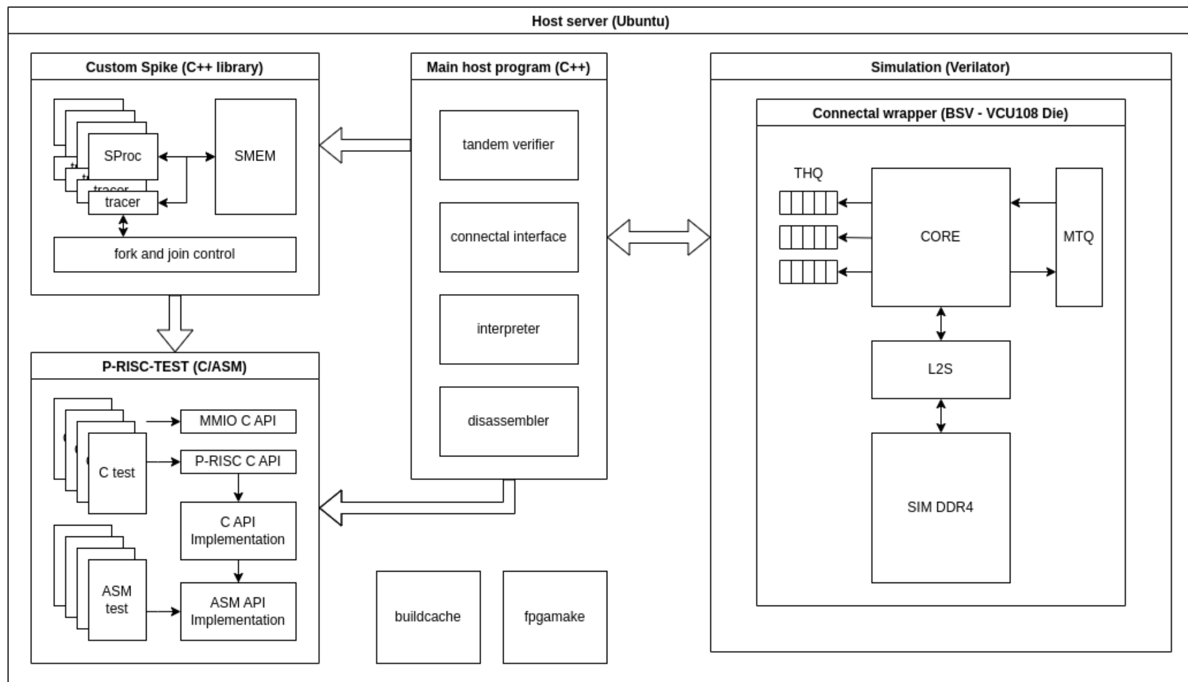
6.2.1 RTL simulation

In simulation, the main memory of the target device is allocated in the memory of the host server. In our case, we are defining a 2GB physical memory space for the device, as we use 32-bit addresses. In the BSV implementation, we have abstracted main memory using the `WideMemDDR4` module. This

[1]The C++ compiler used depends on the architecture and tools available on the host server running the simulation rather than on the design of the project itself or the upstream tools.



(a) FPGA testbench



(b) Simulation testbench

Figure 6.2: Simulation testbench

module exposes a standard BSV interface that is connected to the rest of the system. However, it is parameterized. `WideMemDDR4` infers through environment variables the target synthesis environment and changes the implementation. When compiling for a simulation environment, it instantiates a 2GB register file. This obviously does not make sense for real hardware, but it can be inferred by Verilator and allocated efficiently in the memory of the host server during the execution of the simulation.

The drawback of this method is the speed mismatch of the module. This technique produces a main memory module with a single cycle of latency. In order to make the simulation realistic, we can artificially add latency to the module. This is done by concatenating many FIFOs that form a forwarding pipeline. The length of this pipeline dictates the latency to main memory. This is implemented as a simple `WideMemDelay` which is instantiated in the `WideMemDDR4` module under this environment. The latency can be configured.

The caches of the system use BRAM memory blocks to store the data. These devices have a native single-cycle latency in the real FPGA. BSC defines an abstracted BSV interface for these memory cells. The module implemented is also parameterized and under simulation, this is inferred and simulated by Verilator. We do not need to add any extra logic as this is taken care of by the BSC libraries. Also, all the functional units in the core pipeline, including the Mul and Div, can be specified in high-level combinational logic for simulation.

6.2.2 FPGA synthesis

We have used an AMD Xilinx Virtex UltraScale VCU108 board for FPGA evaluation, although the project should be portable to other boards thanks to the parameterized implementations of the BSV libraries and Connectal components. Table 6.2 shows the specifications of the FPGA. This evaluation board comes with a total of 4GB of DDR4 memory split into two controllers that we use as main memory for our device. As we only require a 2GB physical memory space for our core, we can use a single controller. Although this is the simplest configuration, it could be beneficial to split the memory space between two controllers, shall more memory bandwidth be required.

VCU108 specification	
System Logic Cells (K)	1,176
DSP Slices	768
Memory (Mb)	60.8
GTH 16.3 Gb/s Transceivers	32
GTY 30.5 Gb/s Transceivers	32
I/O Pins	832

Table 6.2: AMD Virtex UltraScale FPGA VCU108 specification [Xil].

The VCU108 used for testing is connected to the host server as a PCIe card, as we saw in figure 6.2a. Connectal interprets the communication interfaces defined and implements a bridge through the PCIe connection. This way, the differences in physical implementation between RTL simulation and FPGA simulation are abstracted by the queuing interface and managed by Connectal. We do not need to make any explicit changes in our design to adapt the interfaces to an FPGA environment, they can be used in this environment without any extra requirements.

In order to use the on-board DDR4 memory as our main memory we use the parameterized `WideMemDDR4` module. Under this environment, this module instantiates the main memory controllers. The top-level module of the project must export the memory bus pins and pass them through to the controller. In listing 6.1 we show a snippet of the report obtained after synthesizing an instance of the core. In particular, the `ddr4_wrapper` is shown. This module encloses the memory controller. We can

observe that it is a fairly expensive module in relation to the core itself. It should be noted that the core itself does not include the memory controller but it does include the first level of instruction and data cache - 16KB and 32KB of BRAM respectively.

The BRAM module instantiated to use on the cache systems of the core is inferred by Vivado and it decides how to map it to the actual BRAM blocks available in this specific board. In listing 6.1 we also can observe the instance of the BRAM modules for the two caches. The data cache uses `RAMB36` blocks for the data and `RAMB18` blocks for the tags and metadata. The instruction cache uses both `RAMB36` and `RAMB18` blocks for data and only `RAMB18` blocks for the tags and metadata. This inference is done automatically by the build system. Notice that the `mkCore7SS` entry corresponds to the main core instance and includes the L1I and L1D. The BRAM blocks that are reported to be in use by that module are actually the ones from the caches.

Most functional units of the pipeline can be implemented using high-level combinational logic and synthesized into logic LUTs. However, multiplication and, especially, division units can be complicated to implement efficiently. Xilinx provides proprietary IP with an efficient implementation of these modules for their boards. We can include the configuration files needed for the project and instantiate them using another parameterized wrapper module. This same module instantiates them as combinational logic for other targets. The Xilinx IP modules have configurable latency. For our targeted speed, we required 2-cycle multiplication and 8-cycle division modules. Listing 6.1 also shows their instances. The multiplication modules use DSP48 blocks to perform the operations while the division blocks use a combination of SRL blocks and logic LUTs. They both use FFs blocks to store partial results. We can observe that the cost of the division units is quite high. Notice here that the resources used by these units are also reported in the `mkCore7SS` entry.

Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	URAM	DSP48 Block
ddr4_wrapper	13894	13287	556	51	17272	25	1	0	3
mkCore7SS	76985	76767	72	146	68981	23	5	0	24
dataCache_data	1083	1083	0	0	0	16	0	0	0
dataCache_meta	52	52	0	0	0	0	1	0	0
dataCache_tags	1	1	0	0	0	0	1	0	0
instCache_data	0	0	0	0	0	7	1	0	0
instCache_meta	0	0	0	0	0	0	1	0	0
instCache_tags	0	0	0	0	0	0	1	0	0
mkMul_xdcDup_1	168	168	0	0	237	0	0	0	12
mkMul	199	199	0	0	237	0	0	0	12
mkDiv_xdcDup_1	1462	1389	0	73	835	0	0	0	0
mkDiv	1430	1357	0	73	835	0	0	0	0

Listing 6.1: Resource usage report from VCU108 synthesis. The snippet shows the total cell usage for the memory controller, the core module, the instruction and data caches, and instances of the multiplication and division units. This synthesis of the core was configured with a 16KB instruction cache and a 32KB data cache. The core has 2 arithmetic lanes and it instantiates two mkMul and mkDiv modules for them. The modules use the Xilinx IP implementation configured with 2 and 8 cycles of latency respectively. Notice that the first-level caches and the multiplication and division units are included in the core. The resource usage reported by the core entry includes those as well.

6.2.3 ASIC synthesis

The main goal of synthesizing this project in an ASIC environment is to evaluate the quality of the design and the implementation of the core. We are not trying to evaluate the quality of the second

level of caching nor the DDR4 controller, as this was never the focus of this work. Thus, for ASIC synthesis purposes, we only synthesize the SMT core itself - including the first level of caching - but not the whole hardware system. We do not use any of the testbench profiles described here, as we are trying to get as close as possible to a real synthesis, and having extra instrumentation aimed toward verification and evaluation would affect the synthesis. We did however run specific synthesis on some of the components - namely the arbiter module - that have become critical in terms of area and timing in order to better understand the quality of the different implementations and solutions.

We have used a GlobalFoundries 22nm library with an 8-track configuration. The synthesis has been performed using Cadence Genus. Table 6.1 also lists the exact version of that tool. Listing 6.3 shows the constraint file and listing 6.2 shows the exact corner conditions. As can be seen, we targeted a period of 900ns under a corner with typical conditions; which was achieved after improving a few of the critical paths. We discuss the full evaluation and results in a later section.

```
####
# opcond
####
create_opcond -name op_cond_typ -voltage 0.8 -temperature 25

####
# timing_cond
####
create_timing_condition -name timing_cond_typ -opcond op_cond_typ -library_sets {typ_25C}

##############################################################################
## typical RC corners defined for setup analysis
##############################################################################

create_rc_corner -name nominal_025C \
   -qrc_tech /technos/GF22FDX/PDK/PEX/QRC/10M_2Mx_4Cx_2Bx_2Jx_LBthick/nominal/qrcTechFile \
   -pre_route_res 1.0 \
   -pre_route_cap 1.0 \
   -pre_route_clock_res 1.0 \
   -pre_route_clock_cap 1.0 \
   -post_route_res "1.0 1.0 1.0" \
   -post_route_cap "1.0 1.0 1.0" \
   -post_route_clock_res "1.0 1.0 1.0" \
   -post_route_clock_cap "1.0 1.0 1.0" \
   -post_route_cross_cap "1.0 1.0 1.0"

####
# Delay corner
####
create_delay_corner -name delay_corner_typ -timing_condition timing_cond_typ \
-rc_corner nominal_025C

####
# Constraints
####
create_constraint_mode -name _default_constraint_mode_ -sdc_files {./genus_scripts/chip.sdc}

####
# Analysis views
####
create_analysis_view -name view_typ  -constraint_mode _default_constraint_mode_ \
-delay_corner delay_corner_typ

set_analysis_view -setup {view_typ}
```

Listing 6.2: Corner conditions used in the Genus synthesis

```
#set_units -time ns

# Set the current design
current_design top
create_clock -name "CLK" -period 900 [get_ports CLK]

set_input_delay -clock CLK 0.05 [all_inputs]
set_output_delay -clock CLK 0.05 [all_outputs]

#current_design top

set_max_fanout 5.000 [current_design]
set_max_transition 0.2 [current_design]

set_load 0.5 [all_outputs]
set_input_transition 0.2 [all_inputs]
```

Listing 6.3: SDC constraint file used in the Genus synthesis

Some of the components that need to be tailored to the environment are not present in this synthesis. That is the case with the Connectal interfaces and the main memory wrapper. The Connectal interfaces are not instantiated when compiling the BSV design for this target, as we synthesize without any testbench profile. The main memory wrapper and controller are not part of the core itself and are not included in this synthesis either. Finally, the second level of caching is also not considered, although the first level is and requires some modifications.

The first level of caching is composed of the `L1I` module and the `L1D` module, as discussed in the previous chapters. Those modules internally instantiate the `BareInstCache` and `BareDataCache` modules respectively. All of these modules are implemented using high-level BSV. They use the BRAM Server modules from BSC to store the actual data. In both cases, the memory is split into three modules. The `dataArray` is the largest one, storing the actual memory lines of 512 bits each. The `tagsArray` stores the tags for those lines of 26 bits each. Finally, `metaArray` holds the metadata about that particular entry, with only 1 or 2 bits respectively for the instruction and data caches. For the synthesis target, the BRAM Verilog implementation provided by BSC has been modified to become parameterized and instantiate SRAM modules under synthesis conditions.

The SRAM macros have been pre-generated in several sizes. The parameterized module instantiates the corresponding SRAM if the size is matched. Otherwise, it instantiates a flip-flop-based implementation. We decided to synthesize with cache sizes of 16KB and 32KB. This requires SRAM blocks of 256 and 512 entries respectively. With that many entries, the library does not provide SRAM blocks with 512-bit entries. Therefore, the parameterized module actually splits the data arrays into 8 SRAM blocks with 64-bit entries. This decision is fully transparent to the user and does not affect the interface nor the behavior observed. For the tag arrays, we generated SRAM blocks directly with 26-bit entries. Finally, we considered that the metadata arrays were too small to benefit from SRAM blocks and were kept as generic flip-flop-based implementations.

The multiplication and, especially, the division modules present in processors are usually some of the critical functional units in terms of area, power, and timing. Implementing an efficient design is not trivial. Our case is no different. So far we have discussed the use of a parameterized module that discriminates between simulation and FPGA environments. Under simulation, it instantiates a naive high-level implementation and under the FPGA environment, it instantiates proprietary Xilinx modules. The Xilinx IP cannot be ported into this environment and the high-level implementation yields a poor design that does not meet our corner conditions. As this is not the focus of this work, we decided to skip the multiplication and division units and leave them as black boxes. Any relatively good implementation of those units should be able to meet the corner conditions. Of course, skipping those modules has an effect on the total area. Therefore, one must keep in mind that the area results shown in this work for the core synthesis do not include these two units.

6.3 Testbench profiles

The testbench for this project may be set up with different profiles. Those profiles configure the implementation of the device on top of the different environments - although not all profiles can be used in all environments. The profiles described here can be activated by means of setting environment variables that are inferred by the different modules during the prepossessing of the implementation. The main purpose of them is to modify how information is extracted from the device being tested. These profiles apply mainly to the RTL simulation and FPGA environments. For ASIC, we only synthesize the core, mimicking that of a real device, without any means of actually extracting information from it at runtime. In the following sections, we describe the main profiles in detail.

6.3.1 Tandem verification

One of the key parts of hardware design is verification. In this section, we discuss the verification strategy that we followed in this work. The goal of this part of the project is to implement a mechanism that detects and reports implementation and/or design errors in our device. We use this to check that our device is behaving as expected. In this section, we will use the term *error* specifically to refer to behaviors that could be observed in the device that are considered incorrect and should be reported by the verification system so that they can be fixed. Our hardware implementation will only be considered correct when no errors are detected by the verification system. By the end of the project, our device has been proven by this method to execute several tests with over 400 million instructions each without any errors.

We use a co-simulation approach with tandem verification. Our strategy consists of running the same test program on both the *Device Under Test* (DUT) and a custom Spike instance side by side and comparing the results. We monitor the behavior of both devices as they execute the target program. When the DUT behaves differently than Spike, an error is reported. This verification method can be used in both FPGA and RTL simulation environments for this project. However, especially when running large programs on the target device, it is much faster to run on FPGA.

We run the verification on a per-instruction basis. Every time the DUT commits an instruction, that information is sent to the host server using the corresponding THQ as a *Commit Report* (CMR). Then, the host program asks the Spike instance to run one step and report the results. Then, the tandem verifier component is used to compare both sides. If they match, the instruction is considered correct and execution continues. If it a mismatch is detected, the simulation stops and the discrepancies are reported using the interpreter on the host program.[2]

Spike is a standard simulation tool for a RISC-V system. It can be configured to run on the 32-bit and 64-bit distributions and standard extensions of the RISC-V ISA family [Sof]. The standard Spike implementation instances a number of simulated processors and a main memory component. Usually, it can also be configured to simulate the use of a simple level of caching for each processor. We call the simulated processor instances *SProc* and the simulated memory *SMEM* to differentiate them from real devices. This simulation is contained within a C++ program. Then, we only need to add the logic necessary to report the results of the executed instructions.

Arithmetic and control-flow instructions are fairly simple to verify, as the only architectural state that they modify is the destination register and the PC, respectively. When the host receives a CMR, it runs one step on the corresponding SProc and reads those two registers. These become the reference values that are checked against the CMR from the DUT. Memory instructions are expected to read or write from or to memory. One can use SMEM tracers to track the memory accesses that they generate and use that as the reference value. These are standard verification techniques that can be used for most designs.

[2]The DUT is actually never stopped, as we need the verification to be as little intrusive as possible. The system always keeps running and pushing reports into the THQ. However, after a mismatch is detected, the host may ignore the following reports and halt the simulation.

We can use the described technique to verify most instructions in our system. However, in P-RISC we introduce two new types of instructions - Fork and Join - and dynamic parallel behavior. This is not considered in standard Spike. We need to customize a few top-level components of the software simulation in order to be able to run and verify actual P-RISC programs. This customized version is compiled as a library and loaded by the main host server program when running tandem verification.

The first step of adapting Spike is adding the new instructions. Adding instructions is fairly easy as long as they are arithmetic or control-flow instructions with regular constraints. In that case, the following steps are needed:

1. Define the arithmetic behaviour of the new instruction at *insns/new_instruction.h*.
2. Add the encoding masks to *riscv/encoding.h*.
3. Add the decoding information to *riscv/opcodes.h*.
4. Define in *riscv/riscv.mk.in* which extensions of the ISA should support this instruction.
5. Add the instruction to the disassembler at *disasm/disasm.cc*.
6. Recompile spike.

These steps are needed for Fork and Join instructions. However, it is not enough, as their behavior differs a lot from the usual arithmetic nature of most instructions. The custom version of Spike needs to be able to host multiple streams of instructions and execute them concurrently. We thought about two possible approaches for this.

One option was trying to emulate the behavior of the actual hardware. That would imply instancing a few SProcs - each matching a Hart from the device - and some structure to emulate the MTQ. Then, most instructions can be simulated normally by the SProc. Fork instructions add elements to the MTQ twin. Whenever a context switch is produced at the DUT, that should be reported and tracked by Spike. This approach could work, however, keeping track of the streams is complicated and it would be tailored to match one specific hardware implementation of P-RISC. It does not scale well when many different implementations need to be verified.

The other option, and the one that we use, is dynamically instantiating new SProcs to match the streams of the program. Each SProc is responsible for simulating a linear stream of instructions, matching the behavior proposed by the execution model. This way, each SProc does not correspond to any hardware feature of any particular implementation and it is agnostic to the scheduling policies of the DUT. One concern with this approach is the danger of instantiating too many SProc objects during simulation. This does not appear to be a problem in practice, as the number of simultaneous streams generated by a program is limited and SProcs are relatively lightweight C++ objects.

To implement this feature, Fork and Join instructions need to be snooped by the top-level module. From a local SProc point of view, Fork instructions do nothing, as they do not modify neither the local state nor the memory. Join instructions only write to memory. Still, the top-level module detects these instructions and acts accordingly. When a Fork instruction is detected, a new SProc is instantiated, added to the system, and attached to memory. When a Join kills a stream, the SProc is removed from the system. The rest of the instructions can execute normally within their respective SProcs. During tandem verification, when CMRs are received from the DUT, they are matched and the corresponding SProc is asked to simulate one step. Then the results are checked. Now the difficulty lies in how to match instruction streams executing on the DUT with SProcs on the simulation.

To solve the stream matching problem, we add a verification component to the DUT. This is a simple device that can be asked to generate a new unique verification identifier (VerifID). Each active stream has a VerifID that is included in the CMRs reported to the host. The host maps VerifIDs into SProc instances. Under this context, when a fork instruction is executed on the DUT, a new VerifID is reserved. When the commit of this fork instruction is reported to the host, the CMR includes that VerifID as a child. Then, Spike instances the new SProc and maps it to the reported identifier. From

now on, Spike expects to receive CMRs tagged as coming from that stream and will forward them to the corresponding SProc.

The Join instructions also require special attention. Executing a Join instruction could cause the corresponding stream to terminate. In that case, the processor being evaluated must not keep executing instructions from that stream. Spike must also consider this mechanism and verify that the device behaves as expected. When Spike simulates an unsuccessful Join instruction, it removes and deletes the corresponding SProc object. From now on, if it receives a CMR associated with that stream, the co-simulation will flag it as incorrect and report the error.

The interpreter component of the main host server program prints the instructions that are being committed by the device in a user-friendly manner. Figure 6.3 shows snippets from a few execution traces. The first ones correspond to executions where errors were detected. The last one corresponds to the end of a correct execution. The golden lines correspond to Spike and the white ones to the DUT. Mismatches are reported in red, alongside relevant information. In that case, co-simulation stops. Notice that, at the beginning of every line, the VerifID is also reported alongside the number of cycles on every device. A disassembly of the instruction is also included. This is done using the standard disassembler that comes with Spike [3].

Tandem verification has been used extensively during the development and implementation of this project to verify that the device always produced correct results, both in simulation and in FPGA environments. It needs to be noticed, however, an inherent flaw of this technique. When executing tandem verification, especially in an FPGA environment, the host side is much slower than the target side. This means that the host cannot consistently keep up we the CMRs that are sent by the DUT and the THQs get full. This generates back-pressure into the core and it stalls. Thus, tandem verification is invasive, as it can stall the DUT. This could potentially alter the results of the test - the DUT behaves correctly when under back-pressure but generates wrong results when it runs freely, at full speed. In short tests, this problem can be overcome by increasing the size of the THQs. This would guarantee that the first few hundreds of instructions have been executed at full speed without back-pressure. However, this only applies to very short tests.

6.3.2 Cycle-accurate pipeline visualization

The second evaluation method is a cycle-accurate visualization of the pipeline. This technique is based on displaying, for every cycle, the state of the different stages and elements from the pipeline. This is crucial to understand, at a very fine grain, what is happening on the pipeline and checking if it is behaving as expected. This can be useful not only to debug potential errors but also to understand how the design behaves under different loads and to find possible modifications that could be implemented to improve performance.

This type of visualization must be very accurate and non-invasive on the system; as we do not want to modify how the device behaves when observed. It also reports a lot of information every cycle. The amount of information required to be reported per cycle is too large to be extracted from an FPGA environment and be reported by a queue-based system at a fast enough pace. Therefore, this visualization has been directly implemented using Verilog display statements that are non-invasive and it is only available in the simulation environment.

We show in figure 6.4 an example portion of a trace obtained using this method of evaluation. Every block represents the state of the machine on one cycle. From top to bottom, we can see a partial trace of five cycles. On the left, we show the total number of instructions and cycles executed until that point. Then, we can observe eight rows. Each shows the information corresponding to one hart of the device. First, this view shows the identifier of the stream that is currently hosted on that hart and the corresponding state. This can be used to track context switches and the scheduling of streams. After that, each column corresponds to one stage of the pipeline, showing which instruction is on flight at

[3] Once the new instructions have been added to Spike, as described, they are also considered in the provided disassembler. We generated a library that reuses that same disassembler and includes it in the main host server program. This way, the host server can print the instructions being executed in a user-friendly output.

```
              cycle:           3874 | pc: 0x00002070 | iType:      Alu | res: 0x00007018 | addi    a3, a3, 4

----------------------- Result mismatch! -----------------------
 --> Spike result: 0xDC1B0E40
 --> Dut result  : 0xC5A476B9
[id:      1 ] cycle:            322 | pc: 0x00002074 | iType:      Alu | res: 0xDC1B0E40 | mul     a4, a4, a1
              cycle:           3884 | pc: 0x00002074 | iType:      Mul | res: 0xC5A476B9 | mul     a4, a4, a1

[id:      0 ] cycle:            323 | pc: 0x00002B28 | iType:    Forkr | res: 0x00002B3C | forkr   a5, 0
              cycle:           3894 | pc: 0x00002B28 | iType:    Forkr | res: 0x00000003 | forkr   a5, 0

----------------------- Result mismatch! -----------------------
 --> Spike result: 0xE0B60640
 --> Dut result  : 0x7776B6B9
[id:      1 ] cycle:            324 | pc: 0x00002078 | iType:      Alu | res: 0xE0B60640 | add     a2, a2, a4
              cycle:           3904 | pc: 0x00002078 | iType:      Alu | res: 0x7776B6B9 | add     a2, a2, a4
```

(a) Spike co-simulation detects incorrect multiplication result.

```
----------------------- Result mismatch! -----------------------
 --> Spike result: 0x1D7A8E10
 --> Dut result  : 0xE28571F0
 --> Spike addr: 0x0000B018
 --> Dut addr  : 0x0000B018
[id:      1 ] cycle:            331 | pc: 0x00002064 | iType:       Ld | res: 0x1D7A8E10 | lw      a4, 0(a5)
              cycle:           3974 | pc: 0x00002064 | iType:       Ld | res: 0xE28571F0 | lw      a4, 0(a5)
```

(b) Spike co-simulation detects incorrect load result.

```
              cycle:          39794 | pc: 0x0000207C | iType:       Br | res: 0x00002064 | bne     a5, a0, pc - 24

[id:      4 ] cycle:           3915 | pc: 0x00002060 | iType:      Alu | res: 0x0000B010 | mv      a5, t3
              cycle:          39794 | pc: 0x00002060 | iType:      Alu | res: 0x0000B010 | mv      a5, t3

----------------------- Unexpected commit from verifID: 0 -----------------------
[id:      5 ] cycle:           3916 | pc: 0x00002064 | iType:       Ld | res: 0x59FA0B4D | lw      a4, 0(a5)
              cycle:          39834 | pc: 0x00002064 | iType:       Ld | res: 0x59FA0B4D | lw      a4, 0(a5)

[id:      7 ] cycle:           3917 | pc: 0x00002064 | iType:       Ld | res: 0x1090EB6D | lw      a4, 0(a5)
              cycle:          39844 | pc: 0x00002064 | iType:       Ld | res: 0x1090EB6D | lw      a4, 0(a5)

[id:      1 ] cycle:           3918 | pc: 0x00002064 | iType:       Ld | res: 0x6E711BE7 | lw      a4, 0(a5)
```

(c) Spike co-simulation detects unexpected commit from a terminated stream.

```
[id:      3 ] cycle:          71677 | pc: 0x0000206C | iType:      Alu | res: 0x0000B19C | addi    a5, a5, 4
              cycle:         717414 | pc: 0x0000206C | iType:      Alu | res: 0x0000B19C | addi    a5, a5, 4

[id:      6 ] cycle:          71678 | pc: 0x0000207C | iType:       Br | res: 0x00002064 | bne     a5, a0, pc - 24
              cycle:         717444 | pc: 0x0000207C | iType:       Br | res: 0x00002064 | bne     a5, a0, pc - 24



---------- STATS ----------

[id:      0 ] commits:            764
[id:      1 ] commits:           8852
[id:      2 ] commits:           8914
[id:      3 ] commits:           8917
[id:      4 ] commits:           8921
[id:      5 ] commits:           8933
[id:      6 ] commits:           8963
[id:      7 ] commits:           8961
[id:      8 ] commits:           8454
[id:      9 ] commits:              0
[id:     10 ] commits:              0
[id:     11 ] commits:              0
[id:     12 ] commits:              0
[id:     13 ] commits:              0
[id:     14 ] commits:              0
[id:     15 ] commits:              0
[id:     16 ] commits:              0

        total commits:          71679

----------------------------
```

(d) Spike co-simulation ends after correct execution and reports the number of commits for each stream.

Figure 6.3: Spike co-simulation traces. The traces show, for every commit instruction, the verification ID of the corresponding stream, the simulation cycle, the program counter, and the instruction type, result, and disassembly. The golden lines are the results produced by Spike and the white lines are the results produced by the DUT. After a correct simulation, the number of total commits is shown for every stream generated.

that stage. In decode stage, we can observe which instructions are ready to be scheduled and which are stalled waiting on operands. In the arbiter stage, we can also observe both what instructions are available for execution and which ones are being selected. Finally, it also displays a disassembly of the instructions being committed. This can be used when trying to analyze the execution of a particular trace of instructions.

```
--------------------------------------------------------------------------------------------------------
  54311 ||   7 || Full  h | F 0x00002074 | D 0x00002070 | A 0x00002068 |              | M 0x00002064 |              |
  83033 ||   6 || Full  m | F 0x00002080 | D 0x0000207c | A 0x00002078 |              |              |              |
        ||   5 || Full  h | F 0x00002084 | D 0x00002080 | A            |              | M 0x0000207c | C 0x00002078 | add r12 = r12 r14
        ||   4 || Full  h | F 0x00002074 | D 0x00002070 | A 0x00002068 | E 0x00002064 |              |              |
        ||   3 || Full  m | F 0x00002080 | D            | A 0x00002078 |              |              |              |
        ||   2 || Full  h | F 0x00002068 | D 0x00002064 | A            |              |              | C 0x00002080 |
        ||   1 || Full  h | F 0x0000207c | D 0x00002078 | A            |              |              | C 0x00002074 | mul r14 = r14 r11
        ||   8 || Full  h | F 0x0000207c | D 0x00002078 | A 0x00002074 | E 0x00002070 | M 0x0000206c |              |
--------------------------------------------------------------------------------------------------------
  54312 ||   7 || Full  h | F 0x00002074 | D 0x00002070 | A 0x0000206c | E 0x00002068 |              | C 0x00002064 | lw r14 = [r15 0x0]
  83035 ||   6 || Full  m | F 0x00002080 | D            | A 0x0000207c | E 0x00002078 |              |              |
        ||   5 || Full  h | F 0x00002084 | D 0x00002080 | A            |              |              | C 0x0000207c | bne r15 r10 0xffffffe8
        ||   4 || Full  h | F 0x00002074 | D 0x00002070 | A 0x00002068 |              | M 0x00002064 |              |
        ||   3 || Full  m | F 0x00002080 | D            | A 0x0000207c | E 0x00002078 |              |              |
        ||   2 || Full  h | F 0x0000206c | D 0x00002068 | A 0x00002064 |              |              |              |
        ||   1 || Full  h | F 0x0000207c | D 0x00002078 | A            |              |              |              |
        ||   8 || Full  h | F 0x0000207c | D 0x00002078 | A 0x00002074 |              | M 0x00002070 | C 0x0000206c | addi r15 = r15 0x4
--------------------------------------------------------------------------------------------------------
  54313 ||   7 || Full  h | F 0x00002078 | D 0x00002074 | A 0x0000206c |              | M 0x00002068 |              |
  83037 ||   6 || Full  m | F 0x00002080 | D            | A            | E 0x0000207c | M 0x00002078 |              |
        ||   5 || Full  h | F 0x00002088 | D 0x00002084 | A 0x00002080 |              |              |              |
        ||   4 || Full  h | F 0x00002074 | D 0x00002070 | A 0x00002068 |              |              | C 0x00002064 | lw r14 = [r15 0x0]
        ||   3 || Full  h | F 0x00002080 | D            | A            | E 0x0000207c | M 0x00002078 |              |
        ||   2 || Full  h | F 0x00002070 | D 0x0000206c | A 0x00002068 | E 0x00002064 |              |              |
        ||   1 || Full  m | F 0x00002080 | D 0x0000207c | A 0x00002078 |              |              |              |
        ||   8 || Full  h | F 0x0000207c | D 0x00002078 | A 0x00002074 |              |              | C 0x00002070 | addi r13 = r13 0x4
--------------------------------------------------------------------------------------------------------
  54314 ||   7 || Full  h | F 0x00002068 | D 0x00002074 | A 0x0000206c |              |              | C 0x00002068 |
  83039 ||   6 || Full  h | F 0x00002080 | D            | A            |              | M 0x0000207c | C 0x00002078 | add r12 = r12 r14
        ||   5 || Full  m | F 0x00002064 | D 0x00002084 | A 0x00002080 |              |              |              |
        ||   4 || Full  h | F 0x00002074 | D 0x00002070 | A 0x0000206c | E 0x00002068 |              |              |
        ||   3 || Full  h | F 0x00002084 | D 0x00002080 | A            |              | M 0x0000207c | C 0x00002078 | add r12 = r12 r14
        ||   2 || Full  h | F 0x00002074 | D 0x00002070 | A 0x00002068 |              | M 0x00002064 |              |
        ||   1 || Full  m | F 0x00002080 | D            | A 0x0000207c | E 0x00002078 |              |              |
        ||   8 || Full  h | F 0x0000207c | D 0x00002078 | A            | E 0x00002074 |              |              |
--------------------------------------------------------------------------------------------------------
  54315 ||   7 || Full  h | F 0x0000206c | D 0x00002068 | A 0x00002070 |              |              |              |
  83042 ||   6 || Full  h | F 0x00002084 | D 0x00002080 | A            |              |              | C 0x0000207c | bne r15 r10 0xffffffe8
        ||   5 || Full  m | F 0x00002064 | D            | A            |              |              |              |
        ||   4 || Full  h | F 0x00002078 | D 0x00002074 | A 0x00002070 | E 0x0000206c | M 0x00002068 |              |
        ||   3 || Full  h | F 0x00002084 | D 0x00002080 | A            |              |              | C 0x0000207c | bne r15 r10 0xffffffe8
        ||   2 || Full  h | F 0x00002074 | D 0x00002070 | A 0x0000206c | E 0x00002068 |              | C 0x00002064 | lw r14 = [r15 0x0]
        ||   1 || Full  m | F 0x00002080 | D            | A            | E 0x0000207c | M 0x00002078 |              |
        ||   8 || Full  h | F 0x0000207c | D 0x00002078 | A            |              | M 0x00002074 |              |
--------------------------------------------------------------------------------------------------------
```

Figure 6.4: Example of a cycle-accurate trace when executing several streams on 8 harts. In the D stage, a grey entry corresponds to an instruction that is already in that stage but cannot yet move forward because it is waiting on operands. In the A stage, a grey entry corresponds to an instruction that was ready to be executed but has not been scheduled by the arbiter in this cycle. The highlighted instructions in the very last column are the ones being committed. A grey entry here depicts a memory instruction that was supposed to commit but cannot do so due to a memory miss.

6.3.3 Event counters

The previous method is useful for very fine-grained analysis, however, we need something to understand the behavior and measure the performance when executing larger portions of code in our machine. We do this with event counters. In different modules of the system, we add counters that count the occurrences of different events - for instance, a cache miss or a branch mispredict. These counters are implemented using registers that are incremented when such an event is detected. These counters must be implemented carefully, as a wrong design could affect the behavior of the system under BSV.[4]

Once the event counters have been implemented, we must decide on a method to display them. In other words, we need to be able to create a snapshot of the event counters when needed by the target program and report that to the host. We decided to implement that using MMIO addresses in the

[4]BSC always guarantees a physical implementation that maps to sequential execution of the different rules within one cycle. Writing to or reading from wires, registers, and Ehrs can add constraints to the scheduling of the corresponding BSV rules. Counters must be implemented carefully and making sure that they do not affect the scheduling or behavior of the rules.

user memory space of the device being tested. We define a few special addresses that, when written to, trigger a special side effect. When such store instruction is detected, the value of the different counters is read and placed into a message. This message is sent to the host side using the corresponding THQ and the interpreter from the main host server program displays the results. To make these calls accessible to target user code written in C, we have added a very simple and lightweight MMIO API that holds a few functions that access the MMIO devices.

The user-triggered approach is very useful when trying to study the performance of an algorithm. When we need to study performance on a specific region of interest of the program, we can place MMIO stores before and after that piece of code using the API. This triggers the send of two messages. One can then observe how the different counters have increased during the execution of the region of interest. In our implementation, we use the stored value as a message that is also forwarded to the host together with the stream identifier. This easily identifies the different output blocks produced and displays extra information on self-checking tests.

We use the counters defined here extensively in the evaluation section in order to measure the performance of different algorithms. This can be used to understand the pressure that a given algorithm puts on the different resources of the device and decide whether they might become a bottleneck of the system. The following is a comprehensive list of the main MMIO addresses of our device and the event counters mapped to them.

1. Simple message: Report the VerifID of the caller as well as a user-defined message and the current cycle and instruction count. Available in text and hexadecimal variants.

2. Memory statistics: Report the count and hit rate for the different cache levels alongside a simple message. Specifically, it reports:

   (a) L1I: Number of accesses and hit rate.

   (b) L1D: Number of accesses and hit rate divided by Load, Store, and Join instructions.

   (c) L2S: Number of accesses and hit rate and number of write-backs to main memory.

3. Core statistics: Reports a distribution on the number of instructions reaching each relevant stage of the pipeline and the causes of their stalls, if any. It reports the frequency with which n instructions reach that stage in a given cycle and can be forwarded for n ranging from zero to the maximum core width. It specifically count:

   (a) Hart: Number of harts in use.

   (b) Fetch: Number of instructions that were fetched. This could be reduced due to L0I misses.

   (c) Decode: Number of instructions that were decoded. This could be reduced due to filtering wrong-path instructions, hart locks due to data cache misses, and stalls due to data hazards through the register file.

   (d) Arbiter: Number of arithmetic and memory instructions reaching the arbiter stage. This could be reduced due to filtering wrong-path instructions.

   (e) Commit: Number of instructions that were committed. This could also be reduced due to filtering wrong-path instructions. Furthermore, recall that the backend stages - arbiter output to commit stage - are narrower than the frontend stages - fetch to arbiter input - and the maximum throughput here is limited by that width.

6.3.4 Memory subsystem testing

Testing the P-RISC system on an FPGA adds a degree of non-determinism due to the memory system performance. The latency and memory throughput offered by the DDR4 system is not deterministic to the core and depends on the specific board being used. Furthermore, the DDR4 subsystem and the main FPGA subsystem are in different clock domains. This means that the memory performance observed by the core depends on the clock speed at which it is synthesized and executed.

In order to better comprehend the performance of the DDR4 subsystem in relation to the main clock domain, we added a `WideMemTester` component to the implementation. `WideMemTester` tests the memory system at a hardware level before starting the actual core. This test skips the cache hierarchy and directly probes into the memory controller module. The specific tests are launched from the host server through the corresponding THQ. The results are reported back to and displayed by the host server.

We implemented five types of tests in the `WideMemTester`:

1. RDLAT (Read latency): The test module issues a series of read requests one by one. It measured the request-response latency.
2. MXLAT (Mixed latency): The test module issues a series of write-then-read request pairs one by one. It measures the request-response latency.
3. RDTHP (Read throughput): The test module issues a series of read requests back to back. On one side, it measures the length of the time window required for the DDR4 controller to accept the requests. On the other side, it measures the length of the time window required to receive the different responses.
4. WRTHP (Write throughput): The test module issues a series of write requests back to back. It measures the length of the time window required for the DDR4 controller to accept the requests.
5. MXTHP (Mixed throughput): The test module issues a series of write-then-read request pairs back to back. On one side, it measures the length of the time window required for the DDR4 controller to accept the requests. On the other side, it measures the length of the time window required to receive the different responses.

The host server can request any number of tests of any type before starting the actual execution of the target program. Every test can be configured through three parameters. First of all, the test type - as described above. Second, the test length. This decides the length of the series of requests sent. This parameter is especially relevant in throughput tests. Third, the test stride. On consecutive accesses, the address is increased. The stride indicates the address increase in every access. A test configured with a stride of zero issues all the requests to the same address.

In figure 6.5 we show the results obtained when running the FPGA with a 62.5MHz main clock. Sub-figure 6.5a shows the latency results and sub-figure 6.5b shows the throughput results. The latency reported is the sum of latency for the individual access. We can extract the following conclusions:

1. RDLAT (Read latency): The average latency for the three strides tested is in the range of 17 to 18 core clock cycles. We observe no meaningful difference depending on the stride.
2. MXLAT (Mixed latency): The average latency for the three strides tested is in the range of 18 to 19 core clock cycles. This latency is marginally higher than in the pure read test. We observe no meaningful difference depending on the stride.
3. RDTHP (Read throughput): The length of both the transmit and receive windows are in the 8 to 9 core clock cycles per request. This suggests a throughput of about 0.12 transfers per cycle. The similar window length on both sides indicates that the test is long enough to saturate any buffering artifact. We observe no meaningful difference depending on the stride.
4. WRTHP (Write throughput): The length of the transmit window is exactly 1 cycle per request. This indicates that the limiting factor in write throughput is currently the system itself, which only can emit a single request per cycle.
5. MXTHP (Mixed throughput): The length of both the transmit and receive windows are again in the 8 to 9 core clock cycles per request. The throughput with write-then-read request pairs is very close to that of the pure read test.

The tests implemented here are by no means exhaustive and do not offer a deep understanding of the DDR4 subsystem performance. However, this was never the goal as this work does not focus on that. The goal was only to give a basic reference frame for the performance that could be expected in this project. This has implications for the caching system required as the caching system both hides the latency to main memory and reduces the pressure put on it.

We can conclude that the latency expected from main memory is 17 to 18 cycles as observed by the caching system. The throughput that the DDR4 system comfortably handles is about 1 request every 10 clock cycles. Those numbers are not too restrictive. It should be noted that the relative performance of the DDR4 memory decreases proportionally if the frequency of the main clock domain is increased. Also, on an ASIC implementation, the expected main clock frequency would be orders of magnitude higher and, thus, the relative performance of main memory lower. Therefore, in an ASIC implementation, the second - and even third - level of caching will probably become more relevant.

The performance results obtained in the software test are of course affected by the relative speed of the DDR4 memory. In order to better emulate the performance that one could expect on an ASIC-based system we need to artificially throttle the DDR4 system. As explained earlier in this work, this is done by means of artificially adding a delay pipeline to the memory system. Effectively adding latency to the DDR4 memory and mimicking a higher synthesis speed for the main clock domain and the caching system.

```
-------------------------------------------------------------------------------------------
TST     | type :          RDLAT | len   :            100 | str   :                  0 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1747 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          RDLAT | len   :            100 | str   :                  1 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1777 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          RDLAT | len   :            100 | str   :                 16 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1732 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
TST     | type :          MXLAT | len   :            100 | str   :                  0 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1861 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          MXLAT | len   :            100 | str   :                  1 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1873 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          MXLAT | len   :            100 | str   :                 16 |
        |-----------------------|------------------------|---------------------------|
        | lat  :           1859 | delTX:               0 | delRX:                  0 |
-------------------------------------------------------------------------------------------
```

(a) Latency performance tests with a stride of 0, 1, and 16.

```
-------------------------------------------------------------------------------------------
TST     | type :          RDTHP | len   :          10000 | str   :                  0 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           81232 | delRX:              81244 |
-------------------------------------------------------------------------------------------
TST     | type :          RDTHP | len   :          10000 | str   :                  1 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           84012 | delRX:              84026 |
-------------------------------------------------------------------------------------------
TST     | type :          RDTHP | len   :          10000 | str   :                 16 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           82566 | delRX:              82578 |
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
TST     | type :          WRTHP | len   :          10000 | str   :                  0 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           10000 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          WRTHP | len   :          10000 | str   :                  1 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           10000 | delRX:                  0 |
-------------------------------------------------------------------------------------------
TST     | type :          WRTHP | len   :          10000 | str   :                 16 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           10000 | delRX:                  0 |
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
TST     | type :          MXTHP | len   :          10000 | str   :                  0 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           82613 | delRX:              82621 |
-------------------------------------------------------------------------------------------
TST     | type :          MXTHP | len   :          10000 | str   :                  1 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           83070 | delRX:              83080 |
-------------------------------------------------------------------------------------------
TST     | type :          MXTHP | len   :          10000 | str   :                 16 |
        |-----------------------|------------------------|---------------------------|
        | lat  :              0 | delTX:           82618 | delRX:              82626 |
-------------------------------------------------------------------------------------------
```

(b) Throughput performance tests with a stride of 0, 1, and 16.

Figure 6.5: Performance of the DDR4 subsystem on the VCU108 evaluation board. The tests were conducted using a single - instead of dual - memory controller and domain. For this test, the performance is measured in cycles from the main clock domain - where the P-RISC core is found. The main clock domain had a 62.5MHz clock and the DDR4 controller used a 300MHz system clock.

Chapter 7

Software stack and benchmarks

In this chapter, we discuss a simple software stack developed for the P-RISC architecture. The software stack is not the most comprehensive and implements a limited fork-join execution paradigm. We believe however that more expressive task-based execution models could be also implemented using the architecture developed in this work - provided the necessary synchronization primitives. The software stack shown here is composed of a low-level interface that is built on top of and respects the C ABI for RISC-V and a higher-level API that exports more expressive methods to the user. Finally, we explain how we used this software stack to extract parallelism from a few common algorithms. Those implementations are later used as benchmarks to test the final performance of this system.

7.1 C ABI and ISA extension

The first step in order to develop software for P-RISC is to define the software-hardware interface. That is, defining the ISA extension and encoding for the instructions implementing the fork and join operations. In this section, we discuss the ISA modifications made to the base RISC-V ISA and the lowest level C abstraction implementing a parent-child parallel execution model. One of the reasons we decided to use RISC-V as the base for our ISA is the modularity that it provides. The base ISAs only define the fundamental instructions. Other instructions are included as extensions that not all pieces of hardware must implement. The RISC-V ISAs provide a few opcodes that are left available for the definition of custom instructions. RISC-V provides base ISAs for 32-bit and 64-bit machines. A 128-bit ISA will be defined in a future release.

At the beginning of the project, we made the decision to base our system in the RV32I ISA[1]. The reasoning behind that choice was that the RV32 hardware state is roughly half of that of the RV64. At that point, we believed that one of the challenges when building a P-RISC system would be allocating and managing those. We believed that the nimbler architectural state would make for a simpler initial implementation. However, RV32 is not mainly targeted toward high-performance devices, but rather towards embedded devices. RV64 should be the base ISA used for high-performance machines. Although we stuck with RV32 for this work, we do believe that RV64 will probably be a better choice for future projects that try to implement a general-purpose machine; and that the findings and results from this work should translate reasonably well into that context - Albeit all the token-managing modules like the MTQ would be more resource intensive.

On top of the base ISAs, RISC-V also defines a standard *Application Binary Interface* (ABI). The ABI does not define any characteristics of the hardware itself or the software-hardware interface, but rather the basic interface for pieces of software. Fundamentally, the job of the ABI is to standardize the use of registers by the software and to define the subroutine calling convention. If a RISC-V machine

[1] We also implement the standard extension for Integer Multiplication and Division. Our base ISA is actually RV32IM and not just RV32I.

is used with fully custom software, the ABI does not bear much relevance. However, building software that respects the ABI guaranteed compatibility with existing compilers, libraries, and other software environments built for that ISA.

For P-RISC, we want to be able to define software that respects that ABI. Fully custom pieces of software are also an option. However, we believe that respecting the ABI eases the development of efficient parallel software. Specifically, we would like to be able to directly "call" ABI methods in parallel using fork instructions and then collect their result using join instructions. This means that we should be able to develop and parallelize C code easily. This idea defines a parent-child parallel execution model. The main program offloads subtasks into streams created by fork instructions and waits for their results when they are needed using join instructions. This execution paradigm is not the most refined. Many frameworks - like OpenMP or Cilk - have moved into task-based paradigms that can expose more parallelism. We discuss those latter in this text as they are more complicated to implement and our machine might not have the synchronization primitives needed for an efficient implementation. Let us now discuss the requirements of the parent-child calling convention.

Figure 7.1 shows the register usage for the C calling convention. This ABI calls subroutines by jumping into them usually with JAL or JALR instructions. Those instructions also store the return address into the `ra` register; to which the subroutine returns at the end of the execution. Registers `a0-a7` are used to pass the subroutine arguments and `a0-a1` are reused at the end to store the return value. Our calling convention uses the same registers. However, there is an important difference. Under the subroutine calling convention, the subroutine code path uses the same software stack as the caller and pointed to by the `sp` register. In P-RISC, subroutines "called" with Fork instructions execute in parallel to the calling routine. Thus, they cannot pile on top of the same stack. A stack for the subroutine must be allocated beforehand unless it is guaranteed that that particular subroutine does not require one - this is an invariant of some C methods when compiled with a high enough level of optimization.

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5–7`	`t0–2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–11`	`a0–1`	Function arguments/return values	Caller
`x12–17`	`a2–7`	Function arguments	Caller
`x18–27`	`s2–11`	Saved registers	Callee
`x28–31`	`t3–6`	Temporaries	Caller
`f0–7`	`ft0–7`	FP temporaries	Caller
`f8–9`	`fs0–1`	FP saved registers	Callee
`f10–11`	`fa0–1`	FP arguments/return values	Caller
`f12–17`	`fa2–7`	FP arguments	Caller
`f18–27`	`fs2–11`	FP saved registers	Callee
`f28–31`	`ft8–11`	FP temporaries	Caller

Figure 7.1: RISC-V calling convention register usage [WA19a].

When returning from a subroutine call, the calling convention states that the subroutine simply stores the return value in the `a0-a1` registers and jumps into the address indicated by `ra`. In P-RISC, we want to observe a similar behavior. We would like a primitive that awaits for the subroutine to

finish using Join instructions and collects their results. However, two things could happen. If the caller reaches the merging point *after* the target method has finished, it can simply collect the aforementioned results. If the caller reaches the merging point *before* the target method has finished, it will not be able to keep executing. We want to avoid active waiting. Thus, we propose assigning a software frame in which the caller's hardware state is saved and then the caller stream is terminated. Then, when the target method ends, that stream will load the context stored by the caller and "mutate" into it. This way, we avoid any kind of active waiting.

The synchronization of parent and child streams is always done using the preallocated software frame. If the target method reaches the merging point first, it stores the return value in the addresses corresponding to `a0-a1` of the software frame and then terminates execution. The parent stream will infer the return values from there. If the caller method reaches the merging point first, it stores its architectural state into the software frame and then terminates execution. Then, the stream executing the target method will restore the context stored in the software frame and keep executing as if it was the parent.

The join operation itself is based on an atomic memory operation. The `JOIN` instruction reads from and writes to a memory address atomically. The result read indicates whether or not that stream is the first one to reach the merging point and must be terminated. The RISC-V ISAs declare that register `r0` does not actually store any value but rather ignores writes and always reads zero. Therefore, there is no need to explicitly store that value in the software frame and we can use that entry to piggyback the synchronization address that is used by the `JOIN` instructions.

We have identified five constructs that must be passed into the target function of a fork call. The following enumeration summarizes them.

1. `pc` - as in RISC-V ABI
2. `ra` - as in RISC-V ABI
3. arguments - as in RISC-V ABI
4. software stack - must be allocated beforehand
5. software frame - must be allocated beforehand

Given the constraints identified here, we have decided to implement fork operations as the two instructions `FORK` and `FORKR`. These instructions are, in a way, analogous to the standard `JAL` and `JALR` instructions. They are J-type instructions that fork the current stream of instructions into two. One of them keeps executing "pc+4" and the other one jumps into the target address. They both get a copy of the same register set values. The join operation has been implemented as an I-type instruction that read-then-modifies a memory position with the side effect of terminating the instruction stream when the read value is unsuccessful.

Listing 7.1 shows the C declarations of the low-level `fork_sync` and `wait_sync` methods. These methods are the lowest level C abstraction of the fork and join operators for the parent-child calling convention. They are used to call a target function in parallel and then collect the return value. The `fork_sync` primitive takes up to five user arguments that will be bypassed into the target function. Then, it takes pointers to the target function, the software frame, and the software stack. If the target function expects fewer than five arguments, the rest of them may be ignored. The stack pointer argument may be ignored if it is guaranteed that the child function does not use or try to access the stack. The `wait_sync` method takes the software frame pointer as the only argument and it is assumed that the first entry of that construct will be used as the synchronization point.

Listing 7.2 shows the assembly implementation of the `fork_sync` primitive. This primitive prepares the context and forks the stream. One of them acts as the parent and returns to the caller. The other jumps into the specified C ABI target function. When entering this primitive, we expect that it has been called using the standard C ABI. Therefore, the arguments of the original call are located in `a0-a7`. We selected this specific ordering of the arguments for the primitive so that the target

function arguments may be directly bypassed. That is, they are already in the correct place when the newly generated stream jumps into the target function. Therefore we only need to take care of the P-RISC-specific arguments.

We know that the `FORK` instruction generates a new stream with a copy of the current register set values. Because this primitive has been called using the ABI and the stream jumps into an ABI subroutine, most of the registers already hold the values that the target function expects. There are three exceptions. First, we must modify the return address so that the target function will return into the `child_return` tag instead of directly into the calling function. Second, we must move into the `sp` the value provided by the `a7` argument. Third, we must save the pointer to the software frame. This is not needed by the target function itself, but it will be needed in order to execute the merge when it returns to `child_return`. We piggyback this value into `s11`. Because the target function respects the ABI, we know that this value is preserved throughout the call. These actions correspond to lines 8-12. Registers `ra`, `sp` and `s11` are supposed to be saved by the callee. Because we modify them here, we also must save them and restore them before return. We do that in lines 5-7 and 15-17 using temporary registers.

Listing 7.3 shows the assembly implementation of the `wait_sync` primitive. This primitive is called by the parent stream when it is ready to collect the results from the target function executed by the child stream. The pointer to the software frame is passed as an argument. Before executing the actual join, the stream must save the current architectural state in the frame so that it might get restored by the child stream. Those are lines 5-20. Then, the actual join is executed. If the parent stream survives the join operation, it means that the child stream was terminated and it had stored the return values in the software frame. Now it is only needed to collect those values and return to the calling routine. Notice that not all registers are being saved in the software frame. As the `wait_sync` primitive is expected to be called as a C ABI function; we only must save the registers that are expected to be saved by the callee. The rest will be restored by the original caller after the return from the `wait_sync` method. A software implementation with better compiler support might be able to further refine this and only save the registers that will still be relevant after the call.

Back in listing 7.2, the `child_return` tag indicates the code that is executed by the child stream on return of the target function. Before executing the actual join, the stream must save the return value so that it might be later inferred by the parent stream. Those are lines 20-21. Then the stream executes the actual join. If the child stream survives the join operation, it means that the parent has been terminated. In that case, the child stream must restore the saved architectural state from the parent and return to the calling function. It will appear as if this stream is the parent returning from the call to the `wait_sync` primitive. These actions correspond to lines 25-42. Recall that in the `fork_sync` primitive we saved the software frame pointer in `s11` and that this value has been preserved by the C ABI target function. Therefore, we can use it to address the software frame.

Finally, we must discuss the actual encoding of the new `FORK`, `FORKR`, and `JOIN` instructions. Table 7.1 shows the base RISC-V opcode map. This mapping includes four "custom" opcodes. Those opcodes can be used to define custom instructions and extensions. We use one of them to encode the P-RISC extension, as shown in table 7.2. As aforementioned, the `FORK` and `FORKR` instructions are similar in encoding to the `JAL` and `JALR` instructions. These instructions use a J-type encoding. This encoding is very expensive though, as it only encodes one instruction per opcode. This encoding includes a destination register, which is not strictly necessary for `FORK` and `FORKR`. The join instruction uses an I-type encoding, like the load memory operations. However, we do not need to explicitly state a destination register. Therefore, we use a single opcode for all three instructions and use the destination register field to store the minor opcode from each individual instruction. This way, we can encode all of the P-RISC instructions using a single opcode.

```
void fork_sync(int a0, int a1, int a2, int a3, int a4,
                void *child_func, char *child_frame, char *child_stack);

int wait_sync(char *child_frame);
```

Listing (7.1) C declaration of the fork_sync and wait_sync primitives

```
.align 2
.text
.globl fork_sync
fork_sync:
    mv      t0,ra
    mv      t1,sp
    mv      t2,s11
1:
    auipc   ra,%pcrel_hi(child_return)
    addi    ra,ra,%pcrel_lo(1b)
    mv      sp,a7
    mv      s11,a6
    sw      zero,0(a6)
    .word   0x0007808b  # forkr a5
    mv      ra,t0
    mv      sp,t1
    mv      s11,t2
    ret
child_return:
    sw      a0,4*10(s11)
    sw      a1,4*11(s11)

    .word   0x000D810b  # join 0(s11)

    lw      x1,4*1(s11)
    lw      x2,4*2(s11)

    lw      x8,4*8(s11)
    lw      x9,4*9(s11)

    lw      x18,4*18(s11)
    lw      x19,4*19(s11)
    lw      x20,4*20(s11)
    lw      x21,4*21(s11)
    lw      x22,4*22(s11)
    lw      x23,4*23(s11)
    lw      x24,4*24(s11)
    lw      x25,4*25(s11)
    lw      x26,4*26(s11)
    lw      x27,4*27(s11)

    ret
```

Listing (7.2) RV32I Implementation of the fork_sync primitive.

```
.align 2
.text
.globl wait_sync
wait_sync:
    sw      x1,4*1(a0)
    sw      x2,4*2(a0)

    sw      x8,4*8(a0)
    sw      x9,4*9(a0)

    sw      x18,4*18(a0)
    sw      x19,4*19(a0)
    sw      x20,4*20(a0)
    sw      x21,4*21(a0)
    sw      x22,4*22(a0)
    sw      x23,4*23(a0)
    sw      x24,4*24(a0)
    sw      x25,4*25(a0)
    sw      x26,4*26(a0)
    sw      x27,4*27(a0)

    .word   0x0005010b  #join 0(a0)

    lw      a1,4*11(a0)
    lw      a0,4*10(a0)

    ret
```

Listing (7.3) RV32I Implementation of the wait_sync primitive.

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq$ 80b

Table 7.1: RISC-V base opcode map, inst[1:0]=11 [WA19b].

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	**PRISC**	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq$ 80b

Table 7.2: RISC-V base opcode map *with P-RISC instructions*, inst[1:0]=11 [WA19b].

7.2 C API and libraries

In the previous section, we discussed the low-level `fork_sync` and `wait_sync` C primitives. Although those two primitives are enough to start coding P-RISC algorithms in C, they are very bare-bones and do not offer the most expressive interface. Here we showcase a higher-level API implemented on top of the base primitives that makes coding in this parallel environment more intuitive. We also introduce a very simple API that gives access to the MMIO performance reports explained in the earlier sections from C programs. Finally, we added a simple fixed-point library implementing real and complex number approximations and operations - as our core does not yet implement floating-point extensions. The three libraries introduced here are used in the following section to implement the different benchmarks.

Listing 7.4 shows the declaration of the fork and wait for API methods. The first set of methods introduced here is the `fork`N family. These methods are simple inline wrappers of the `fork_sync` primitive that restrict the number of arguments in order to match those of the target child function[2]. As they are inline implementations, they only serve as a nicer facade and actually instantiate a call to `fork_sync` without extra overhead. Some of the target functions might not require a stack if they are optimized to only use registers to save the information. The `fork`N`NS` methods are similar wrappers of the same primitive but are called with *No Stack*. Finally, we add the `wait` method. This method wraps the `wait_sync` primitive and can be paired with any of the previous methods.

```
1  //////////// Full feature fork ////////////
2
3  inline void fork0(
4          void *child_func, char *child_frame, char *child_stack);
5
6  inline void fork1(int a0,
7          void *child_func, char *child_frame, char *child_stack);
```

[2]The methods have been formalized with 32-bit `int` arguments. However, functions with other types of arguments can actually be used simply by overriding the integer arguments declared here using the ABI standards. Neither the wrapper methods nor the primitives actually access the target arguments and they are forwarded directly to the target function.

```
8
9  inline void fork2(int a0, int a1,
10         void *child_func, char *child_frame, char *child_stack);
11
12 inline void fork3(int a0, int a1, int a2,
13         void *child_func, char *child_frame, char *child_stack);
14
15 inline void fork4(int a0, int a1, int a2, int a3,
16         void *child_func, char *child_frame, char *child_stack);
17
18 inline void fork5(int a0, int a1, int a2, int a3, int a4,
19         void *child_func, char *child_frame, char *child_stack);
20
21 //////////// No-stack fork ////////////
22
23 inline void fork0NS(
24         void *child_func, char *child_frame);
25
26 inline void fork1NS(int a0,
27         void *child_func, char *child_frame);
28
29 inline void fork2NS(int a0, int a1,
30         void *child_func, char *child_frame);
31
32 inline void fork3NS(int a0, int a1, int a2,
33         void *child_func, char *child_frame);
34
35 inline void fork4NS(int a0, int a1, int a2, int a3,
36         void *child_func, char *child_frame);
37
38 inline void fork5NS(int a0, int a1, int a2, int a3, int a4,
39         void *child_func, char *child_frame);
40
41 //////////// Wait ////////////
42
43 inline int wait(char *child_frame);
```

Listing 7.4: C parent-child P-RISC API

Two very common patterns in parallel paradigms are the geometric block and reduce calls. For instance, in commercial GPU platforms like CUDA, it is very common to find primitives introduced specifically for these types of patterns [Lin+08]. In those cases, the output is an array - or similar construct - that can be divided into smaller blocks. Parallelism is obtained by computing each block in parallel. We believe that this construct also makes sense in our architecture. Therefore we introduce a few more methods to the API as displayed in listing 7.5. We also add reduce methods.

The block methods are introduced in 1, 2, and 3-dimensional variants. The block methods are provided with $\Phi 0$, $\Phi 1$, and $d\Phi$ arguments for each dimension and a shader function. Multiple instances of the shader function are called in parallel through the parallel grid and they must take as arguments the coordinates of the point of the grid they are being called on. The result grid is walked in intervals $[\Phi 0,\Phi 1)$ with a $d\Phi$ stride. Notice that the block methods' declaration does not accept software frames nor stacks to use in the target function as arguments. The software contexts are pre-allocated by the API internally. The amount of them and the size of the stacks allocated can be tuned using define clauses or may be ignored and they will be allocated using the API default values. The user cannot assume anything about the order in which the points in the grid are processed. This family of methods can be used, for instance, for matrix multiply implementation.

The reduce methods are also introduced in 1, 2, and 3-dimensional variants. They are very similar in syntax and semantics to the block family of methods. The shader functions here also are expected to have a return value. We also add a reduce function with arguments of the same type. As the shader function calls are completed, the results are reduced using the reduce function. As the order

in which the points are processed is not deterministic, the reduce operation must be commutative and associative. This family of methods can be used, for instance, to compute the number of points belonging to Mandelbrot's set in a given region of the complex space.

```
//////////// Block ////////////

void block1D(int x0, int x1, int dx,
        void (*shader_func)(int));

void block2D(int x0, int x1, int dx, int y0, int y1, int dy,
        void (*shader_func)(int,int));

void block3D(int x0, int x1, int dx, int y0, int y1, int dy, int z0, int z1, int dz,
        void (*shader_func)(int,int,int));

//////////// Reduce ////////////

int reduce1D(int x0, int x1, int dx,
        int (*shader_func)(int), int (*reduce_func)(int,int));

int reduce2D(int x0, int x1, int dx, int y0, int y1, int dy,
        int (*shader_func)(int,int), int (*reduce_func)(int,int));

int reduce3D(int x0, int x1, int dx, int y0, int y1, int dy, int z0, int z1, int dz,
        int (*shader_func)(int,int,int), int (*reduce_func)(int,int));
```

Listing 7.5: C block and reduce P-RISC API

Finally, we introduce the MMIO calls in listing 7.6. These are very simple methods that map the MMIO devices defined in the particular system implementation. A call to each of these methods triggers a write into the corresponding MMIO address. They can be used to print the value of the different performance counters in order to measure the performance and resource impact of different algorithms and code regions in this hardware platform. These implementations serve not only as benchmarks for performance evaluation but also to illustrate how the API is employed by a user.

```
int print_MSG(char c);

int print_HEX(int c);

int print_MSR(char c);
```

Listing 7.6: C MMIO API

7.3 Software benchmarks

Software benchmarks work as a standard metric to evaluate the performance of a system. Although there are many standardized performance benchmark implementations, they obviously do not use the parallel C constructs defined specifically for P-RISC. Therefore, we needed to make our own implementations in order to actually benchmark the benefits of this architecture. We chose a range of common algorithms and developed both reference C sequence implementations and P-RISC parallel implementations. All of the benchmarks have been implemented using either base C syntax or constructs introduced in our base API. Here we explain at a high level how they were implemented. In the next section, we evaluate the performance results.

The sorting algorithms considered here not only benefit from parallel P-RISC implementations but also exemplify the parallel implementation of recursive algorithms. This is a scheme that we believe can be transferred into other algorithms that are based on a similar recursive strategy. We start by implementing mergesort as it very clearly exemplifies how to map recursive calls into parallel streams. We then move into quicksort, which benefits from a more refined implementation of the same basic

concepts and yields a faster solution. Finally, we discuss heapsort. This is not a very efficient nor practical implementation but motivates an interesting parallelization strategy.

7.3.1 Mandelbrot

Computing Mandelbrot's set is a very straightforward parallelism test that is widely used as an initial benchmark. Given a point c in the complex plane, sequence 7.1 is defined. The point c is said to belong to Mandelbrot's set iff it makes 7.1 bounded. Experimentally, this is tested by iterating over the sequence. If the absolute value of the terms is observed to grow over the *infinite* threshold, c is not considered to belong to Mandelbrot's set, and the computation terminates. After *maxiter* iterations, if the terms have been kept below the *infinite* threshold, c will be considered from the set. This is of course an approximation and bears no mathematical rigour. However, it makes for an interesting computational problem.

$$\begin{cases} Z_0 = 0 \in \mathbb{C} & \text{(initial term)} \\ Z_{n+1} = Z_n^2 + c & \text{(recursive term)} \end{cases} \tag{7.1}$$

The problem we are trying to solve here is to compute the number of points from a given grid that belongs to Mandelbrot's set. This roughly computes the probability of a randomly chosen point on a bounded set from the complex plane from belonging to Mandelbrot's set. An obvious choice is to use the reduction methods defined in the API. The two-dimensional reduction method can be parameterized to walk any rectangular grid of points with regular stride. The shader function simply takes the coordinates, defines a chunk of the grid, and iterates over 7.1 for the points in that chunk. It returns the number of points in that chunk that were approximated to belong to the set. The reduction function is a simple addition.

A usual rule of thumb to parameterize parallel algorithms is to roughly divide the problem into as many tasks as computing devices available. This is assumed to yield a balanced load and efficient use of resources. For our single-core system, this would imply dividing the grid into as many chunks as harts in the core. However, it is challenging to find a division that yields balanced loads. Although all child tasks defined by this algorithm call the same shader function, the number of iterations of 7.1 computed can differ vastly in each point c. This creates very unbalanced tasks.

The strategy that we use to overcome this issue is simply dividing the grid into many chunks. At any given time, only a few of them will be processed by the corresponding tasks hosted in the harts. The rest will wait in the MTQ. As the "faster chunks" get processed, new ones will be pulled in. On average, this should distribute the total load evenly. We can afford to do this in our architecture thanks to the lightweight fork-join model. Other architectures with larger overhead cannot afford to execute such fine-grained parallelism.

Notice that in this implementation we do not need to worry about assigning neither software frames nor stacks to the child tasks. This is due to two reasons. Compiling the target function with a high-enough level of optimization - typically O2 or O3 in GCC - generates a target function that does not use a stack. In any case, the frames and stacks used when calling this method have been pre-allocated by the API library rather than explicitly by the user.

7.3.2 Matrix multiply

Matrix multiplication is also a widely used simple benchmark. Specifically, we implement matrix multiply and accumulate with a transposed B matrix for better cache locality. We already discussed this problem as an example when we defined the P-RISC execution model. For clarity, we include here the base code again in listing 7.7. Recall that the core of the computation is composed of the three nested loops I, J, and K. The three loops might be reordered in any way without modifying the results but with performance implications.

```
void mmt() {
```

```
  for (int i = 0; i < MSZ; ++i) // I
    for (int j = 0; j < MSZ; ++j) // J
      for (int k = 0; k < MSZ; ++k) // K
        C[i][j]+=A[i][k]*B[j][k];
}
```

Listing 7.7: Simple sequential matrix multiplication code. The matrices are assumed to be squared and from size MSZ. Matrix B is assumed to be transposed.

Variables j and k iterate through a row of the matrices, which are stored in sequence throughout a few cache lines. Iterating over these variables, therefore, generates a memory access pattern that reuses cache lines for a few consecutive accesses. Variable i iterates through a column of the matrices. This generates a memory access pattern that changes the memory line being accessed every time - unless the matrix is tiny and several rows fit in the same cache line. Therefore, we want to keep i in the outer loop.

This problem can be seen as a three-dimensional problem. The work could be divided by splitting the three loops into chunks using a 3-dimensional block. We could also choose to divide the work using 2-dimensional or 1-dimensional blocks over only a few of the loops. Dividing the work over the K loop creates a data race as several tasks might try to read-then-write to the same position in C. We, therefore, treat this as either a 1-dimensional or 2-dimensional block problem on loops I and J. We implement it as a 2-dimensional block problem. Each shader instance computes a square chunk of C. This approach not only discovers plenty of parallelism but also allows the simultaneous execution of several tasks working with the same subset of i and j indices, allowing cache line reuse among tasks. The block methods actually spawn the tasks in order. Thus, it is reasonable to assume that some of them might be working simultaneously with the same set of is.

7.3.3 SAXPY

SAXPY is a widely used sparse kernel based on the vector operation $Z = \alpha X + Y$ where X,Y, and Z are vectors and α is a scalar. This is an embarrassingly parallel problem that can be parallelized very efficiently with P-RISC. We simply divide Z into blocks and use a block method from the API in order to manage the creation and synchronization of tasks. The parallelization strategy is very straightforward and naturally yields a well-distributed load.

7.3.4 mergesort

Mergesort is based on a very straightforward dive-and-conquer strategy. The input array is split into two halves and then each half is sorted recursively. It is very intuitive to consider the idea of executing each of the two recursive calls in parallel. In fact, we already used this very example when explaining the P-RISC execution model. It is very easy to justify that this idea makes sense due to two key facts. First, the sections of the array being accessed and sorted by each recursive call do not intersect. Thus, they can access and modify them without the need for extra synchronization overhead. Second, parallelism is also discovered recursively in parallel. Only the very first call discovers the parallelism and splits into two in series. From here downward, the tasks are forked from already parallel tasks. This helps in hiding the overhead required by task forking and result collection.

Let us now discuss the data structures required by mergesort and how they are accessed by the recursion tree. Mergesort is not an in-place sorting algorithm. When two array sections are stored in a given memory region they cannot be merged directly in the same memory region. A scratchpad region must be allocated in order to proceed with the merging process. This can simply be overcome by allocating beforehand a memory array parallel to the one hosting the original array to be used as a scratch. Each recursive call used the scratch section corresponding to the array section being sorted. Each recursive call also requires a software frame and stack in order to execute in parallel and merge the results.

The mergesort algorithm actually has a very convenient recursive call structure. The recursion tree is actually a complete binary tree. This type of tree can be indexed as a heap. The root node is issued an identifier of 0. Then, each child of node id will be issued the identifiers $2 \times id + 1$ and $2 \times id + 2$. In a complete binary tree, this creates a pattern of indexes that maps perfectly the tree into an array with the same number of entries. This is a well-known property that is typically used to efficiently store complete binary trees. We use it to assign software frames and stacks.

Each non-leaf recursive call must fork into two. This means that every node of the recursion tree must have access to one frame and stack to be assigned to their child task. Our implementation pre-allocates an array of frames and stacks. Then, the mergesort calls are tagged by forwarding the heap identifiers of each node. Each recursive call has thus a unique identifier. This identifier is used to index the frame and stack arrays and recursively assign them to the child tasks. The recursive calls only fork into parallel calls when the identifier fits within the pre-allocated arrays. Afterward, they are called in series. The length of the pre-allocated arrays implicitly dictates the number of forks that might be executed and thus the granularity of the parallelism obtained.

7.3.5 quicksort

Quicksort is also based on a similar divide-and-conquer strategy. However, here the input array is not divided into two equal halves. The call starts by selecting a pivot and then splitting the array into two sections containing respectively the elements that are smaller or equal than the pivot and the ones that are larger. Then, these two sections might be sorted in parallel. After the recursive sort calls, the whole array becomes sorted. The core idea of this sorting strategy is very similar to that of the mergesort. The two main implementation differences are that the split of the array is executed before the recursive calls rather than afterward - unlike the merge - and that the implementation is in-place. We do not need to assign scratch memory.

The largest difference in terms of performance and parallelization strategy is however that the recursion tree defined by quicksort is not complete. This algorithm also defines a binary recursion tree. It is only natural therefore to implement this in parallel using the same heap strategy as we did in Mergesort. However, it must be kept in mind that quicksort does not necessarily split the array into two equal halves. Therefore, the binary recursion tree does not need to be complete. This creates two problems. First, the parallel recursive calls become unbalanced. We can mitigate this by working with finer granularity, as we did with Mandelbrot. Second, the mapping from recursive tasks into the frame and stack arrays are not necessarily exhaustive. Depending on the input data, the recursion tree will not exhaust the parallelism enabled by the number of frames and stacks preallocated.

Depending on the input data, the actual split operation can bear a significant part of the total execution time of the algorithm. We believe that a heuristic parallelization strategy can be used to further parallelize this part of the algorithm. It must be noted that this is probably not needed in the deeper recursive calls of the main recursion tree, as those are already being executed in parallel. However, this might be exploited in order to obtain more parallelism at higher levels of the tree. The parallelization strategy is as follows.

Consider we call a split function that performs the split only in the even positions of the array. Simultaneously, we call a split function that acts only in the odd positions of the array with respect to the same pivot. These calls return two indices $m1$ and $m2$ respectively; indicating the split point of each sub-array. It is guaranteed that all elements, whether even or odd, left of $min(m1, m2)$ are smaller than the pivot. It is also guaranteed that all elements right of $max(m1, m2)$ are larger than the pivot. Only the elements in the interval $[min(m1, m2), max(m1, m2)]$ are assorted. We now only need to execute a sequential split in this section of the array. When executing quicksort on randomly generated arrays, the intersection becomes small in relation to the whole array.

We can further refine this strategy in two different ways. First, we might divide the original array into finer partitions than even-odd, exposing thus more parallelism. This can also be done recursively, discovering even more parallelism in parallel. Second, one may further refine the interval $[min(m1, m2), max(m1, m2)]$ by executing again parallel split using a different partition strategy on

that intersection. Experimentation has shown that this is usually not needed, as the intersection obtained in random arrays is already small enough. We do implement a binary recursive split strategy tough.

We have now established that each node of the main recursion tree might execute the split operation also in parallel as a binary recursion tree. We need therefore to find a way to assign frames and stacks to those tasks. The solution to this problem is however trivial. The recursive split is never executed in parallel to the child calls of the main recursion tree. Thus, we can use the same frames and stacks that were already pre-allocated for them. We tag the recursive split calls with the same heap identifier as the main recursion tree and reuse the same pre-allocated memory.

7.3.6 heapsort

Heaport is a very interesting storing algorithm. It is based on a two-step process. First, the input array is transformed into a binary heap. Here we use the term *heap* to refer to the specific case where the binary tree holds the property of each node being always larger than both their children. Then, the root of the tree is removed, as we know that this is the largest element of the array. The two remaining sub-trees also define heaps that must be merged into a single one by a process commonly referred to as *heapify*. This process is iterated until the heap has been exhausted and a sorted array has been obtained.

The first step of the problem is relatively easy to parallelize. One can simply split the array into several sections and transform them into heaps in parallel. Then, merge the partial heaps using heapify. The second part is more challenging. The process of remove-then-heapify is by nature sequential, as is the heapify process itself. We cannot remove the following root until the array has been heapified again. Consider now removing two elements of the heap each time, instead of a single one. In some cases, we might be able to parallelize the heapify process.

Consider the problem where we want to remove the two largest elements of the heap. We know that the maximum is located a the root of the heap. If we remove that element, the next maximum could be either child. Let us suppose it is the right one - the left one would create an equivalent situation. If we remove it, we obtain the tree shown in Sub-figure 7.3a. Then, F1, F2, and F3 define the *frontier* of the current tree. We know that the frontier of the tree contains the next candidate maximum. There are the following cases:

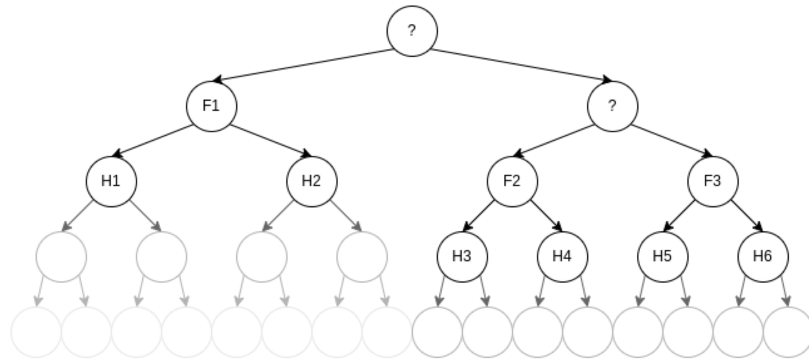
1. F1 is the current frontier maximum. If we float that element to the root, we reach sub-figure 7.3b. Now we might heapify both children in parallel.

2. F2 is the current frontier maximum. Floating that element to the root yields to sub-figure 7.3c. We reached an equivalent situation in the right sub-tree.

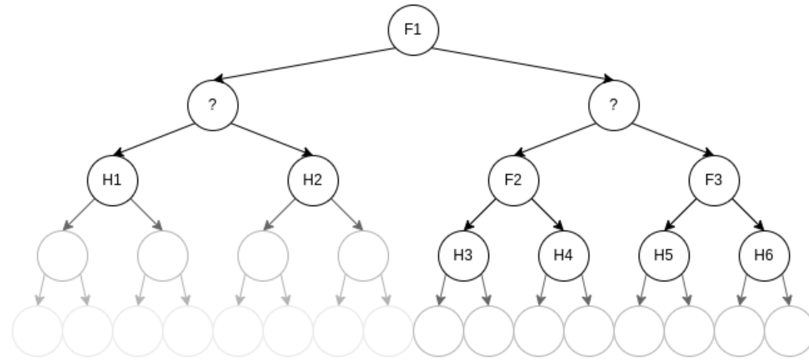
3. F3 is the current frontier maximum. Floating that element to the root yields to sub-figure 7.3d. We reached the same situation in the right sub-tree.

This call does not always discover parallelism immediately. However, every time it gets called recursively, it has a chance of doing so.

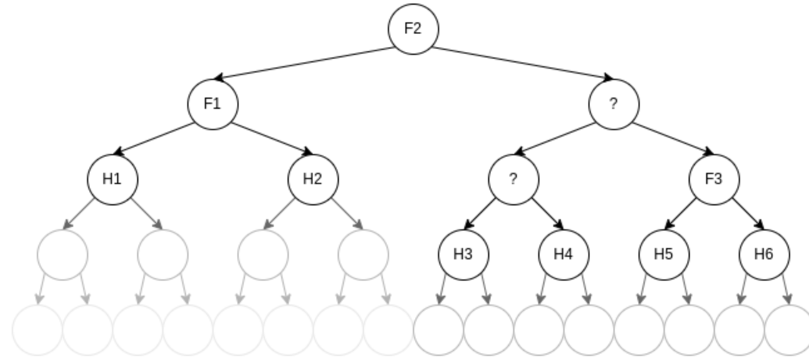
The current implementation of parallel heapsort uses the strategy described here. This limits the parallelism to two concurrent streams of instructions during the remove-then-heapify process. More parallelism could be extracted from removing several elements in each iteration. However, unfolding such an operation by hand becomes tedious. We believe that there might be a way of automating this process to remove an arbitrary amount of elements every iteration. Then select the maximum from the obtained frontier and dynamically consider which resulting heapify operations might be executed in parallel. The automatic code unfolding discovered here promises to be an interesting topic of research. It is far from the scope of this work however and it will not be considered here.



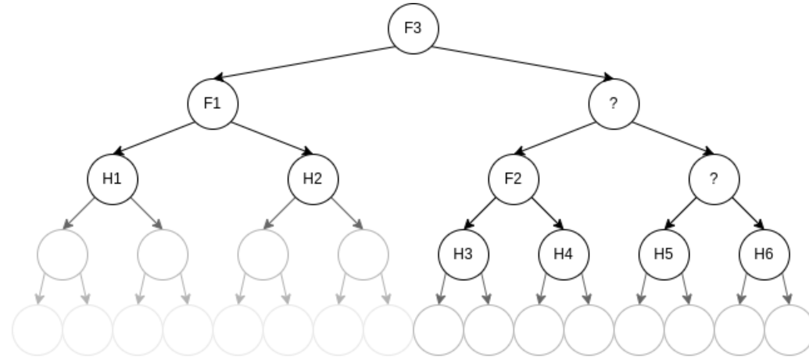
(a) Root and right child are maximums.



(b) Root and right child are maximums. F1 is next.



(c) Root and right child are maximums. F2 is next.



(d) Root and right child are maximums. F3 is next.

Figure 7.3: Heapsort parallel call case evaluation.

Chapter 8

Evaluation

This chapter evaluates the results of the experiments conducted in this work. We divided this chapter into three sections. The first one evaluates the quality of the physical design and synthesis of the core. We focus firstly on the arbiter module under several configurations and then study the synthesis of the whole core under three configurations of increasing frontend width. The second one discusses the performance results obtained in the software benchmarks. We first introduce the hardware configurations tested and a graphical representation that we believe transmits a lot of information about the behavior of the algorithms executed in the SMT core. We then discuss the results of the different benchmarks using that representation. The third one gives a summary of the results.

8.1 Timing and area in ASIC

8.1.1 Arbiter

The instruction arbiter has undoubtedly been the most problematic element of the core in terms of area and timing. As already outlined in the corresponding section, a total of three major arbiter versions have been considered throughout this work. The initial one did not offer a circuit of acceptable quality. The other two work in most configurations, although the latter one pulls ahead in larger ones. This component bears significant relevance in our design as not only it is a module that is unique to this architecture but it also has become the timing bottleneck on many occasions. We believe that in order to prove the scalability of our core in terms of frontend and backend widths it is important to prove the scalability of this module. Here we focus on synthesizing the arbiter module using different configurations in order to understand such property.

Recall that we started the development of the arbiter with a naive implementation that did not use scheduling priorities and only tries to maximize the number of scheduled instructions. Using the ASN abstraction defined in the corresponding sections, the naive arbiter implementation only sorts instructions according to validity. This implies that the equivalent ASN only uses single-bit comparisons in each node.

The initial implementation proved to scale poorly on the FPGA and later on in the ASIC synthesis. Furthermore, we wanted to enhance the scheduling process by associating each instruction with a priority level. As discussed earlier, we use a 3-bit speculation counter as our priority indicator. Each node of the equivalent ASN requires a 4-bit comparison. It is clear that the naive design cannot handle such a feature.

The second iteration of the arbiter module was to move to a design that explicitly used a hardware sorting network to sort the candidate instructions according to both validity and priority. This design uses the ASN scheme defined in the earlier sections. This scheme was initially designed using an insertion sort sorting network implemented in high-level BSV syntax. This version proved to be much

better than the naive one. However, it was not fast enough, and wider configurations require lowering the precision of the priority to 2 bits to be competitive.

The third and final iteration improved upon the second one by using a bitonic sorting network instead of an insertion sorting network. We used a bitonic network included in the internal BSC libraries. This network exposes a BSV interface but offers directly an efficient Verilog implementation. We decided to substitute the custom ASN from the second iteration with an instance of this function. This improved the quality of the design significantly.

We decided to synthesize the two latter iterations of the arbiter with widths ranging from 2 to 16. The standard configuration of our core has an 8-wide frontend and a 3-wide backend. We call this an 8×3 configuration. The arbiters tested here use $n \times n$ configurations. Meaning that both the inputs and outputs are matched in size. We expect realistic instances of this module to be $n \times m$ with $m < n$. In that case, part of the ASN output is ignored by the following stage. Thus, we expect the synthesis tools to trim the unused logic and reduce the total area of the ASN. In fact, one can observe a significant reduction in area when comparing the 4×4, 8×8, and 12×12 arbiters synthesized here and the 4×3, 8×3, and 12×3 arbiters included in the core synthesis of the next section. However, in order to normalize the tests we restrict the synthesis to the more demanding $n \times n$ configurations.

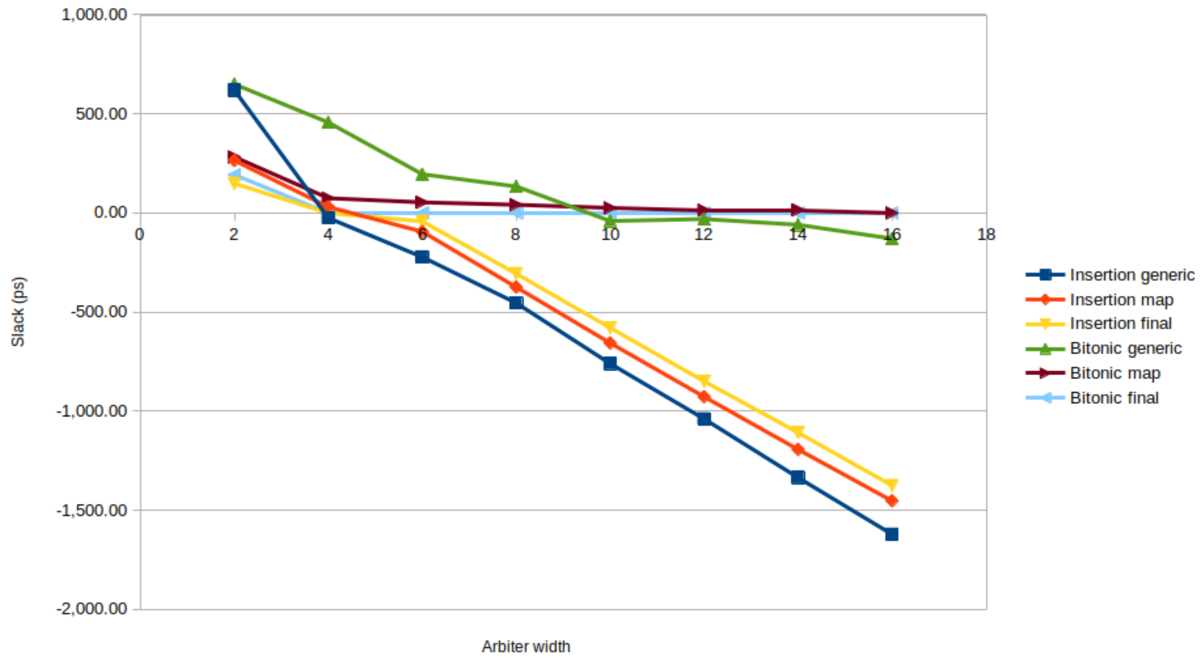
Figure 8.1 summarizes the results obtained from the arbiter synthesis with a 900ps clock cycle target for both insertion and bitonic variants. The naive implementation was not considered here as we only managed to synthesize it with very small configurations. As a frame of reference, this is the clock target used to synthesize the whole core in the following section; it takes an area of about $350000\mu m^2$ and just about meets timing constraints[1]. We tested configurations ranging up to 16×16 as we would not believe that a larger configuration of our current core would be reasonable.

Sub-figure 8.1a summarizes the slack obtained by both implementations. The insertion arbiter only meets timing constraints in the generic and map stages for up to 4×4 configurations. After optimization in the final synthesis, it roughly meets timing constraints also for 6×6 configurations. However, it does not scale further, reaching a slack that almost doubles the target clock cycle. The bitonic variant scales properly in up to 16×16 configurations. As we can observe, in the smaller configurations, the generic stage actually has more slack than the map and final stage. As timing constraints were already met, the tool tried to minimize area at a timing cost. This suggests that configurations of the arbiter up to 8×8 may actually scale to a faster clock cycle.

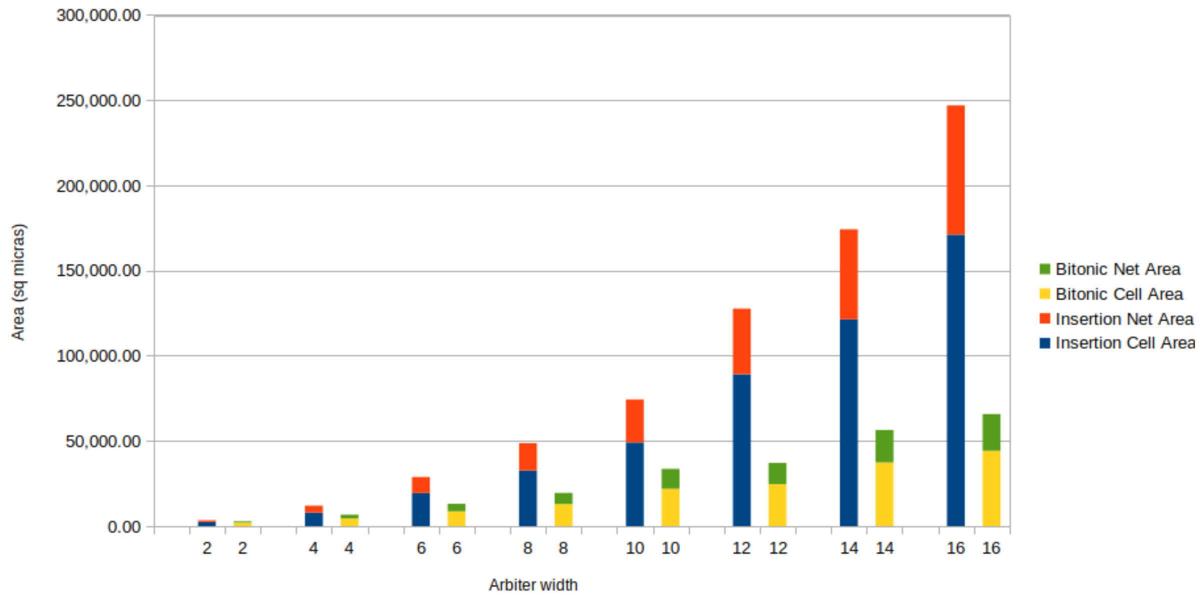
Figure 8.1b summarizes the area obtained by both implementations divided into cell and net area. Under the smaller 2×2 and 4×4 configurations, both variants take a relatively small amount of area. However, in 6×6 and 8×8 configurations, the arbiter starts taking a significant amount of area in relation to the whole core. It is clear by these results that the insertion arbiter might be an acceptable solution for smaller instances, but it does not scale properly into larger ones as it occupies an amount of area comparable with the whole core. The bitonic arbiter also takes a significant amount of area in larger configurations but at a much more reasonable scale. The larger configurations show a 3.00x to 3.75x difference in area among both variants.

We showed that the naive BSV implementation of the arbiter does not synthesize into a circuit of acceptable quality unless it is used in very small configurations. The ASN-based solutions are much better in both quality of the synthesis and functionality - as they allow for priority scheduling. The insertion ASN variant scales reasonably well into 6×6 and 8×8 configurations, it is not suited however for larger configurations. The bitonic ASN variant scales reasonably well up to 16×16 variants and it is faster and smaller than the insertion variant across the range. Recall that the $n \times n$ configurations synthesized here will probably be trimmed into $n \times m$ with $m < n$ variants requiring fewer resources in realistic instances. All in all, it has become clear that the bionic arbiter is the better implementation and scales reasonably well into larger instances. For the remainder of this work, this is the variant that we use.

[1]This synthesis includes the core and a first level of caching with 16KB for instructions and 32KB for data. It does *not* include the multiplication and division units, however.



(a) Slack obtained by the insertion and bitonic arbiter with a 900ps clock cycle target.



(b) Area obtained by the insertion and bitonic arbiter with a 900ps clock cycle target.

Figure 8.1: Slack and area obtained after synthesizing the insertion and bitonic arbiters with Cadence Genus under a 900ps clock cycle target. Slack results are shown for generic, map, and final synthesis phases. Area results are shown divided into cell and net area for the final synthesis phase only.

8.1.2 Full core

We have synthesized three different configurations of our core with different frontend widths. We wanted to study how well would our design scale and what frontend width would be appropriate in order to saturate our execution resources. Therefore we decided to synthesize 3 SMT designs with 4, 8, and 12 harts. All of them share the same backend with two arithmetic lanes and a single memory lane as well as the first-level caches. We call these hardware configurations SMT.04x3, SMT.08x3, and SMT.12x3 respectively. We expect them to scale reasonably well, as the bitonic arbiter has already been proven to scale properly to the more demanding 4x4, 8x8, and 12x12 configurations and otherwise it is mostly a matter of replicating the frontend resources. In order to give a point of reference in the 22nm technology that we are using we also decided to synthesize the minimal SMT.02x2 configuration - with two harts and single arithmetic and memory lanes - and include data about the synthesis of the Sargantana and RiscyOO cores.

Sargantana is an optimized 7-stage RV64G core that was developed to serve as a solid foundation for future academic designs in [Sor+22]. This core is way more refined than the base 5-stage pipeline that we extended to SMT in this work. Sargantana includes a single-issue frontend and a configurable backend with out-of-order write-back. This core was proven to be competitive when compared to academic cores like the Rocket core or Ariane [Asa+16; ZB19]. It must be noted that Sargantana implements a full 64-bit RISC-V ISA with all the G extensions - IMAFD, Zicsr, Zifencei [WA19b]; which makes it, of course, more resource intensive than our 32-bit design.

RiscyOO is a more powerful superscalar core developed using the *Composable Modular Design* in [Zha+18]. This core implements a full out-of-order pipeline with a dual-issue frontend and it is targeted at a higher tear of performance. It has been proven to be competitive in several configurations with Rocket, ARM cores, and other academic designs. RiscyOO implements a 64-bit RISC-V ISA with the G and C extensions; which also makes it more resource-demanding than our design.

We summarize the results of our synthesis in table 8.1. This table includes the three designs that we are studying as well as the three reference designs. All of them include basic information about their configurations. We show the total area and the slack obtained under our typical corner with a 900ps target. We also include the portion of the area that is specific to the arbiter in the SMT designs. It must be kept in mind that both Sargantana and RiscyOO are 64-bit designs with a more extensive ISA and that they must not be considered as a direct point of comparison but rather as a rough reference of what can be expected of this synthesis process. It is expected that scaling our SMT design to a 64-bit ISA would roughly double the area and resources as most functional units and stateful elements scale linearly on the data width. It must also be kept in mind that the SMT synthesis does *not* include the multiplication and division functional units.

Our results show that the bitonic arbiters included in the cores use significantly less area than the $n \times n$ configurations of the previous section. It approximately takes 90% of the area in the smaller SMT.04x3 and 63% in the larger SMT.08x03 and SMT.12x03. This is expected as the synthesis tools can trim part of the unused sorting network and miscellaneous arbiter logic with the narrower backend. The difference in the larger configurations is much more noticeable as the width difference is more significant. This shows that the bitonic arbiter in realistic designs should actually scale better than shown in 8.1b in terms of area. This is not necessarily true in terms of timing, as trimming the network might not reduce the critical path.

The increase in the number of harts has a significant impact on area. We observe that the overall core area increased by about 30% and 33% when scaling from SMT.04x3 to SMT.08x3 and to SMT.12x3. This suggests a difference of about 22000 to 32000 μm^2 per extra hart at this scale. Recall that not all components necessarily scale linearly. Apart from replicating some frontend components, the arbiter and the commit logic do not scale linearly as they multiplex from the frontend width to the backend width and vice-versa. The split-phase memory system tags the requests by hart ID, which slightly increases in size with a wider frontend. This slightly increases the resource usage of the L1I and L1D. We have observed a minor difference between them.

We observe an absolute difference of about 90000 μm^2 when scaling from SMT.04x3 to SMT.08x3

Base core	ISA	L1I	L1D	Frontend	Backend Arithmetic	Backend Memory	Arbiter (μm^2)	Total (μm^2)	Slack (ps)
SMT.04x3	RV32IM	16KB	32KB	4 hart	2 lanes	1 lane	6003	299000	0
SMT.08x3	RV32IM	16KB	32KB	8 hart	2 lanes	1 lane	12260	390000	0
SMT.12x3	RV32IM	16KB	32KB	12 hart	2 lanes	1 lane	23500	520000	-34
SMT.02x2	RV32IM	16KB	32KB	2 hart	1 lane	1 lane	2706	246000	-2
Sargantana	RV64G	16KB	32KB	single issue	1 lane	1 lane	NA	460000	0
RiscyOO	RV64GC	32KB	32KB	dual issue	2 lanes	1 lane	NA	926000	-13

Table 8.1: Microarchitectural configuration of the target core and resulting area and slack under a typical corner in Genus Synthesis with a 900ps clock target. Sargantana and RiscyOO are designs borrowed from [Sor+22] and [Zha+18] respectively. The synthesis of Sargantana shown here actually belongs to a 1000ps clock cycle target, as this is the data that we had available at the time of writing. The authors of this work confirmed however that Sargantana could also be successfully synthesized with a 900ps clock cycle target and similar area results. The synthesis of our SMT cores do *not* include the multiplication and division functional units.

and 130000 μm^2 when scaling from SMT.08x3 to SMT.12x3. This indicates an average of 27600μm^2 of extra area per hart. In the synthesis reports we have been able to explicitly account for an average of 12600μm^2 of that due to the following increases in resources: (1) Increase in arbiter area - 2100μm^2, (2) Extra register file - 6100μm^2, (3) Extra scoreboard - 300μm^2 (4) Extra L0I and L1I port - 4000μm^2, and (5) Extra decoder - 100μm^2. We attribute the remaining 15100μm^2 to the increased complexity of the write-back logic at commit stage and the increased size of the hart IDs in both the backend and the first-level caches.

The SMT.04x3 and SMT.08x3 configurations meet timing, but the larger SMT.12x3 has a -34ps WNS under the typical corner. This indicates that the two smaller configurations can run at 1.1GHz under typical conditions. The larger one will need to be slightly slowed down to 1.07GHz. After inspecting the timing reports, we believe that the current critical path in this configuration is located at the bypass redirect and write-back queues going from the commit stage to the frontend. This makes sense as the write-back logic at commit stage must route each element to the correct hart. This logic becomes larger and slower when a wider frontend is configured.

8.2 Software benchmarks

We use a combination of four graphs that allow us to measure and understand the reasons behind the performance obtained in each software benchmark. In the next section, we explain how to read these figures using the results obtained in a merge sort execution as an example. Next, we present and analyze the results for each benchmark. The information displayed in these graphs has been extracted from the device using the MMIO-mapped event counters described in the testbench section. All of the tests have been written in C and use the P-RISC API as described in the previous chapter. They have been compiled using the following GCC flags with O3 optimizations: `riscv64-unknown-elf-gcc -O3 -march=rv32im -mabi=ilp32 -static -nostdlib -nostartfiles -fno-builtin -mcmodel=medany`

8.2.1 Hardware configuration

Each performance benchmark has been run on five different hardware configurations. The first three are the SMT.04x3, SMT.08x3, and SMT.12x3 already presented in the previous section with 2, 8, and

12 harts at the fronted and the same 3-wide backend. Those configurations are run on the VCU108 FPGA with a 4-way skewed associative 128KB second-level cache with 5 cycles of latency. To the three base configurations, we add the SMT.NPA and SMT.NL2 variants from SMT.08x3. SMT.NPA is identical to SMT.08x3 except in that the arbiter does not use priority scheduling, it only tries to maximize the number of issued instructions, regardless of speculation level. SMT.NL2 is identical to SMT.08x3 except in that we do not include the second-level cache. First-level misses are forwarded directly into main memory. We add the execution of the sequential equivalent of each benchmark as a reference marked as SEQ as run in our core. It should be noted though that the performance of our machine in sequential programs is fairly low due to the added arbiter stage and the general lack of optimizations targeted towards sequential execution.

Recall that we estimated that the main memory controller of the VCU108 board can handle a throughput of 1 request every 10 core clock cycles with a latency of 17 to 18 core clock cycles in our configuration. Between the second-level cache and the main memory controller, we artificially add 100 cycles of latency to mimic the execution in an ASIC environment. In the SMT.NL2 configuration this latency is added between the first level of caching and main memory. Therefore, the expected latency to main memory of this system is 117 to 118 cycles.

8.2.2 IPC, speedup, and instruction distribution graphs

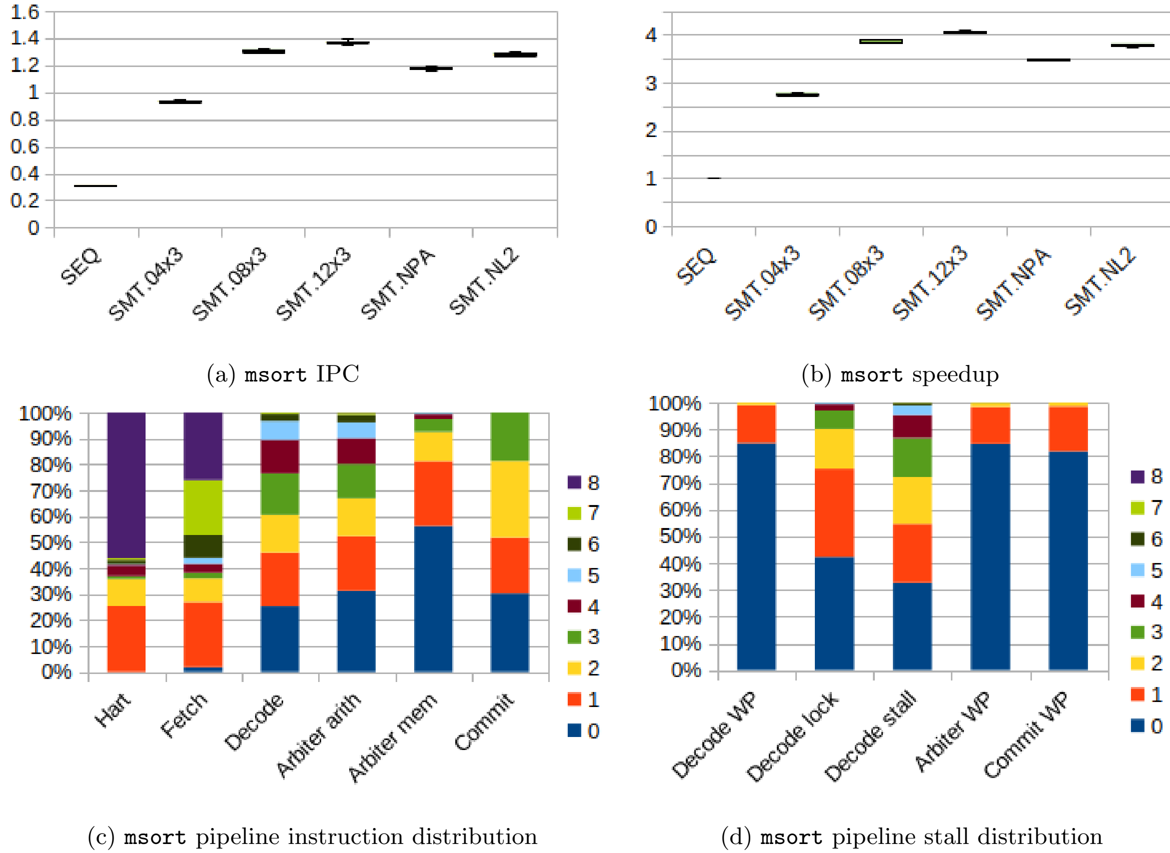
Figure 8.2 shows an example of the graphs that we use in the following sections. We have used as an example the results obtained when executing mergesort, but the same methodology and data representation methods apply to any other tests in this section. Each test has been run 10 times on randomly generated data. The generated data is carried over the different hardware configurations. The first two sub-figures present box and whiskers plots of the 10 executions in each hardware configuration. They are straightforward representations of the absolute IPC obtained in each hardware configuration as well as the speedup in relation to the sequential configuration. The latter two represent the distribution of instruction throughput through the pipeline and the distribution of events that could harm such throughput. Those show only the metrics of one representative execution of the algorithm in the SMT.08x3 hardware configuration. They show the pressure that each algorithm puts on the core and help us understand the reasons behind the performance obtained.

Sub-figure 8.2c shows the distribution of instruction throughput through the pipeline. In each cycle we measured how many valid instructions reached every stage of the pipeline; ranging from 0 to 8 for the frontend stages and the arbiter input and from 0 to 3 in the arbiter output and the backend stages. Using local counters we accumulated the frequency of each possible count. At the end of the algorithm execution, this information was reported. For instance, in sub-figure 8.2c we can see that in 30% of the cycles, no instructions are committed, in about 20% of the cycles only one is committed, in about 30% of the cycles two of them are committed and in about 20% of the cycles three are commit. The graph displays a distribution for every significant stage of the pipeline.

The *Hart* entry is slightly different than the rest. It shows to occupancy of the harts. Not all algorithms discover enough parallelism and create enough parallel instruction streams to use all the available harts. For instance, in sub-figure 8.2c we see that for mergesort about 55% of the time all harts are hosting a stream, about 20% of the time 1 to 7 harts are occupied, and about 25% of the time only one stream is being executed. After that, we show the instruction distribution reaching the Fetch, Decode, Arbiter, and Commit stages of the pipeline. The instruction distribution at arbiter stage is split among arithmetic and memory instructions. Notice that at commit stage we never observe a count higher than 3, as that is the width of the pipeline. The Execute and memory stages of the backend cannot cause any stalls or invalidate any instructions. Therefore, their distributions would be equivalent to that of commit stage. To simplify the graph, we display only the commit.

At each stage of the pipeline the instruction throughput drops due to different factors. The decrease between Hart and Fetch is mostly due to an instruction fetch misses. The decrease between Fetch and Decode could be due to several factors. If a hart is locked due to an in-flight data miss, it is allowed to fetch the next instruction and forward it into decode stage. The decode stage is not allowed to read

the source operands and issue the instruction to the arbiter until the stream has been unlocked. Data hazards through the register file also stall instruction streams at decode stage. Finally, wrong-path instructions are filtered at decode, arbiter, and commit stages. Sub-figure 8.2d shows the distribution of instructions dropped due to the aforementioned reasons at decode, arbiter, and commit stages.



(a) `msort` IPC

(b) `msort` speedup

(c) `msort` pipeline instruction distribution

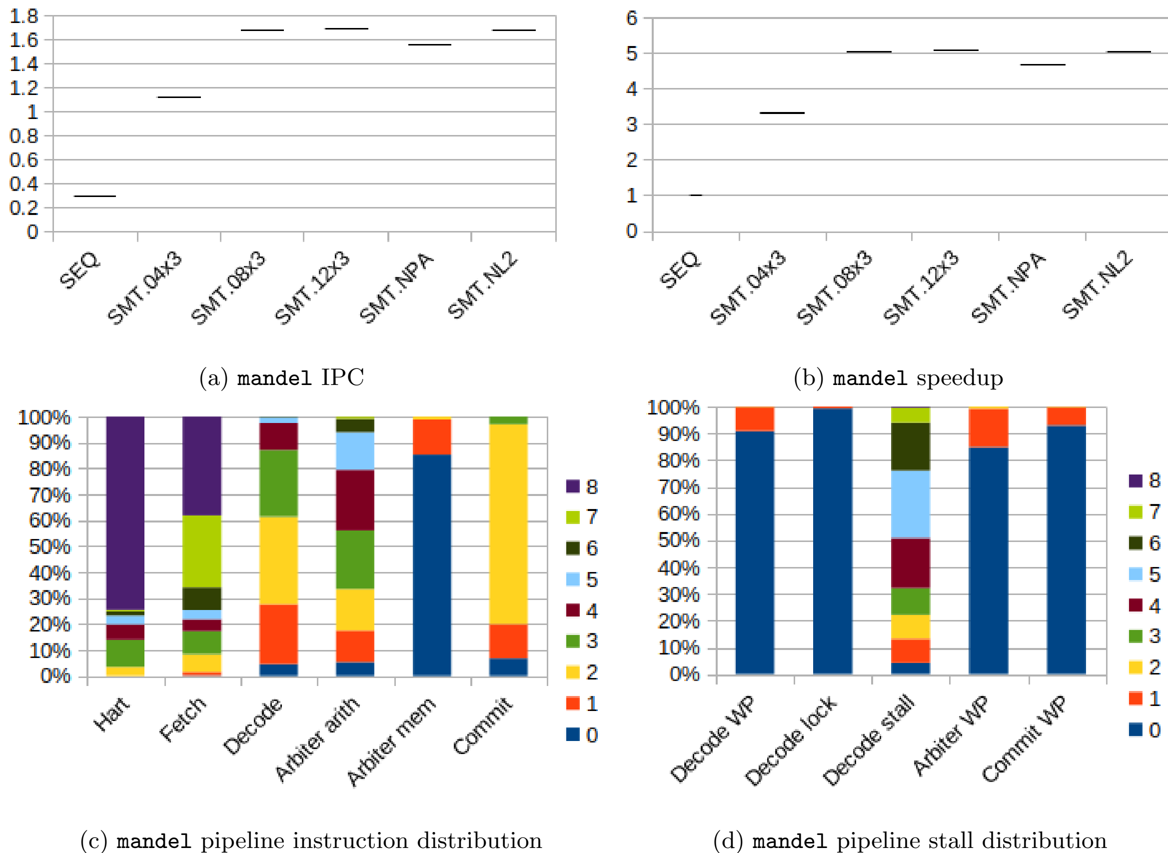
(d) `msort` pipeline stall distribution

Figure 8.2: `msort` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.3 Mandelbrot

Figure 8.3 shows the performance results for the Mandelbrot test. The first observation is that the execution time is very consistent as the box plots do not even appear as such. This makes sense as this problem does not actually depend on input data and the degree of parallelism extracted is always very similar. We can observe that the hart occupancy, although is not perfect, is fairly high as 75% of the time all harts are hosting a stream. We can observe a huge performance advantage when moving from SEQ into SMT.04x3 and a clear advantage when moving from SMT.04x3 to SMT.08x3. However, scaling up to SMT.12x3 shows almost no benefits. This is due to the fact that this algorithm is compute-bounded.

Most of the instructions executed are arithmetic and the SMT.08x3 configuration is already saturating the two arithmetic lanes. The evidence of that is that 82% of the time two or more instructions are ready to be issued into the backend at arbiter stage, saturating already the two arithmetic lanes. We can reach a similar conclusion by taking a look at the IPC. SMT.08x3 already gets close to the theoretical maximum for arithmetic instructions of 2 IPC. Reassuring that this program is compute-bound is the fact the decode stage shows almost no locks due to data cache misses, as there are almost no data memory accesses. It does show however a fairly high amount of stalls due to data hazards through the register file. We can also observe that the priority scheduling of the arbiter supposes a significant advantage when sifting through such a large amount of arithmetic instructions.



(a) `mandel` IPC

(b) `mandel` speedup

(c) `mandel` pipeline instruction distribution

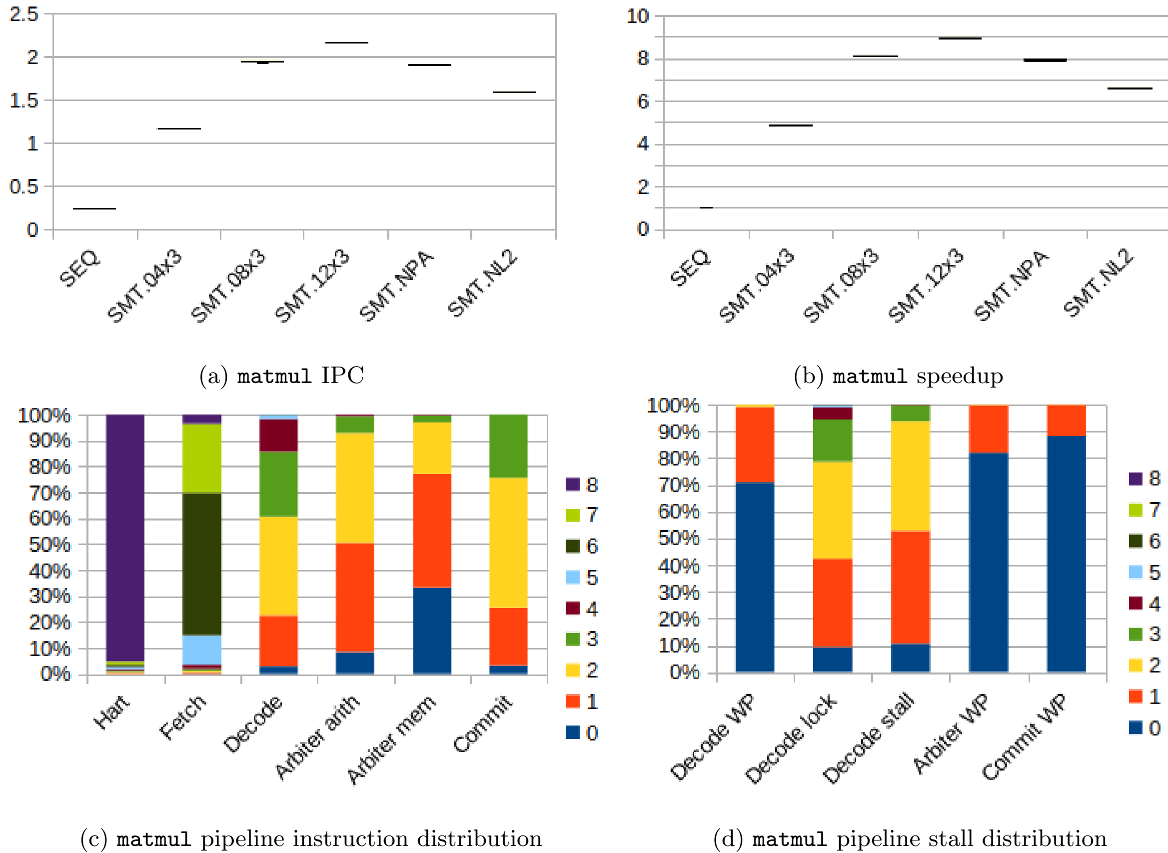
(d) `mandel` pipeline stall distribution

Figure 8.3: `mandel` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.4 Matrix multiply

Figure 8.4 shows the performance results for the matrix multiply test. In this test, the execution time is also very consistent. The code path followed by matrix multiply does not depend on the input data and therefore it always produces a similar parallelization degree. There is a single repetition with a slight decrease in performance in SMT.08x3. We suspect this might be due to the nondeterminism introduced by the DDR4 memory controller. The hart occupancy is almost perfect and 96% of the time all of the harts are occupied due to the embarrassingly parallel nature of the problem. SMT.04x3 and SMT.08x3 produce super-linear increases in performance. We believe those are caused by better overall cache locality. Having several instruction streams working on blocks of the result matrix that are close may generate better reuse of the input data. However, SMT.12x3 does not show such a noticeable performance uplift probably due to a memory bound.

Matrix multiply is an algorithm that can be highly memory-bounded. The SMT.08x3 configuration already saturates the single memory lane during 65% of the cycles. SMT.12x3 is capable of increasing the total throughput over SMT.08x3 but becomes bottlenecked by the memory pipeline. The decode stage suffers from many locks due to the large amount of data that is being processed by the program. Related to this, the removal of the second-level cache also decreases performance significantly. It also suffers from a large number of stalls due to data hazards as the computations of the inner loop generate many back-to-back pairs of instructions with data dependencies.



(a) `matmul` IPC

(b) `matmul` speedup

(c) `matmul` pipeline instruction distribution

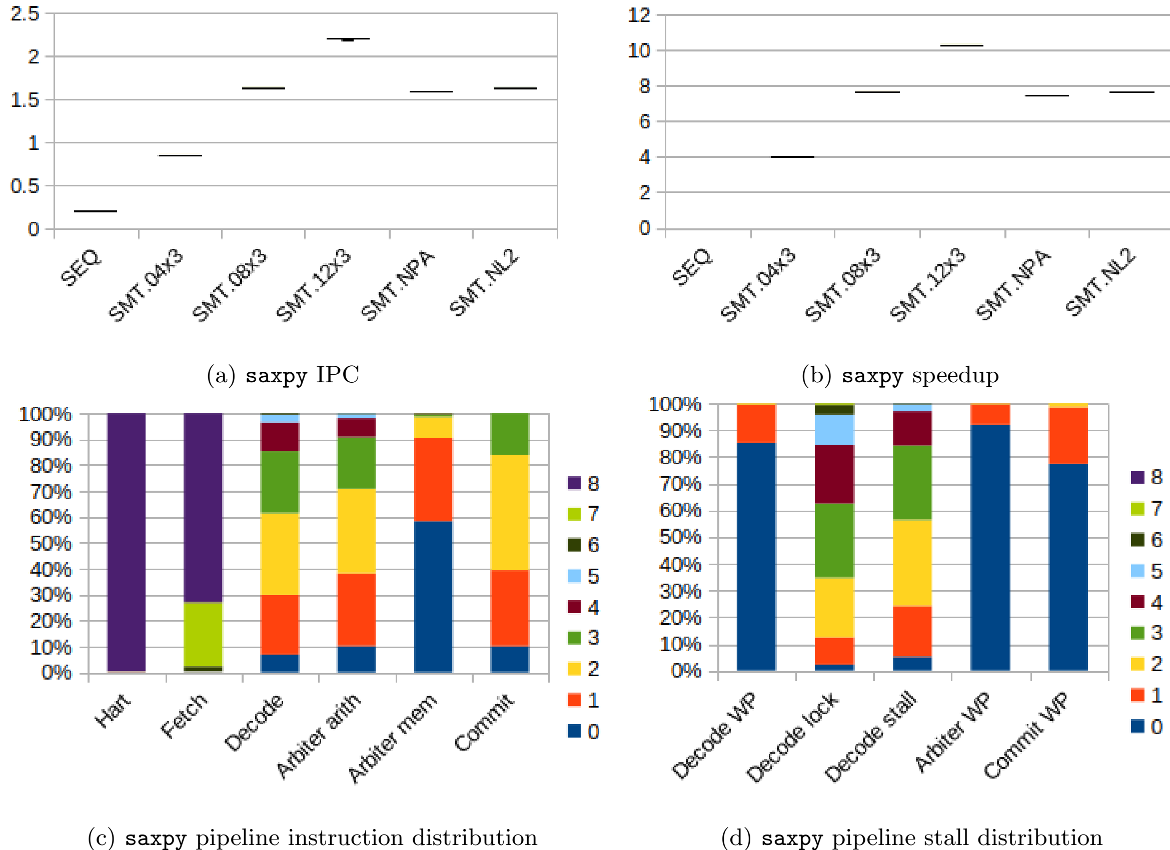
(d) `matmul` pipeline stall distribution

Figure 8.4: `matmul` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.5 SAXPY

Figure 8.5 shows the performance results for the SAXPY test. This program is embarrassingly parallel and all of the tasks created have the exact same execution time as the code path is never data-dependent. This makes it scale almost perfectly ranging from SMT.04x3 all the way to SMT.12x3, having complete utilization of all the harts and a very consistent execution time. We notice a slight improvement thanks to the priority arbiter, but not nearly as much as in Mandelbrot. Removing the second-level cache makes no difference. This is actually expected. SAXPY benefits from very short-term spacial reuse of data thanks to linearly accessing the arrays but will not benefit from temporal reuse of data nor longer-term spatial reuse. The L1D is enough to cover all the reuse available.

There are three main reasons why SAXPY scales so well when increasing the frontend width. First of all, the proportion of total arithmetic instructions to total memory instructions closely matches the 2 to 1 offered by the backend. Second of all, the code is tiny and the inner loop of the kernel actually fits within a single line. The fetch stage gets almost no misses from the individual L0Is and puts very little pressure on the L1I. Finally, what limits the performance of SAXPY is actually fetching the data from main memory. The more harts are added, the longer is the window available to wait on memory as more on-flight misses are supported. The evidence of that is that the execution is mostly bounded by the large number of locks at decode stage.



(a) `saxpy` IPC

(b) `saxpy` speedup

(c) `saxpy` pipeline instruction distribution

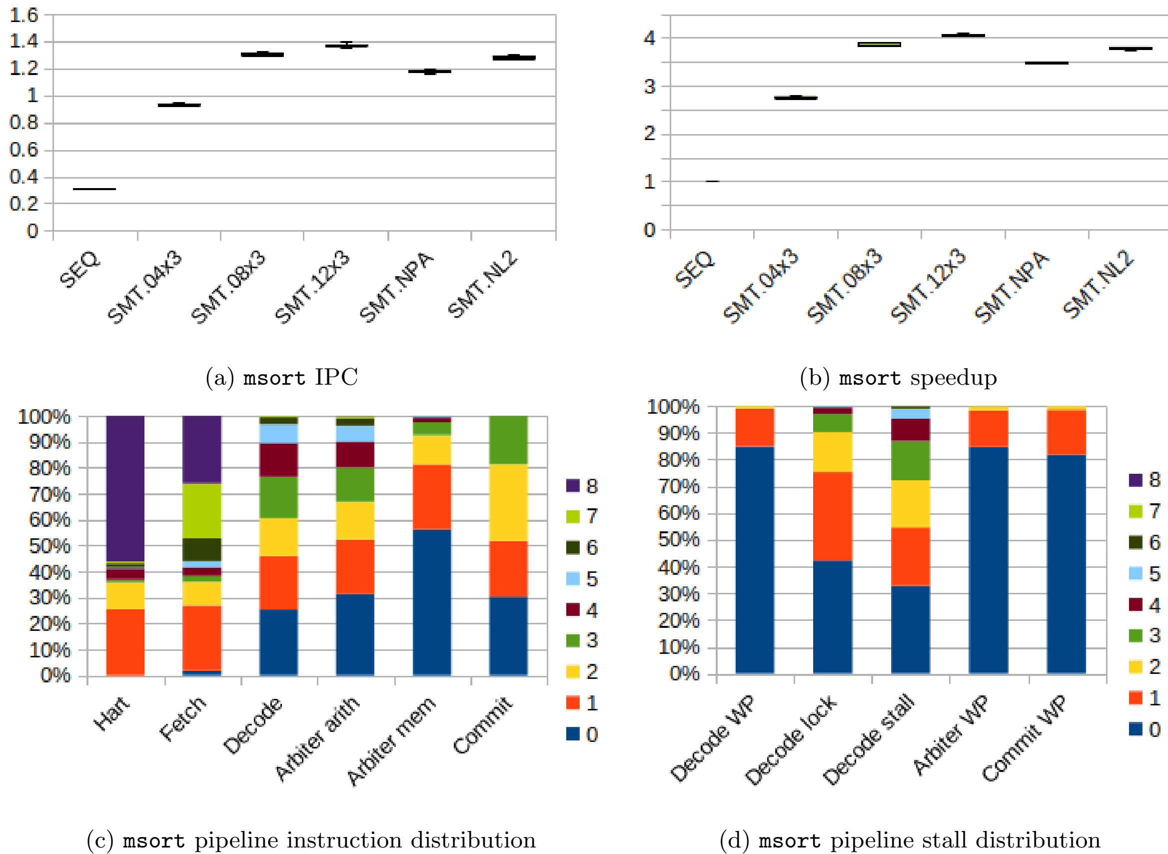
(d) `saxpy` pipeline stall distribution

Figure 8.5: `saxpy` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.6 Mergesort

Figure 8.6 shows the performance results for the mergesort test. The first thing that makes it stand out when compared with the previous test is the much lower hart usage. The parallelism in mergesort is discovered as the recursion tree is explored. Therefore, the first few levels of recursion offer limited parallelism. Those first levels spend a significant amount of time executing the linear array merge. A significant region of the execution has therefore very limited parallelism and does not saturate all the available harts. There is a variation in execution time among runs. The code path in mergesort is slightly data-dependent and changes in the input array slightly affect overall execution time.

Mergesort scales relatively well into SMT.04x3 and SMT.08x3 but shows limited improvement with SMT.12x3. This is in great part bounded by the fact that some regions of the execution have limited parallelism. Adding extra harts accelerates only the execution of the highly parallel regions. The improvement that we do observe is mostly due to the increased amount of on-flight misses supported in the highly parallel regions. Mergesort does benefit significantly from the arbiter priority scheduling as has many control-flow operations that would be otherwise forwarded into the backend.



(a) `msort` IPC

(b) `msort` speedup

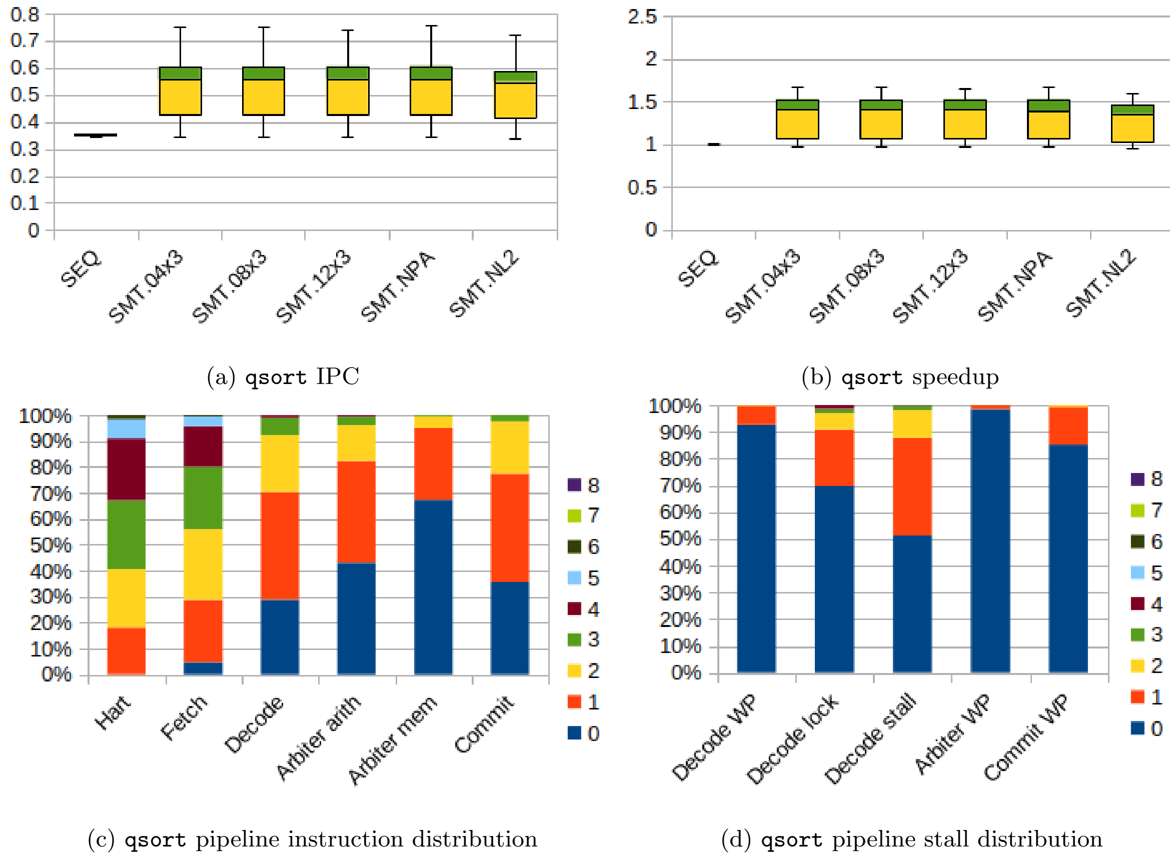
(c) `msort` pipeline instruction distribution

(d) `msort` pipeline stall distribution

Figure 8.6: `msort` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.7 Quicksort

Figure 8.7 shows the performance results for the quicksort test. Quicksort is the only test that has a significant variation in execution time depending on the data. The hart usage and the overall performance improvements are poor. Both things are evidence of the same fact. Most of the parallelism in quicksort is discovered as the recursion tree is traversed. The shape of the recursion tree in quicksort is highly data-dependent and some inputs make it very unbalanced. This generates very unbalanced tasks that do not take the same amount of time to execute; resulting in regions with very limited parallelism. The pressure put on the backend is not significant. The number of stalls and locks is also limited. This is a consequence of the limited parallelism offered by this test. The only real difference in results among hardware configurations is the decrease in performance without the L2 cache.



(a) `qsort` IPC

(b) `qsort` speedup

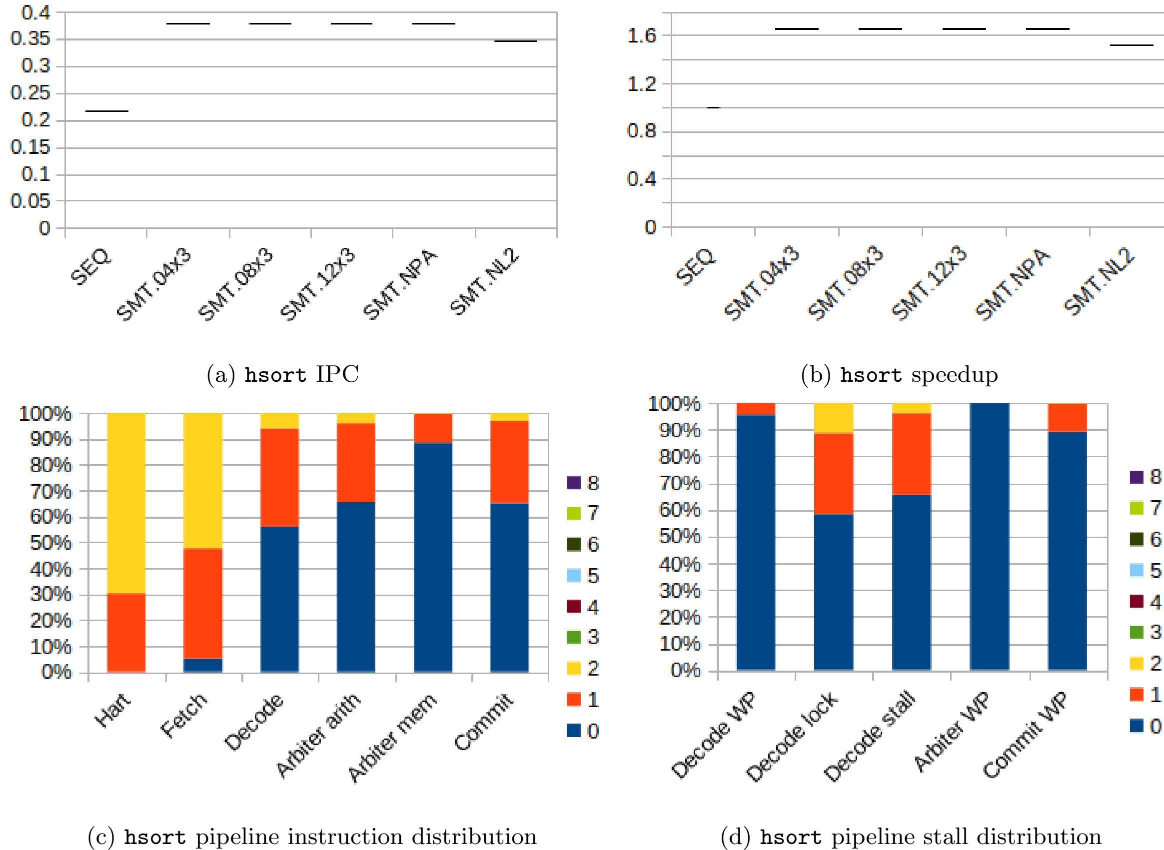
(c) `qsort` pipeline instruction distribution

(d) `qsort` pipeline stall distribution

Figure 8.7: `qsort` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.2.8 Heapsort

Figure 8.8 shows the performance results for the heapsort test. The heapsort parallelization proposed in this work offers very limited parallelism. By construction, there are never more than two simultaneous instruction streams. Notice that 70% of the time two harts are in use. This shows that the limited parallelism that does exist is being taken advantage of and offers a significant speedup. However, as there are never more than two simultaneous threads, heapsort does not benefit at all from wider frontend configurations. It does not benefit from priority scheduling either, as the pressure put on the backend is very small. It does benefit from the second-level cache, however, as it has a significant amount of data reuse.



(a) `hsort` IPC

(b) `hsort` speedup

(c) `hsort` pipeline instruction distribution

(d) `hsort` pipeline stall distribution

Figure 8.8: `hsort` IPC, speedup and pipeline distributions across different SMT core configurations. The IPC and speedups are shown as box plots on 10 repetitions of randomly generated data sets. The data sets are pre-generated and carried over the different core configurations. The distributions show the metrics of a single representative of the aforementioned 10 iterations in SMT.08x3.

8.3 Summary

The arbiter with a bitonic sorting network scales well and meets timing up to 16×16 configurations. However, the larger configurations already start showing a significant impact in area. We have synthesized the whole core with up to 12 harts. The performance tests show that this is enough to saturate the execution resources of the 3-lane backend if enough parallelism is discovered by the algorithm. A configuration with more than 12 harts does therefore not make sense unless the backend throughput is increased as well. Increasing the arithmetic instruction throughput in this design is simple, as the number of arithmetic lanes in the backend is parameterized. The memory instruction throughput, however, cannot be easily increased and has become the bottleneck in memory-bound tests.

Each test has shown a slightly different bottleneck when it comes to overall performance. Mandelbrot already saturates the arithmetic throughput with the 8-hart configuration. Matrix multiplication almost saturates the single memory lane with the 8-hart configuration as well. Mergesort does not increase performance significantly with the larger configuration either. Quicksort and heapsort are bottlenecked by that lack of parallelism discovered by the program rather than the hardware itself, and the smaller 4-hart configuration already provides the maximum observed speedup for them. SAXPY is the only benchmark that clearly benefits from the 12-hart configuration, but the increase in performance comes mostly from the increase in supported in-flight misses.

The mid-size 8-hart configuration shows a significant advantage over the 4-hart configuration and already provides enough instruction throughput to saturate the backend in most situations if enough parallelism is discovered by the program. The 12-hart configuration, however, mostly provides the benefit of supporting more in-flight misses, and this is only taken advantage of by SAXPY. Although all SMT configurations manage to partially hide the first-level misses, including a second-level cache still shows an advantage on several of the benchmarks. The arbiter with priority scheduling also offers a performance benefit in the tests that manage to saturate the backend. All in all, we believe that the SMT.08x3 is the most balanced configuration across the range with the only major limitation being the maximum number of supported in-flight misses in SAXPY.

Chapter 9

Conclusions and future work

9.1 Conclusions

We have studied and implemented an SMT core that benefits from P-RISC capabilities. When compared to previous SMT research, our core provides a microarchitecture that is focused on efficiently sharing hardware resources when plenty of TLP is available. Our design minimizes the amount of hardware resources dedicated to improving single-threaded ILP and the overhead associated with adding more harts to the machine. We have achieved that by extending the base RISC-V ISA with Fork and Join instructions, adding the hardware necessary to manage many simultaneous continuations, adding an arbiter module to schedule the use of execution resources, and implementing an extension of the split-phase memory system proposed by P-RISC.

When compared to the original P-RISC research, we provide a microarchitecture that ports the model into a modern RISC-V ISA and interleaves the multiple continuations efficiently thanks to the SMT capabilities. Instead of having a single token queue that circulates all the continuations and executes their instructions one by one; several continuations are simultaneously hosted in the core and keep executing continuously until they stall due to a long latency memory access. We combine the split-phase memory system with a caching system. The first level of caching reduces the amount of split-phase accesses and thus lowers both the amount of instruction stream stalls and the pressure put on the split-phase memory system. The second level of caching increases the performance of the split-phase memory itself. Although the way we interpret and use them is unconventional, the actual implementation of the caches themselves is not that different from standard ones.

On top of the hardware implementation proposed, we provide a few tools that have facilitated its development and testing. The system implemented can be synthesized into RTL simulation, FPGA, and ASIC environments. We modified the Spike ISA simulator to make it suitable for the verification of hardware that implements the P-RISC model and instructions. On top of the hardware, we built a basic API that abstracts the Fork and Join instructions with C primitives and implemented a few programs with them. The programs have been used as a benchmark to test the performance of the machine and the parallelization capabilities of P-RISC.

We have tested and obtained experimental results related to both ASIC synthesis and software performance benchmarks. The core can be synthesized with 8 harts at 1.1GHz and $390000\mu m^2$ of area, and scales up to 12 harts at 1.07GHz and $520000\mu m^2$ of area. The increase in total area observed per each added hart is relatively modest at $27600\mu m^2$ and comparable to 6% of a 64-bit refined in-order core or 3% of a larger 64-bit out-of-order core - although 64-bit ISAs and extensions that add more architectural state could become problematic in our architecture. The software benchmarks have shown speedups ranging from 1.4× to 10× in relation to the execution of the equivalent sequential code in our hardware. We have observed two main conditions that determine why some programs are able to saturate the execution resources and achieve large speedups while others do not. The first one is related to the parallelism discovered in the algorithm itself. If the amount of parallel tasks discovered is poor

or very unbalanced, a large portion of the execution becomes lightly threaded and does not saturate the execution resources. The second one is the number of in-flight misses. Our microarchitecture limits the amount of first-level misses to one instruction miss and one data miss per hart. Programs that are mostly memory-bound can be restricted by this and will suffer from many locks associated with memory accesses.

9.2 Future work

The hardware implemented in this work provides very limited synchronization mechanisms. The only atomic memory operation is the Join instruction. The single AMO that it implements is very simple and is always linked with the potential termination of the associated stream. We believe that the A extension for RISC-V ISAs will be necessary for future work. This extension defines a standardized set of atomic memory operations and load-reserved and store-conditional constructs. If that extension is included, the option of substituting the Join instruction with a more flexible *Terminate* instruction should be considered. The Terminate instruction would not execute an AMO, but rather directly terminate the stream of instructions if a more generic register-based condition is fulfilled - with semantics and encoding similar to those of branch instructions. This would allow us to decouple the semantics of Join into two separate instructions and offer more synchronization options. Still, a standardized AMO followed by a Terminate instruction would mimic Join semantics.

The current Fork and Forkr instructions implement a register set copy with very coarse granularity. Every new stream of instructions receives a copy of the full register set of the caller. We only include an integer register set, but a more comprehensive implementation will also include floating-point registers and CSRs. The amount of architectural state that an RV64G machine includes is much larger than that of our RV32IM machine. Maybe not all new streams need to receive copies of all of it. In that case, two interesting options to consider would be the lazy load/store of part of the architectural state or fork instructions that can provide finer control over what parts of the register sets are transferred to the new continuation. Having finer control over the architectural state transfer could not only make forking and scheduling new streams faster but also reduce the total amount of physical registers that a microarchitecture must implement. This would make larger core configurations with many harts easier to justify as the overhead per extra hart would be lower.

In order to make the P-RISC SMT architecture easier to justify and adopt, it must be ensured that the P-RISC SMT cores are compatible with other pieces of hardware. There are two key points to consider to integrate such a core into a larger system. First, the caching and memory systems must be compatible. The first-level caches implemented in this work are specialized. However, if they were modified in order to implement a standardized coherence protocol, the rest of the hierarchy can most likely be substituted by a standard memory subsystem. Second, the system must provide a place to store continuations and schedule them among several cores in multi-core configurations. It is not clear yet what should be the interface for that, but it is also important to solve this problem in order to facilitate the adoption of P-RISC.

The software integration of P-RISC is even more challenging than hardware integration. Modern operating systems already include the notion of threads and processes - e.g. POSIX Threads. It is not clear how the multi-stream nature of P-RISC should be integrated into an operating system. This architecture could either be used to accelerate the creation and scheduling of threads by the operating system or add a finer, user-defined, tear of TLP within the existing thread and process notions. On top of that, it should be considered how higher-level parallel paradigms - like OpenMP or Cilk - could make use of P-RISC in order to expose an expressive task-based model but reduce the runtime overhead thanks to the native fork and join constructs. Maybe P-RISC architectures should also provide mechanisms that allow runtime software to better infer the current load on the machine in order to balance the granularity and discovery of parallelism. The software integration and usability of P-RISC SMT architectures still have many unknowns that must be studied.

Bibliography

[NA89] R. S. Nikhil and Arvind. "Can Dataflow Subsume von Neumann Computing?" In: *SIGARCH Comput. Archit. News* 17.3 (Apr. 1989), pp. 262–272. ISSN: 0163-5964. DOI: 10.1145/74926.74955. URL: https://doi.org/10.1145/74926.74955.

[Sez93] André Seznec. "A case for two-way skewed-associative caches". In: *ACM SIGARCH computer architecture news* 21.2 (1993), pp. 169–178.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing on-Chip Parallelism". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA '95. S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, pp. 392–403. ISBN: 0897916980. DOI: 10.1145/223982.224449. URL: https://doi.org/10.1145/223982.224449.

[Tul+96] Dean M Tullsen et al. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor". In: *Proceedings of the 23rd annual international symposium on Computer architecture*. 1996, pp. 191–202.

[Lo+97] Jack L Lo et al. "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading". In: *ACM Transactions on Computer Systems (TOCS)* 15.3 (1997), pp. 322–354.

[Eme+99] Joel Emer et al. "Simultaneous multithreading: Multiplying alpha's performance". In: *Microprocessor Forum*. Vol. 23. 1999.

[VL00] Steven P. Vanderwiel and David J. Lilja. "Data Prefetch Mechanisms". In: *ACM Comput. Surv.* 32.2 (June 2000), pp. 174–199. ISSN: 0360-0300. DOI: 10.1145/358923.358939. URL: https://doi.org/10.1145/358923.358939.

[TB01] D.M. Tullsen and J.A. Brown. "Handling long-latency loads in a simultaneous multithreading processor". In: *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. 2001, pp. 318–327. DOI: 10.1109/MICRO.2001.991129.

[Esp+02] Roger Espasa et al. "Tarantula: A Vector Extension to the Alpha Architecture". In: *SIGARCH Comput. Archit. News* 30.2 (May 2002), pp. 281–292. ISSN: 0163-5964. DOI: 10.1145/545214.545247. URL: https://doi.org/10.1145/545214.545247.

[Mar+02] Deborah T Marr et al. "Hyper-Threading Technology Architecture and Microarchitecture." In: *Intel Technology Journal* 6.1 (2002).

[TT03] Nathan Tuck and Dean M Tullsen. "Initial observations of the simultaneous multithreading Pentium 4 processor". In: *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2003, pp. 26–34.

[KST04] R. Kalla, Balaram Sinharoy, and J.M. Tendler. "IBM Power5 chip: a dual-core multithreaded processor". In: *IEEE Micro* 24.2 (2004), pp. 40–47. DOI: 10.1109/MM.2004.1289290.

[VD05] H. Vandierendonck and K. De Bosschere. "XOR-based hash functions". In: *IEEE Transactions on Computers* 54.7 (July 2005), pp. 800–812. ISSN: 1557-9956. DOI: 10.1109/TC.2005.122.

[Moo+06] K.E. Moore et al. "LogTM: log-based transactional memory". In: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* 2006, pp. 254–265. DOI: 10.1109/HPCA.2006.1598134.

[FC08] D.J. Sorin F.A. Bower and L.P. Cox. "The impact of dynamically heterogeneous multicore processors on thread scheduling". In: *Micro, IEEE, 28* (2008).

[Lin+08] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[Blu10] Inc Bluespec. "Bluespec SystemVerilog Reference Guide". In: (2010). URL: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf.

[But+11] Michael Butler et al. "Bulldozer: An Approach to Multithreaded Compute Performance". In: *IEEE Micro* 31.2 (2011), pp. 6–15. DOI: 10.1109/MM.2011.23.

[Sez11] André Seznec. "A New Case for the TAGE Branch Predictor". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: Association for Computing Machinery, 2011, pp. 117–127. ISBN: 9781450310536. DOI: 10.1145/2155620.2155635. URL: https://doi.org/10.1145/2155620.2155635.

[MTA12] Rene Mueller, Jens Teubner, and Gustavo Alonso. "Sorting networks on FPGAs". In: *The VLDB Journal* 21.1 (Feb. 2012), pp. 1–23. ISSN: 0949-877X. DOI: 10.1007/s00778-011-0232-z. URL: https://doi.org/10.1007/s00778-011-0232-z.

[MHS14] Daniel Molka, Daniel Hackenberg, and Robert Schöne. "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer". In: *Proceedings of the Workshop on Memory Systems Performance and Correctness*. MSPC '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450329170. DOI: 10.1145/2618128.2618129. URL: https://doi.org/10.1145/2618128.2618129.

[KHA15] Myron King, Jamey Hicks, and John Ankcorn. "Software-Driven Hardware Development". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 13–22. ISBN: 9781450333153. DOI: 10.1145/2684746.2689064. URL: https://doi.org/10.1145/2684746.2689064.

[Sin+15] B. Sinharoy et al. "IBM POWER8 processor core microarchitecture". In: *IBM Journal of Research and Development* 59.1 (2015), 2:1–2:21. DOI: 10.1147/JRD.2014.2376112.

[Asa+16] Krste Asanovic et al. "The rocket chip generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[Jef+16] Mark C Jeffrey et al. "Unlocking ordered parallelism with the Swarm architecture". In: *IEEE Micro* 36.3 (2016), pp. 105–117.

[Col17] Caroline Collange. "Simty: generalized SIMT execution on RISC-V". In: *CARRV 2017 - 1st Workshop on Computer Architecture Research with RISC-V*. Vol. 6. First Workshop on Computer Architecture Research with RISC-V. Boston, United States, Oct. 2017, p. 6. URL: https://inria.hal.science/hal-01622208.

[Zha+18] Sizhuo Zhang et al. "Composable Building Blocks to Open up Processor Design". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 68–81. DOI: 10.1109/MICRO.2018.00015.

[WA19a] Andrew Waterman and Krste Asanovic. "The risc-v instruction set manual, former Chapter 18: Calling Convention". In: *EECS Department, UC Berkeley.* (2019).

[WA19b] Andrew Waterman and Krste Asanovic. "The risc-v instruction set manual, volume I: Base user-level ISA". In: *EECS Department, UC Berkeley.* (2019).

[ZB19] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. DOI: `10.1109/TVLSI.2019.2926114`.

[AS20] Maleen Abeydeera and Daniel Sanchez. "Chronos: Efficient speculative parallelism for accelerators". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1247–1262.

[Bou+20] Thomas Bourgeat et al. "The Essence of Bluespec: A Core Language for Rule-Based Hardware Design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. ISBN: 9781450376136. DOI: `10.1145/3385412.3385965`. URL: `https://doi.org/10.1145/3385412.3385965`.

[KK20] Jagadish B. Kotra and John Kalamatianos. "Improving the Utilization of Micro-operation Caches in x86 Processors". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 160–172. DOI: `10.1109/MICRO50266.2020.00025`.

[Sor+22] Víctor Soria-Pardos et al. "Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI". In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pp. 254–261. DOI: `10.1109/DSD57027.2022.00042`.

[Gup+] Anupam Gupta et al. "Scheduling Heterogeneous Processors Isn't As Easy As You Think". In: *Proceedings of the 2012 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1242–1253. DOI: `10.1137/1.9781611973099.98`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611973099.98`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9781611973099.98`.

[Hica] Jamey Hicks. *buildcache*. URL: `https://github.com/cambridgehackers/buildcache`. (accessed: May 2023).

[Hicb] Jamey Hicks. *fpgamake*. URL: `https://github.com/cambridgehackers/fpgamake`. (accessed: May 2023).

[SL] James Sanders and Wayne Lam. *RISC-V Reaches a Turning Point*. URL: `https://www.ccsinsight.com/blog/risc-v-reaches-a-turning-point/`. (accessed: April 2023).

[Sof] RISC-V Software. *Spike RISC-V ISA Simulator*. URL: `https://github.com/riscv-software-src/riscv-isa-sim`. (accessed: May 2023).

[Xil] AMD Xilinx. *VCU108*. URL: `https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html`. (accessed: April 2023).