Complexity-effective optimisations for RISC scalar pipelines

Matthew H. Else
King's College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for Part III of the Computer Science Tripos

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Matthew.Else@cl.cam.ac.uk

May 30, 2019

Declaration

I Matthew H. Else of King's College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11,982

Signed:

Date:

Abstract

Modern CPU pipelines, alongside modern compilers, make use of several optimisations that seek to improve computer program performance. The role of a computer architect is to balance the complexity of each individual instruction with (i) the number of instructions required to represent a certain task and (ii) the complexity of the hardware required to implement a particular instruction-set architecture (ISA). In essence, balancing the instruction-set complexity with hardware complexity.

This thesis describes a series of experiments that were performed to demonstrate the feasibility of complexity-effective modifications to a RISC-V CPU core, and builds upon prior work describing the potential role of macro-op fusion in partially bridging the gap between CISC and RISC instruction sets by executing two simple, dependent instructions in a single micro-op. A complete implementation of macro-op fusion in a classic five-stage RISC pipeline is described, which results in a statistically significant reduction in CPU cycles executed for a number of benchmarks, with no statistically significant performance regressions. Further enhancements relating to macro-op fusion are proposed, including the addition of a RISC-V macro-op fusion optimisation pass in LLVM, improving the speed-up of RISC-V applications targeting cores with macro-op fusion from 1.90 % to 2.76 %. Rocket's CoreMark score (in the absence of additional compiler optimisations) was increased by 1.11 %, with only a 1.51 % increase in core area.

To further demonstrate the potential benefits of complexity-effective optimisations, limit studies are described in which the potential speedup of a number of low hardware-cost optimisations are measured. This includes demonstrating that the frequency of trivial instructions, as first suggested by Yi and Lilja [1] was significantly more limited in the context of RISC-V, due to architectural differences when compared to the architecture described.

Finally, to push the boundaries of complexity-effective optimisations, a limit study was performed to investigate the potential impact of limited dual-issue on RISC-V, notably discovering a potential 5.38% reduction in effective instruction count by implementing dual-issue without additional register read and write ports, 6.95% with an additional read port and up to 21.1% with only a single additional register write port..

Acknowledgements

Thanks to my project supervisor, Robert Mullins for his dedicated and proactive approach to supervising, and to Alex Bradbury of LowRISC, in particular for his help with LLVM and seemingly encyclopaedic knowledge of RISC-V and the surrounding literature. Thanks to Will Miller for the significant amount of time he took to painstakingly proof-read this dissertation. Finally, thanks to Tom, Giulia, my mum, my dad and many other people who've put up with me throughout four years of Cambridge Computer Science up to this pont.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

'Complexity-effective' describes a class of potential CPU core modifications, in which a relatively simple hardware optimisation is implemented, with the aim of providing good 'value for money' performance gains for the hardware resources used, versus performance gained. This optimisation may, or may not, include a small amount of compiler support. However, the key aim of these optimisations is to improve performance in typical computing tasks, using only a small amount of additional core area, power consumption, or critical path length.

Although the ability of complex micro-architectural techniques, such as super-scalar out-of-order designs to produce large performance gains, is well documented, this project starts with a text-book five-stage reduced-instruction set computer (RISC) pipeline and aims to improve performance as far as possible with minimal additional complexity. A range of candidate optimisations are considered throughout this dissertation: macro-op fusion, trivial computations, 'dynamic fusion', and variations of limited dual-issue pipelines.

The 'Kiss rule for RISC microarchitecture design' [2] states that for a 1 % increase in core area due to a particular optimisation, there must be at least a corresponding 1 % improvement in performance. Each of the optimisations considered in this project aim to exceed this requirement; extracting as much additional performance as possible without a significant increase in hardware size or power consumption.

In the case of macro-op fusion, this will be evaluated by implementing the changes and evaluating their impact on the synthesised gate-level netlist, while dual-issue will be implemented in future work.

Macro-op fusion is a technique used in the design of high-performance processing cores, in which two (or more) instructions are combined together in the processor, and executed as a single micro-op by the back-end. This technique has been applied to Intel's line of high-performance consumer processors [3], SiFive's 7 Series [4] and Arm cores. In this thesis, I describe an implementation of macro-op fusion that uses micro-architectural modifications to the Rocket processor [5], an open source in-order core based on the RISC-V instruction-set architecture (ISA).

Inspired by Celio et al. [6], the initial goal of this project was to extend a RISC-V processor with macro-op fusion support. Celio describes how macro-op fusion, in combination with the RISC-V compressed extension (RVC) instructions, could negate the need to add more complex instructions to the RISC-V architecture. Idiomatic pairs of dependent instructions could be recognised dynamically at runtime, consuming the same or only slightly more space than a new instruction, leaving micro-architects with flexibility to make the trade-off between a simple CPU design, or a more complex design that supports macro-fusion. An alternative trade-off can be made in favour of CISC instruction-set architectures, using macro-op fission to split a complex instruction into many simpler micro-ops, reducing code-size and keeping the main pipeline simple by adding a complex decoder to the CPU frontend (as is the case in Intel's x86, and x86-64 CPUs).

To build on macro-op fusion, which simply relies on the static register allocation of instruction pairs, the second stage of this project considers dynamic opportunities for optimisation. 'Dynamic fusion' would allow processor to make optimisations based on runtime register values that could not be predicted at compile-time. Yi and Lilja [1] suggested that mathematical identities could be used to simplify $\sim$12 % of dynamic instructions in a typical processor [1], leading to an 8 % speedup in a typical processor.

I re-evaluate Yi and Lilja [1] using the RISC-V instruction-set architecture (ISA). This result is extended to demonstrate the theoretical maximum speedup available

in RISC-V when dynamic values of register values are used to optimise instructions at runtime.

The term 'dynamic' fusion is used throughout this dissertation to describe a hybrid approach between macro-op fusion, as previously described, and the use of trivial computations to fuse instruction pairs, depending on the dynamic register values read and modified by an instruction pair.

Finally, to push the bounds of complexity-effective CPU optimisations, a limit study into the potential speedup of limited dual-issue in a scalar RISC-V pipeline was developed, considering a variety of cases, both with and without a modified register file.

1.1 Motivation

RISC-V offers an ideal choice of target ISA, with a modern, well-considered design. Furthermore, the availability of Rocket as a processor design that is both open-source and used in industry by SiFive makes it a good choice of target processor.

The 'iron law' of CPU performance [7] allows CPU designers to precisely calculate the performance of a computer program, when running on a particular processor, in terms of (i) program instruction count, (ii) the processor's average cycle count per instruction, and (iii) the duration of each cycle. A number of optimisations can be made to improve these factors, including compiler optimisations to reduce instruction count and micro-architectural features, such as branch prediction and caching, to improve average CPI.

1.2 Highlights

The key results of Celio et al. [6] were reproduced, demonstrating the significant reduction in effective instruction count that macro-op fusion can provide. These results were then extended to a fully automated limit study, allowing analysis of a

cross-section of differing compilers and configurations. This allowed the effectiveness of compiler optimisations to be observed.

It was found that, on average a 4.28 %[1] instruction-count reduction is available by applying macro-fusion without compiler optimisations. This can theoretically increase to 7.08 % with ideal compiler optimisations.

Building upon the results obtained from the automated limit study, an optimisation pass was added to the back-end of LLVM, ensuring that macro-op fusion pairs are scheduled together after register allocation. This optimisation enabled a 36 % increase in fusion opportunities compared to baseline LLVM.

Rocket was modified to include a macro-op fusion optimisation as part of the decode stage of its textbook five-stage pipeline. This was evaluated using RTL simulations, and by synthesizing a netlist for a low-leakage 40 nm technology. The performance gained by Rocket was benchmark-dependent, however no statistically-significant regressions in performance were observed. For a reduced form of the 'rv8-bench' benchmark set (each benchmark was configured to complete in a reasonable length of time in simulation), an average speed-up of 0.79 % was observed. The performance gained by Rocket is smaller than the speed-up predicted by Spike due to caching; however, future improvements to the CPU front-end and running a longer-running set of benchmarks may allow some of this lost performance to be reclaimed.

Building upon the work of Yi and Lilja [1], a second series of experiments investigated potential improvements in CPU performance with the benefit of dynamic register values, from the perspective of trivial computations (i.e. instructions that at runtime are equivalent to a null operation), as well as the possibility of dynamic macro-fusion. The concept of 'trivial computations', described in the 2002 paper is shown not to translate well to the RISC-V ISA, as little can be done to simplify operations in RISC-V, unless the destination register is totally unaffected by an instruction. The relatively high proportion of dynamic zero and one values in registers, however, demonstrates that a significant amount of behaviour may be predictable, if this knowledge can be utilised. Section 4.2 then describes a the-

[1] SPECint compiler with unmodified GCC -O2

oretical hybrid with macro-op fusion, which allows the arithmetic simplifications proposed by Yi and Lilja to be applied to a number of cases in RISC-V.

Chapter 2

Background

This chapter describes the necessary background information required to understand all aspects of the project, describing the key resources: the RISC-V instruction set, the Rocket CPU and its hardware construction language (Chisel), and the compiler framework LLVM.

2.1 RISC-V

RISC-V is an open-source family of instruction-set architectures developed at UC Berkeley [8], which has been widely adopted in academia and industry as a standard platform upon which micro-architects, compiler developers, and others can experiment and collaborate. The combination of a 'production-ready' ISA with 32- 64- and 128-bit variants, mature compiler support in both GCC and LLVM, and an increasing range of tried-and-tested processor designs makes RISC-V an ideal basis for experiments in this project.

2.1.1 Instruction Set

The RISC-V instruction is so-called as it is the fifth major RISC ISA to be developed at UC Berkeley. Its original authors took advantage of the ability to build

a clean-slate ISA, so sought to remain true to principles of reduced instruction set computing, in contrast to the increasingly large ARM architecture, the most prominent commercial RISC ISA. Overall, RISC-V's developers tried to develop an ISA that was both high-performance and low-complexity.

The base RISC-V (integer) instruction set, known as RV[32/64/128]I provides a standard RISC load/store architecture with 31 general-purpose registers, on top of which a number of standard extensions have been defined. These extensions include: the RISC-V compressed instruction set (C), atomic (A), integer multiply/divide (M), single-precision floating point (F), and double-precision floating point (D). The set of extensions required to run Linux is RV(64/32)IMAFD; simply known as RV64G - the experiments performed in this project use RV64GC.

The base instruction set, as well as the standard extensions included in RV64G, consists of 32-bit wide instructions, while RVC consists of 16-bit wide instructions. Each compressed instruction corresponds to a single uncompressed instruction, allowing compression to be performed entirely during the assembly stage, which greatly simplifies hardware implementation. Support for RVC requires a CPU core to support misaligned instruction fetches (i.e., compressed and uncompressed instructions can either be 2-byte aligned or 4-byte aligned in memory, and intermingled freely throughout a program). While this additional hardware leads to a small amount of additional overhead, the increase in code density allowed by RVC compensates by significantly reducing code size. The pre-existing buffering hardware in Rocket enables many of the optimisations in this thesis to be implemented more easily, and with a smaller marginal cost.

2.1.2 Hardware Implementations

A wide range of hardware implementations of RISC-V have been developed, both in academia and industry. One such implementation is 'Rocket', which was initially developed at UC Berkeley [5] as a simple five-stage in-order single-issue core, now used by SiFive in its range of production-ready RISC-V cores. The 'Berkeley out-of-order machine' (BOOM), in contrast, is a high-performance super-scalar out-of-order core [9].

Both of these cores are developed in the hardware construction language 'Chisel' [10], a domain-specific language built in Scala. Chisel's goal is to increase the productivity of hardware developers, and ultimately produces standard Verilog code that has allowed both BOOM and Rocket to be taped-out to hardware multiple times. Chisel also enables developers to quickly simulate designs using Verilator.

In industry, high-performance open-source RISC-V cores have been developed by Western Digital[1]. Nvidia plans to replace its controller chips with RISC-V cores in the near future[2].

The combination of a well-proven design and convenient development made Rocket the preferred choice of target development platform for this project. Its design is discussed in §2.3.

2.1.3 Spike: RISC-V ISA Simulator

Although moderately high-performance simulators for RISC-V can be produced using the RTL of a real core,benchmarks were simulated using a high-performance ISA simulator built in C++, known as Spike[3], to quickly demonstrate the potential performance benefits of several optimisations. In combination with other open-source tools[4], this simulator allowed benchmarks, such as SPECint [11] and rv8-bench [12], to be quickly evaluated using program-counter histograms produced by Spike.

Since Spike supports a wide range of RISC-V extensions and can be easily modified, it will be used throughout to perform dynamic analyses of RISC-V machine code. The RISC-V 'proxy kernel'[5]additionally enables Linux system calls to be emulated, allowing, for example, standard output to be forwarded from simulated code.

[1] `https://www.anandtech.com/show/13964/western-digitals-riscv-swerv-core-released-for-free`

[2] `https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf`

[3] `https://github.com/riscv/riscv-isa-sim`

[4] `https://github.com/ccelio/speckle`

[5] `https://github.com/riscv/riscv-pk`

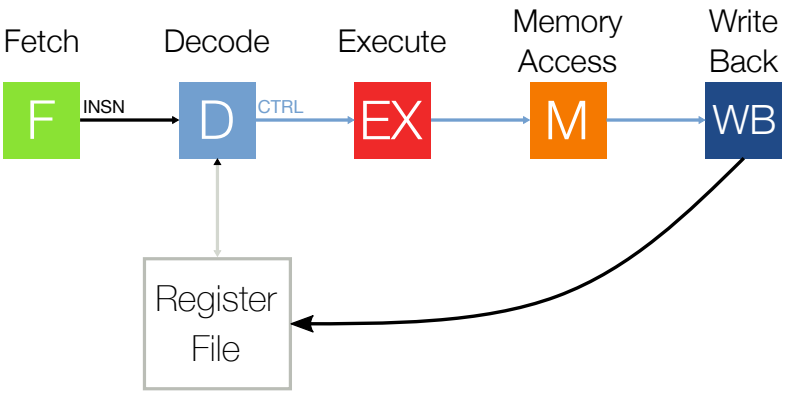


Figure 2.1: Textbook five-stage RISC pipeline

2.2 Processor Optimisations

The textbook RISC pipeline, illustrated in Fig. 2.1, breaks up the handling of instructions into five stages: fetch, decode, execute, memory-access and write-back. This separation provides numerous opportunities for optimisation, and allows hardware implementations to reach higher clock speeds by reducing the amount of processing done in each clock cycle.

2.2.1 Forwarding Network

To reduce the impact of data hazards (i.e., a data dependency from one instruction to the next), a forwarding network is used: multiplexing between a value read from the register file and the forwarding network as the operands are prepared in the decode stage for the execute stage of the pipeline (illustrated in Fig. 2.2. Crucially, this forwarding network may enable future optimisations, as described in §4.3.3. However, in dual-issue/super-scalar pipelines, described below, additional forwarding hardware is required, as multiple results may be produced per cycle.

The ability of a CPU pipeline to forward results from later stages in the pipeline to earlier stages is built upon by the ELM project, in which explicit operand registers are used to store ephemeral values. This avoids the need to write values back to the global register file [13, 14] and allows data to remain local to each

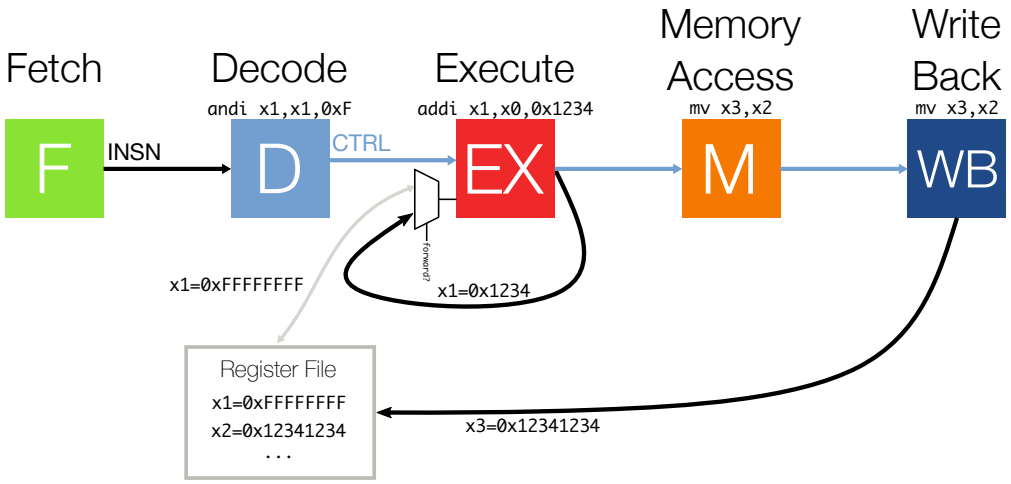


Figure 2.2: Textbook five-stage RISC pipeline, with forwarding illustrated. Note that the result of `addi x1,x0,0x1234` has not been written back to the register file, but its result is already available for the following, dependent instruction.

functional unit. The use of explicit operand registers allows software to more effectively communicate register re-use and short dependencies. In the context of this project, explicit operand registers may be applicable to the investigation of limited dual-issue, and could create a resource-efficient way to produce the required source registers for a pair of instructions.

2.2.2 Macro-op fusion

The concept of macro-op fusion allows micro-architects to optimise the evaluation of a multi-instruction sequence by evaluating them as a single 'fused' micro-op. Processors using this optimisation typically allow the sequence of instructions to be executed in a single cycle, rather than multiple cycles expected. As technology improves and priorities change over time, macro-op fusion allows micro-architects to alter their optimisations in response to changing trade-offs.

On the other hand, for RISC architectures, especially those aiming to cover a wide range of performance targets, such as RISC-V, macro-op fusion allows designers to be restrictive about the base ISA. This allows small implementations to be as small

as possible, while allowing micro-architects to deliberately introduce complexity if performance targets require the additional hardware.

2.2.3 Super-scalar architecture

Bearing in mind the 'iron law of CPU performance' mentioned previously [15] and illustrated in equation 2.1, there are numerous ways to improve the performance of programs running on a CPU. The first term in equation 2.1 describes the number of dynamic instructions executed during the evaluation of a program. The dynamic instruction count is influenced by compiler optimisations, choice of ISA and certain runtime optimisations such as macro-op fusion. Optimisations to the circuit-level design, and RTL optimisations can improve the clock speed. However, to improve the second term we must (on average) execute more instructions per cycle. This constraint requires micro-architects to design CPUs capable of exploiting instruction-level parallelism, prime examples of which are super-scalar cores.

$$\frac{1}{\text{performance}} = \frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{time}}{\text{cycle}} \tag{2.1}$$

Super-scalar micro-architectures are commonly seen in high-performance computers that are relatively unconstrained with respect to power consumption. These cores include multiple 'functional units', each of which is capable of executing a class of instructions independently of other functional units. The simplest case of a super-scalar core is a dual-issue in-order core, in which up to two independent instructions per cycle are issued, then executed by functional units, and results forwarded analogously to Fig 2.2[6].

However, super-scalar cores can be far more complex, making use of out-of-order execution to take advantage of instruction-level parallelism. By scheduling instructions dynamically, the data-path can respond to events that are unpredictable at

[6]The forwarding paths are significantly more complex in an in-order super-scalar system, as multiple results may be produced simultaneously.

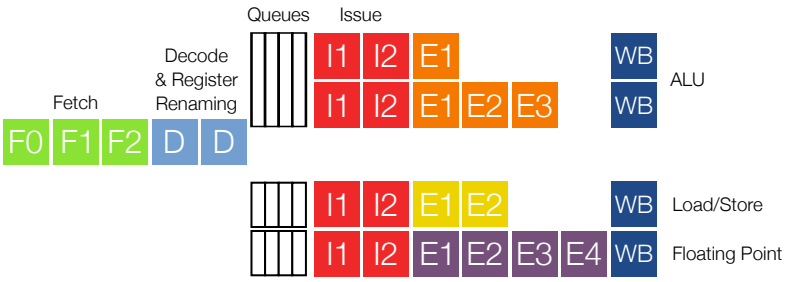


Figure 2.3: Example of a super-scalar micro-architecture, using register renaming for out-of-order, speculative execution. Based on the BOOMv2 micro-architecture [17]

compile time, such as cache misses or branch prediction. The use of register renaming helps to eliminate false dependencies, particularly in ISAs with small register files, such as x86. A typical high-performance super-scalar design is illustrated in Fig. 2.3

The out-of-order techniques aren't applicable to this project due to their significant overhead. Nevertheless, restricted versions of these techniques may have a place in complexity-effective optimisations, as suggested with the use of register renaming by Patsidis et al. [16].

2.2.4 Additional considerations

When making optimisations to CPU pipelines, there are number of potential pitfalls and corner-cases that must be considered. Additionally, micro-architects must ensure that designs are evaluated consistently by using a common set of benchmarks.

Precise exception handling

Exceptions may be produced at multiple points during the RISC pipeline, for example illegal instructions detected in the decode stage and misaligned loads detected during the execute stage. The intuitive purpose of precise exception handling is that if an exception happens, the CPU can vector to a 'trap handler',

which handles the exception, allowing execution to continue as though the faulting instruction had never happened. Faulting instructions can even be emulated in software (as in the case of misaligned loads and stores), and return control to the following instruction as though misaligned loads were supported natively by the hardware.

It is important when adding optimisations to the RISC pipeline that precise exceptions are maintained. Notably, during the development of the macro-op fusion extension to Rocket, as described in §4.1, care had to be taken to ensure that otherwise unused intermediate values were visible to any trap handlers in the event of a faulting fused load pair. Similarly, if the dual-issue experiments presented in §4.3.3 are implemented in hardware, much care is needed to ensure the correct ordering of exceptions and to ensure that intermediate results are written back to the register file.

Benchmarking

To reliably compare performance across a range of CPU designs, benchmarks are typically used to provide a reproducible value that can be compared between different CPU architectures and operating systems. To test the CPU pipeline on its own, CoreMark and Dhrystone are typically used, with benchmark values reported by these benchmarks are reported in CM/MHz and DMIPS/MHz, respectively. However, both CoreMark and Dhrystone are 'synthetic' benchmarks, i.e., programs written specifically for the purpose of benchmarking.

To provide a more realistic picture of everyday applications, the SPEC integer benchmark set is used throughout this project. SPEC's integer benchmark set consists of several compute-heavy real-world applications, including compilers such as the GCC compiler, string handling in Perl, compression using BZip, h.264, etc. The SPECint benchmark suite allows a broader picture of computer performance to be evaluated, including caching, beyond the CPU pipeline, as is the case in Dhrystone and CoreMark. However, the SPEC benchmark is certainly not representative of all workloads, and ultimately the optimal micro-architecture will depend subtly on the exact use-case of the CPU core.

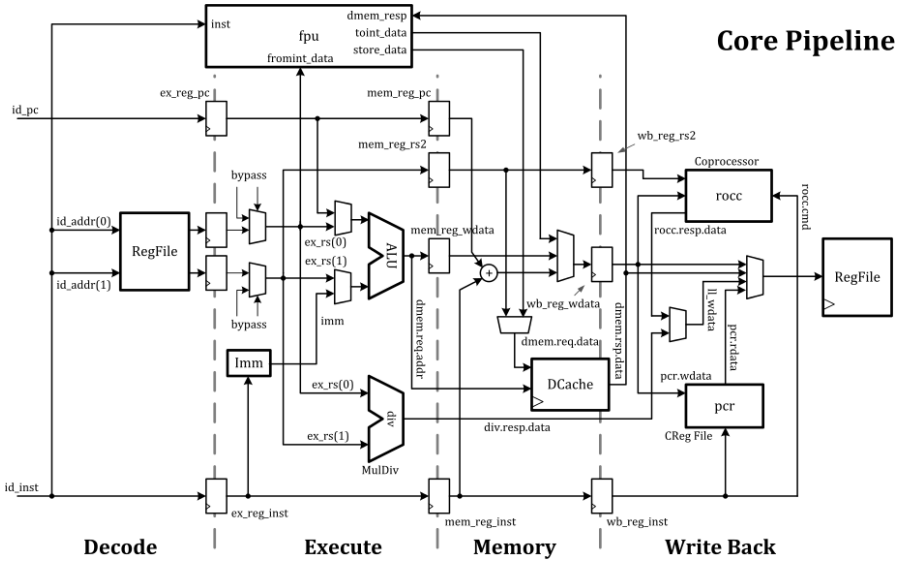


Figure 2.4: Baseline Rocket pipeline, reproduced from
`https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/`

2.3 Rocket

Rocket is an in-order, scalar, five-stage RISC pipeline designed at Berkeley used throughout this project for hardware development and as a reference design for a production-quality in-order processor. Rocket is an ideal candidate for the experiments undertaken in this project, as it implements both 32- and 64-bit RISC-V instruction-set architectures (ISA's), including the base ISA, and standard extensions: A, M, F and D (RV32G and RV64G).

Rocket is actively developed in industry at SiFive and in academia at UC Berkeley, and has a mature code-base developed in the 'hardware construction language' Chisel [10]. It has been taped out to silicon over fourteen times [18, p. 44], has a wide range of tests, allows for straight-forward software simulation, and hence offers a reliable choice of target core. A diagram of the CPU pipeline is shown in Fig. 2.4.

2.4 LLVM

LLVM is an intermediate language and compiler framework, supporting multi-stage optimisation, and the development of novel compiler technologies by offering multiple reusable components with clearly defined interfaces, which can easily be reused in a number of different contexts [19]. In combination with the Clang front-end for the C and C++ programming languages, LLVM provides an ideal framework for experimenting with compiler optimisations for the CPU improvements made throughout this project. Specifically, LLVM is used to develop a compiler optimisation for macro-op fusion, as described in §4.1.3

Chapter 3

Related Work

Computer architecture often involves a trade-off, between power, core area, complexity and performance. A study by Azizi et al. [20] sought to quantify this, and demonstrated that at each power and performance target had a corresponding optimal microarchitecture. Illustrated in Fig. 3.1, this shows that, as may be intuitively expected, small, simple cores are most effective for low power and performance requirements, with complex out-of-order cores dominating the high-end range. A second trade-off is demonstrated by Celio et al. [6], in which the use of macro-op fusion in RISC-V is proposed as a way to simplify the trade-off between ISA and micro-architectural complexity. These two key papers motivate the majority of work described in this project, and this chapter provides an overview of these, followed by key papers and designs in the three target areas of this project: macro-op fusion, 'dynamic fusion' and dual-issue.

3.1 Motivating papers

Azizi et al. [20], provides a detailed, empirical insight into the trade-off between complexity, power and performance in processor design. The study demonstrates that a series of sweet-spots exist at varying performance levels, in which different micro-architectural designs are most cost-effective from the perspective of power.

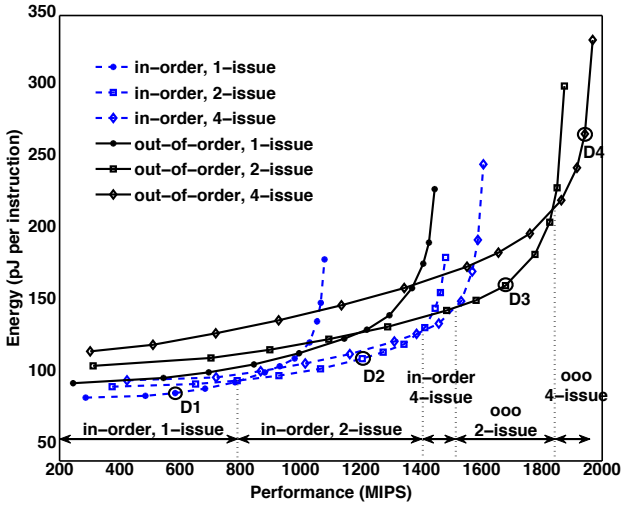


Figure 3.1: Energy-performance trade-off for a range of six micro-architectures. Reproduced from Azizi et al. [20].

The authors argue that the additional cost of pushing a simple five-stage pipeline to higher and higher clock speeds outweighs the complexity cost of moving to a dual-issue in-order design.

This project aims to push beyond the coarse-grained approach taken by Azizi et al. [20], considering how small, but effective optimisations can be used to extract additional performance from each design without a significant increase in complexity. However, complexity, rather than power consumption is the key consideration of this project.

Celio et al. [6] considers the potential impact of macro-op fusion in RISC-V and takes a subtly different approach to complexity, proposing the use of macro-op to avoid ISA 'bloat'. A key result of this project demonstrates that only a small amount of additional hardware is necessary to add macro-op fusion an existing RISC-V core.

The key results of Celio et al. [6] include a detailed analysis of the SPEC integer benchmark set, compiled for RISC-V, which shows that without additional compiler optimisations, on average $\sim$2.30 % performance can be gained in simulation, increasing to $\sim$5 % with perfect optimisations. Note that the impact of macro-fusion on real hardware may differ—caching, branch prediction, and other

existing optimisations may interact with any additions in an in-order pipeline, such as Rocket. The increased decode bandwidth coupled with reduced functional unit utilisation in an out-of-order superscalar pipeline may lead to unexpected benefits—the real-world performance can only be determined by running the benchmarks on real or simulated hardware. The implementation of macro-op fusion in Rocket allows the effect of real hardware to be observed, albeit with a reduced set of benchmarks due to the need to run in simulation.

3.2 Macro-op fusion

Academic literature relating to macro-op fusion is relatively limited despite its widespread use in industry. Intel has been using macro-op fusion in all of its high-performance cores since the Core 2 micro-architecture was released in 2006, making up for the lack of compare-and-branch instructions by fusing compare-and-branch instruction pairs. In combination with the super-scalar architecture, macro-op fusion allows Intel's Core architecture to execute up to six instructions in parallel on a single core[1]. Macro-op fusion has also been used in SiFive's 7 series of CPU cores to optimise short forward branches (see §3.2.1), and in ARM's CPU cores to optimise literal generation from immediates[2].

Intel's application of macro-fusion has seemingly been limited to the optimisation of compare-and-branch instruction pairs in their high-performance out-of-order super-scalar cores. Due to the frequency of the conditional branch idiom, a significant reduction in load on hardware resources is possible, leading to an effective 10% reduction in instruction count in the SPEC integer benchmark [11]. The relatively small reduction in the floating point benchmark (1.76%) demonstrates how sensitive such a specific optimisation can be to the tested workload. Despite illustrating the potential macro-op fusion has to improve performance, these results give little indication of performance benefits available to RISC-V, which has support for conditional branching.

[1] 128-bit multiply, add, load, store and macro-fused compare and branch

[2] `https://github.com/llvm-mirror/llvm/blob/master/lib/Target/ARM/ARMMacroFusion.cpp`

3.2.1 Macro-op fusion in RISC-V

In the context of RISC-V, macro-op fusion has been suggested as a technique to allow the base ISA to remain true to the principles of RISC, while allowing CPU designers to make their own decisions on complexity to improve performance as necessary.

Celio et al. [6] argue in favour of using macro-op fusion in RISC-V to prevent what they term 'ISA bloat', where increasingly 'CISCy' instructions are added to the base ISA for performance reasons, but potentially at the expense of small, simple CPU implementations, which are required to support these more complex instructions. Celio et al. [6] refer to the example of the ARM architecture, which despite its beginnings as a purely RISC architecture, has gradually increased in bulk over time, using the example of the `LDMIAEQ` instruction [21]. The `LDMIAEQ` instruction, read as 'load multiple, increment address if equal', can be used as an idiom `LDMIAEQ sp!, {R4-R7, PC}` 'for pop stack and return from function' and results in six loads from memory and seven register writes. The paper's key argument is that a similar level of performance to more complex ISA's is achievable in RISC-V, without complicating the base ISA.

To determine the most effective approach to macro-op fusion in RISC-V, Celio et al. [6] run the SPEC integer benchmark in simulation, alongside static analysis of program assembly, to ascertain the most frequent dependent instruction pairs that could be fused. The resulting data demonstrate that the most frequent idiom is accessing arrays, which results in a dependent triplet of instructions, known as an 'indexed load':

```
// a5 = ((int64_t*)s9)[a5]
slli a5, a5, 0x2
add a5, s9, a5
ld a5,0(a5)
```

Celio et al. [6] further argue that an indexed load instruction could be easily added to a RISC-V CPU as an addition to the RISC-V integer instruction-set. However, they explain that by using RVC, each fusible pair could be represented as two two-

byte compressed instructions, the same length as a new four-byte instruction. By "lying to your decoder" [21], micro-architects can simply pretend that ISA support exists, and reduce the effective instruction count by one every time a fusible pair is encountered.

As part of their analysis, Celio et al. [6] also compare the RV64GC ISA to a number of other common architectures, including x86-64 and ARMv7/8, to demonstrate that for the benchmarks used, RV64GC uses ∼8 % fewer dynamic instruction bytes than x86-64, and with the use of macro-op fusion, ∼4.20 % fewer micro-ops than x86-64. The comparison between RV64GC and ARMv8 suggests a very similar number of micro-ops, which RISC-V achieves with a 'leaner' ISA.

To demonstrate the effectiveness of macro-op fusion in RISC-V, Celio et al. [6] again analyse the SPEC integer benchmark set, by calculating the effective reduction in instruction count due to macro-op fusion. They then perform manual analysis to determine the remaining potential speed-up available for an optimising compiler to extract. Focussing on three fusion opportunities: indexed load, load effective address and, clear upper word, they determine that a 2.25 % reduction in effective instruction count is available without compiler optimisations. This reduction increases to 5 % with sufficiently intelligent compiler optimisations.

As suggested by Celio et al. [6], the logical progression beyond these results is to implement macro-op fusion the Rocket core, alongside a compiler optimisation pass to take advantage of the remaining 2.75 % of macro-op fusion opportunities in SPECint.

Macro-op fusion is also used in SiFive's 7 Series processor to transform single-instruction forward conditional branches into a single conditionally executed instruction[3]. The benefit obtained is that hard to predict data-dependent brnaches can be avoided, leading to a 10 % improvement in the performance of the CoreMark benchmark[4].

[3] `https://github.com/gcc-mirror/gcc/blob/master/gcc/config/riscv/riscv.h\#L665`
[4] `https://www.mail-archive.com/gcc-patches\@gcc.gnu.org/msg211045.html`

3.3 Dynamic optimisations

To push beyond the relatively simple optimisation of macro-op fusion, enabled by statically-known registers and immediate values assigned by the compiler, this project additionally considers the possibility of dynamic opportunities, in which effective instruction count may be reduced further, with knowledge of dynamic register values.

Dynamic optimisation hardware may allow similar optimisations to those enabled by 'abstract interpretation' in optimising compilers [22], allowing instruction pairs to be fused at runtime if register values are known to fulfil certain requirements.

Since this project ultimately demonstrates that the proportion of potential fusible instructions is far too small, a secondary approach is considered, based on the approach by Yi and Lilja [1]. The authors suggest detecting 'trivial computations' by applying mathematical identities to instructions, and either bypassing or simplifying operations where possible. By applying these rules in simulation, they demonstrate a potential speed-up of 8 % in a variety of integer benchmarks. However, as will be illustrated in §4.2, the architectural assumptions made in the paper do not map well to RISC-V, especially with an in-order pipeline, such as Rocket.

Nevertheless, the results of Yi and Lilja [1] are reproduced in §4.2, motivating the progression made in the third stage of this project to more complex optimisations, namely dual-issue.

3.4 Dual issue

The final stage of investigation in this project involves the use of dual-issue as a complexity-effective optimisation to the Rocket pipeline. This investigation will involve exploring a number of approaches to partial dual-issue, aiming to find the sweet-spot in the trade-off between performance and complexity. Furthermore, to maintain low-complexity hardware, this investigation will focus entirely on in-

order dual-issue cores, as motivated by Azizi et al. [20], which demonstrates that the energy-performance sweet spot for low to mid-range cores is in-order dual issue.

As explained in chapter 2, dual issue is a processor optimisation that can be described as the most restrictive form of super-scalar micro-architecture. Dual issue allows two instructions to be executed simultaneously provided (i) these instructions are independent, (ii) that sufficient hardware exists to produce the necessary arguments for each instruction (from the register file), and (iii) that sufficient hardware exists to execute both instructions.

Assuming sufficient throughput can be maintained by the front-end of the pipeline, dual-issue leads to a straight-forward increase in performance, analogous to the effective reduction in instruction count resulting from macro-op fusion. Maintaining sufficient throughput relies on an accurate branch predictor (or small branch misprediction penalty), large-enough instruction cache and instruction buffer to continuously supply up to two instructions at a time. This requirement is particularly noticeable in the Cortex-A55, described later in this section.

The range of dual-issue processors in industry includes many relatively similar micro-architectures, including some of ARM's application-class 'Cortex-A' processors, and the high-performance micro-controller-class ARM Cortex-M7. Notably, partial dual-issue is used in the Cortex-A7, as described in §3.4.1. Intel's 'Bonnell' and 'Xeon Phi' micro-architectures also make use of dual-issue.

More recently, SiFive's 7 series of RISC-V cores use dual-issue to compete directly with ARM's A55 and M7 CPUs, and provide a zero-cycle load to use latency with the addition of a 'late ALU'.

The open source SwERV core by Western Digital[5]demonstrates the high performance made available by pushing the limits of dual-issue in-order design. This core achieves 5.0 CoreMark/MHz and 2.9 DMIPs/MHz in a 32-bit design, competitive with ARM's Cortex A15 out-of-order super-scalar core, and out-performs the BOOM out-of-order four-way super-scalar core.

Finally, within academia, Patsidis et al. [16] describes a partial dual-issue RISC-V

[5]`https://www.anandtech.com/show/13964/western-digitals-riscv-swerv-core-released-for-free`

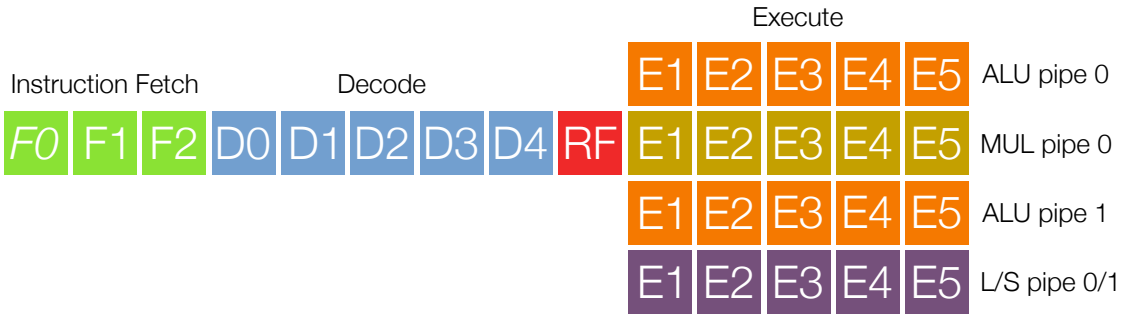


Figure 3.2: ARM Cortex-A8 pipeline diagram

core, making particular use of the RISC-V compressed 'RVC' extension to reduce the hardware impact of implementing dual-issue, and using register-renaming on a subset of the architectural registers to allow pairs of instructions to be dual-issued, even in the presence of false-dependencies.

3.4.1 ARM cores

ARM markets its application-class processing cores, which cover range of a power/performance points, as 'Cortex-A'. Those cores that are in-order demonstrate a small range of possible levels of partial dual-issue.

ARM Cortex-A8: *full dual-issue*

ARM Cortex-A8 (Fig. 3.2) was core the first to implement the ARMv7 ISA, and was ARM's first full dual-issue design (released in 2005). It consisted of a 13-stage pipeline that allows ALU instructions, multiply, and load/stores to be performed in parallel. Therefore, this implementation of dual-issue corresponds to the least-restrictive combination described in Table D.1.

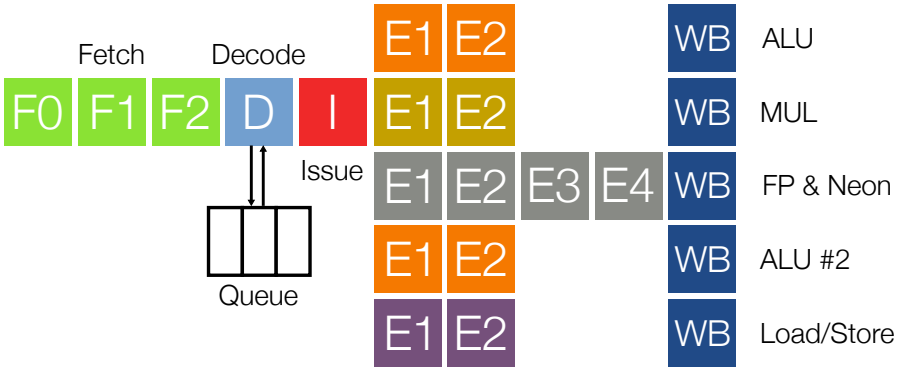


Figure 3.3: ARM Cortex-A7 pipeline diagram

ARM Cortex-A7: *partial dual-issue*

The ARM Cortex-A7 (Fig. 3.3) limits dual issue to pairs of ALU operations, of which at least one must use an immediate as an argument[6]. This requirement suggests that the core is limited to two or three integer read ports, corresponding to the '2× ALU' column, with an additional write port described on Table D.1. The Cortex-A7 counter-intuitively supersedes the Cortex-A8 described previously, as it makes up the expected performance deficit by reducing the pipeline length, which enables a shorter branch mis-prediction penalty[7].

ARM Cortex-A55/A53: *full dual issue*

The ARM Cortex A55 (Fig. 3.4) and A53 cores improve upon the Cortex-A7 providing "symmetric" dual-issue, which allows instructions to dual-issue to any two functional units. The key improvements from the A55 to the A53 are to the memory subsystem, enabling the A53 to sustain throughput when dual-issuing a large number of instructions sequentially.

[6] https://github.com/gcc-mirror/gcc/blob/master/gcc/config/arm/cortex-a7.md
[7] https://www.anandtech.com/show/4991/arms-cortex-a7-bringing-cheaper-dualcore-more-power-efficien

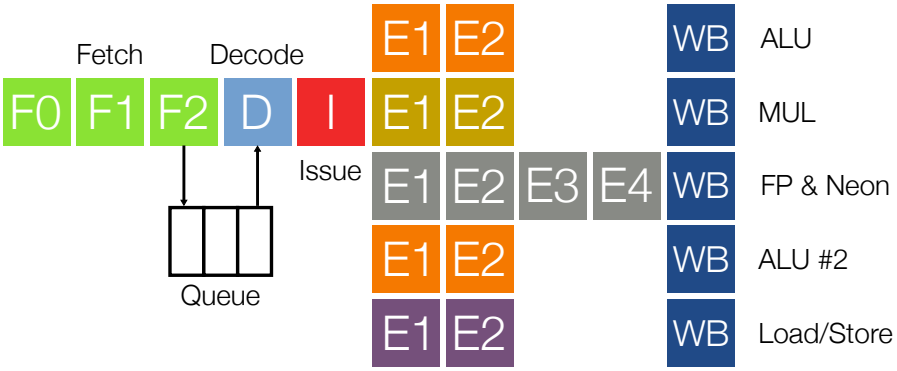


Figure 3.4: ARM Cortex-A55 pipeline diagram

Figure 3.5: ARM Cortex-M7 pipeline diagram

ARM Cortex-M7: *full dual issue*

Unusually for a micro-controller class core, the ARM Cortex-M7 (Fig. 3.5) features full in-order dual-issue, in an effort to extract as much performance as possible, within the restrictive power envelope of an M-class core.

3.4.2 RISC-V cores

Dual-issue designs targeting the RISC-V ISA originate from both industry and academia. This section discuss three key examples: SiFive's 7 series, implementing a limited form of dual issue; Western Digital's SwERV core, which implements full in-order dual-issue and the use of the RISC-V compressed extension to provide resource-efficient dual-issue, as described by Patsidis et al. [16].

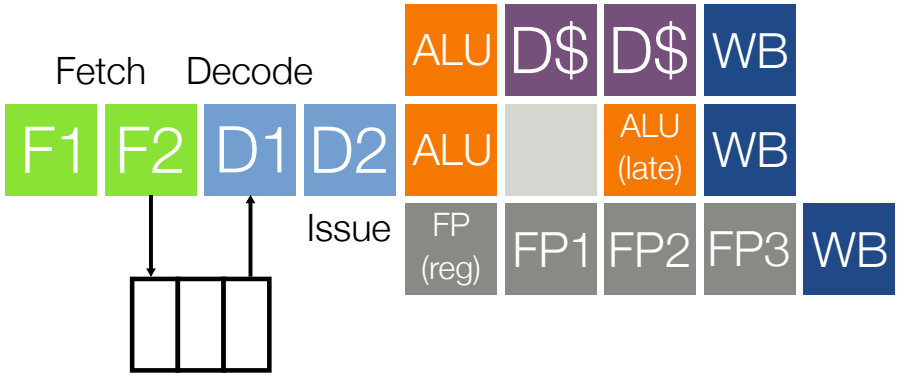


Figure 3.6: SiFive 7 Series Pipeline Diagram

SiFive 7 series

The SiFive 7 series (Fig. 3.6) is particularly interesting, due to its combination of dual-issue in an eight-stage in-order pipeline with macro-op fusion, which enables conditional execution of short forward branches, and an additional 'late' ALU to enable a zero-cycle load-to-use latency.

The dual-issue implementation in these cores consists of two pipelines, the 'address' pipeline (A) and the 'branch' pipeline (B). The A pipeline handles loads, stores, and floating-point to integer register file moves; while the B pipeline handles branches, integer multiplication, integer division, and floating point operations. Both pipelines are able to perform integer ALU operations, allowing two independent integer instructions to be executed simultaneously [4]. This behaviour is similar to the ARM Cortex-M7 and Cortex-A53/55 cores, but and SiFive offers three differently-configured variants of the 7 series.

SwERV

The open-source SwERV core (Fig. 3.7) by Western Digital [23] is a high-performance in-order, dual-issue core that is competitive with significantly more complex cores. The core makes use of three integer read and three integer write ports[8], placing it at the very top of dual-issue configurations considered by the dual-issue experiments in §4.3.3. The core consists of two integer pipelines, a load/store, and

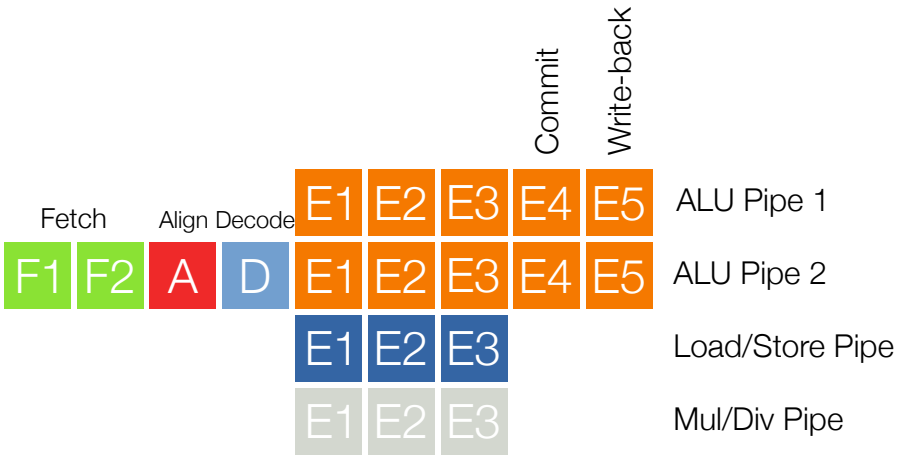


Figure 3.7: SwERV Pipeline Diagram

multiply pipeline.

3.4.3 Intel

In addition to the ARM and RISC-V cores mentioned previously, dual-issue has also been implemented by Intel in the 'Bonnell' and 'Xeon Phi' micro-architectures. Intel's 'Bonnell' micro-architecture was used in the 'Intel Atom' line of CPUs, aiming to maximise performance per watt by using in-order, dual issue alongside multi-threading. In contrast, Xeon Phi used a large number of simple, dual-issue in-order cores to provide a highly-parallel computing platform in an attempt to keep up with future demands for computing power [24].

3.5 Summary

This chapter discussed the key motivating factors for the three main areas of research undertaken in this project: macro-op fusion, dynamic optimisation opportunities and variations of dual-issue. In particular, the experiments described in the following chapters are heavily inspired by two key papers: (i) Celio et al. [6],

[8] `https://github.com/westerndigitalcorporation/swerv_eh1/blob/master/design/dec/dec_gpr_ctl.sv\#L26`

which explains the rationale for adding macro-op fusion to RISC-V and demonstrates the theoretical performance improvements; and (ii) Azizi et al. [20], which demonstrates, in fairly coarse terms, how at each potential power or performance target, there exists an optimal micro-architecture. In broad terms, this result means that as required processor performance increases, the transistor-level design must become increasingly extreme and power-hungry to achieve this performance with simple processor designs. As part of their analysis, Azizi et al. [20] demonstrate that a dual-issue in-order core is optimal for the low- to mid-range CPU's between the simplest in-order single-issue cores and complex out-of-order super-scalar cores.

To build upon the work of Azizi et al. [20], the remainder of this chapter discussed a number of dual-issue designs from ARM, using the RISC-V architecture, and Intel, discussing the trade-offs made in these cores to the extent possible given the relatively limited information available about each core.

Chapter 4

Design and Implementation

This chapter describes the experiments performed, and development undertaken to investigate the effectiveness of a series of potential processor optimisations. These optimisations seek to improve the performance of scalar RISC-V pipelines, while minimising the increase in processor complexity. Three approaches will be taken to improve performance, (i) 'static' macro-op fusion, which relies on the registers used in pairs of dependent instructions; (ii) 'dynamic fusion' opportunities, which rely on the runtime value of registers and (iii), a broad range of dual-issue approaches, which allow for a fine-grained trade-off between performance and core complexity.

This chapter describes work to reproduce the original results of Celio et al. [6] with more recent and more mature RISC-V compiler ports, and then extends this result three times: firstly, by producing an automated limit study to demonstrate the potential speed-up in a system that could fuse non-adjacent instructions; secondly by adding an optimisation pass to LLVM in an attempt to increase the number of potential fusion opportunities; and finally by implementing macro-op fusion in hardware on top of the Rocket core.

Dynamic opportunities represent the range of processor optimisations that rely on the value of each register at runtime. The goal of this investigation was primarily to determine the potential benefits, if any, of knowing the runtime values of instruction arguments, and secondly how this knowledge might produce addi-

tional fusion opportunities. A further investigation reconsidered the conclusions of Yi and Lilja [1], in the context of RISC-V and Rocket. The authors claimed an average speedup of 8% in simulated workloads. Simulated experiments with RISC-V demonstrated that even in the best possible case, significantly fewer fusion opportunities were available.

Finally, to push the boundaries of complexity-effective processor optimisations, a broad investigation into the use of (partial) dual-issue was performed. Through the use of simulated experiments in Spike, a large number of potential configurations were investigated, giving future developers a chance to make a fine-grained trade off between performance and complexity.

4.1 Macro-op fusion

The approach of macro-op fusion, in which we rely on the registers allocated at compile-time to fuse a pair of simple instructions representing a common idiom to be transformed into a single processor micro-op, saves a cycle of computing time per fused instruction pair. To motivate further investigation, an initial experiment was performed to demonstrate the range of potential fusion pairs, followed by reproducing Celio et al. [6] results, in which they demonstrated a potential average speedup of 5% across the SpecINT set of benchmarks.

An automated limit study is presented, which demonstrates the potential speedup available by combining an additional compiler optimisation pass with macro-fusion hardware. Finally, concrete implementations of both a macro-fusion optimisation pass for LLVM and macro-fusion hardware for the Rocket core are described.

4.1.1 Feasibility of macro-op fusion

To quickly demonstrate the potential effectiveness of macro-op fusion in RISC-V and provide targets for further development, a series of initial experiments were performed. These included: a broad analysis of instruction pairs in SpecINT, a thorough reproduction of Celio et al. [6], and an automated limit study that

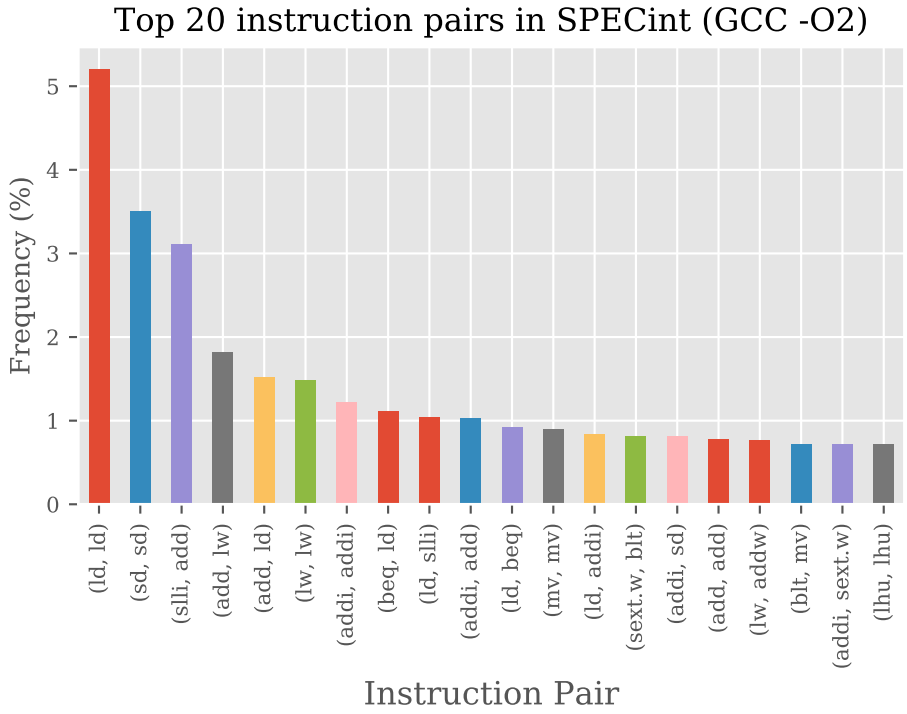


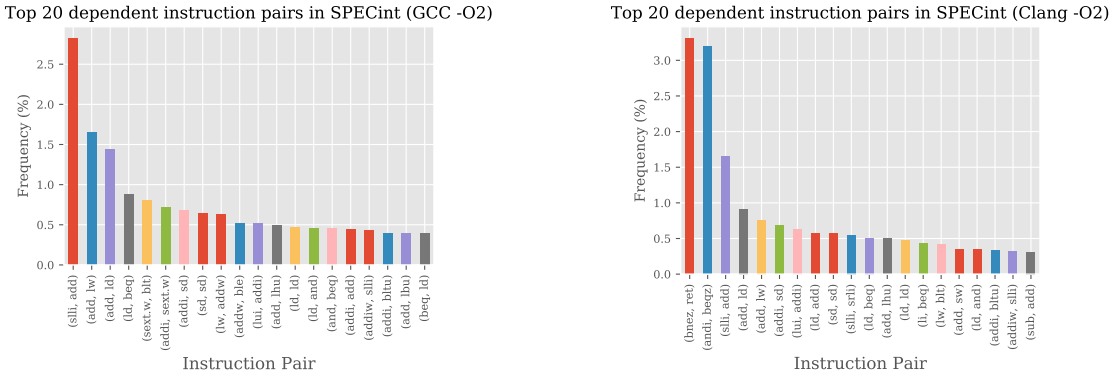
Figure 4.1: Top twenty most frequently executed instruction pairs in SPECint.

demonstrates the effects of perfect compiler optimisations or a more general CPU implementation able to fuse across non-dependent instructions.

Dynamic instruction pairs

The most common dynamic instruction pairs were determined using the RISC-V ISA simulator 'Spike' to produce a histogram of most common instructions, and combined with static analysis of each benchmark program to produce a graph of instruction pairs vs frequency.

This procedure was then extended to include only dependent instruction pairs, where the result of the first instruction is used as an argument to the second instruction and then overwritten. These results are shown in Fig. 4.2, which compares GCC and Clang. However, the code generation is sufficiently similar that the top few fusible idioms (i.e., simple dependent pairs that do not require branching



(a) GCC

(b) LLVM

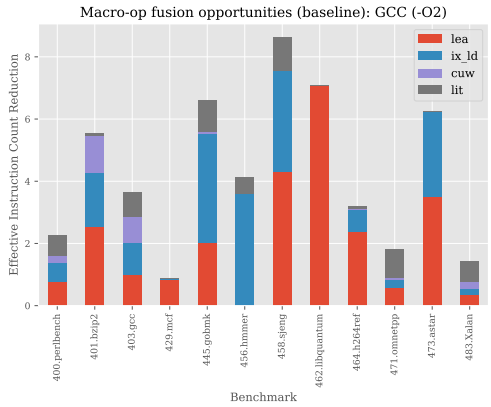
Figure 4.2: Top twenty most frequent dependent instruction pairs in SPECint, when compiled with GCC and LLVM.

or additional register read ports) are approximately the same.

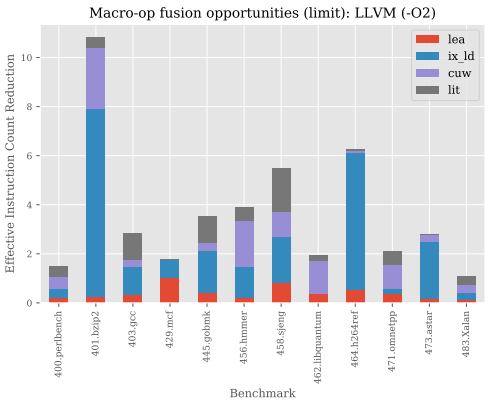
Finally, the four most common fusible idioms were determined: add/ld, shift/add, slli/srli and lui/addi. This result is in contrast to Celio et al. [6], in which only the first three idioms were considered.

Fusion without compiler optimisations

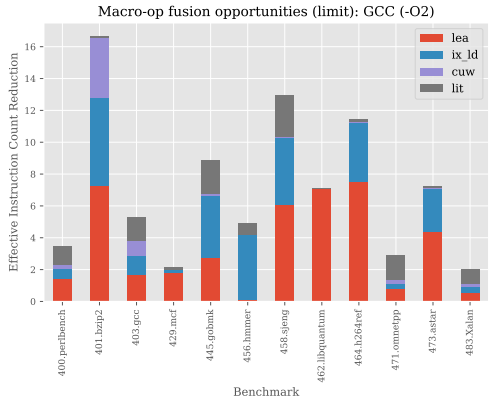
To reproduce the results of Celio et al. [6], a similar approach was taken to the investigation of dynamic instruction pairs, but instead looking at a specific set of instruction pairs that are fusible at run-time without additional compiler optimisations. This requires the instructions to be adjacent, dependent, and overwrite the intermediate result produced (guaranteeing that no later instructions rely on the intermediate value). These results were produced using a limited version of the limit study algorithm described in Listing 1, and are shown in Fig. 4.3a and Fig. 4.3b for GCC and LLVM, respectively.



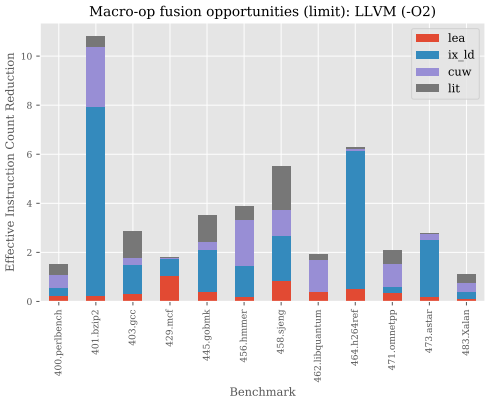
(a) GCC (baseline)



(b) LLVM (baseline)



(c) GCC (limit)



(d) LLVM (limit)

Figure 4.3: Composition of fusion opportunities for GCC and LLVM, including baseline values, and theoretical limits.

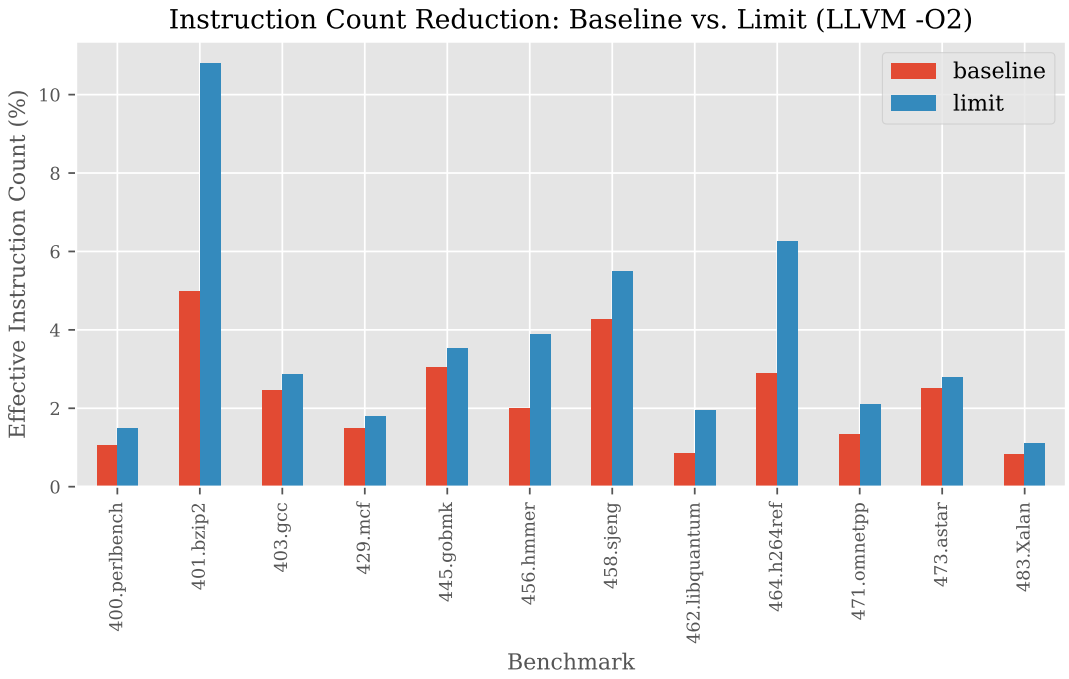


Figure 4.4: Effective instruction count reduction with LLVM: baseline vs. limit

Limit study

The previous results demonstrate the effective instruction count reduction available with no compiler changes. However, in practice, a compiler can re-order these instructions at compile time to provide a greater number of potential fusion opportunities. To determine how significant this improvement could be, an automated limit study was developed. The algorithm used is described in Listing 1.

The analysis performed is similar to the analysis stage of an optimising compiler, working its way through basic blocks to find pairs of fusion candidates.

Conclusion

The results obtained in the three experiments described previously demonstrate the potential effectiveness of macro-op fusion, either with or without an added compiler optimisation pass. In particular, §4.1.1 motivates a processor implementation, and

```
def check_block_lea(block):
  """
  Find limiting number of load effective-address fusion opportunities:

  Look for pairs like:

  slli rd, rs1, 1
  add rd, rd, rs2

  As long as this is not separated by a dependent instruction, this is fine.
  (i.e. nothing reads from or writes to rd).
  """
  watching = {}
  limit = []
  distances = []

  for i, insn in enumerate(block):
    if insn.is_slli():
      watchinig[insn.rd] = i
    elif insn.is_add():
      if insn.rd == insn.rs1 and insn.rd in watching:
        distances.append(i - watching[insn.rd])
        limit.append(insn.pc)

        del watching[insn.rd]
    else:
      # this includes the destination register
      for arg in insn.args:
        if arg in watching:
          del watching[arg]

  return limit, distances
```

Listing 1: Code describing the algorithm used to detect the limiting number of macro-op fusion opportunities, given a block of RISC-V machine code. Example given for detecting load effective address, with equivalent implementations used for clear upper word, indexed load and literal generation.

§4.1.1 motivates a compiler optimisation pass.

4.1.2 Hardware implementation

The development of a macro-op fusion hardware implementation in Rocket aimed to keep in mind both Rocket's goal of being highly configurable, and this project's goal of developing complexity-effective optimisations. The following requirements summarise the goals of this hardware implementation:

- Configurable implementation of macro-op fusion: enable/disable each fusion pair, and enable/disable macro fusion as a whole.
- Small impact on core area and clock speed.
- Focus on three fusion opportunities, as indicated by prior experiments: clear upper word (`slli`/`srli`), load effective address (`slli`/`add`), and indexed load (`add`/`[ld,lw,lh,lb,lwu,lhu,lbu]`).
- Allow any combination of compressed and uncompressed pairs to be fused, regardless of alignment.

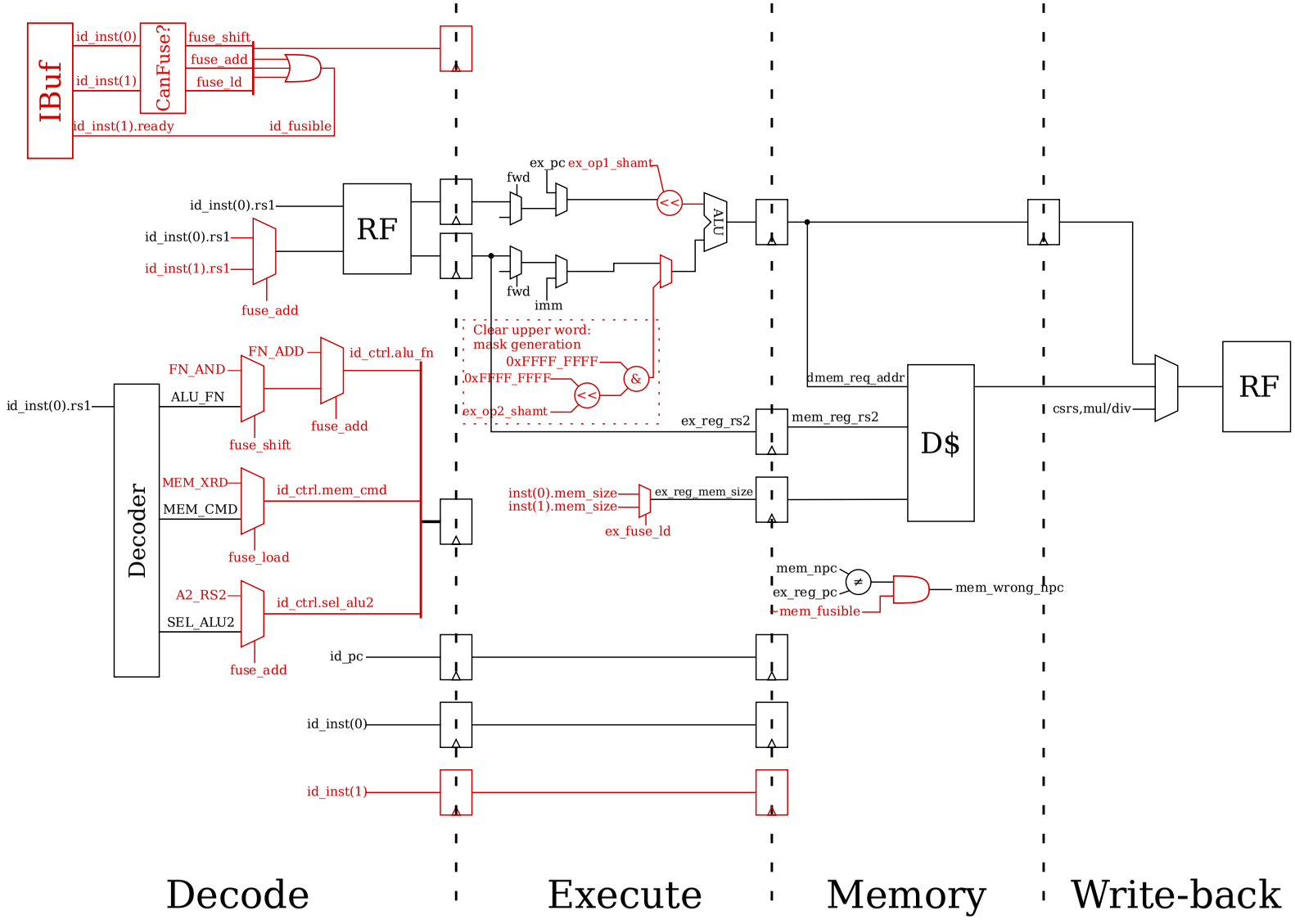


Figure 4.5: Rocket pipeline, modified for macro-op fusion (changes highlighted in red). Simplified by excluding floating point unit and integer multiplier/divider, which are not affected by macro-op fusion in this implementation. Created with reference to the Rocket source code, and similar diagram by LowRISC [25]

Modifications

The necessary changes to Rocket, referencing the relevant stages of Rocket's five-stage pipeline:

1. Fetch stage:
   (a) Re-configure Rocket's instruction cache and instruction buffer to fetch two instructions at a time, and optionally decode two at a time.
2. Decode stage:
   (a) Change control signals for each fusion opportunity.
   (b) Generate shift amount for clear upper word mask. *(If mask generation in the execute stage affects critical path, mask could be generated in the decode stage).*
   (c) Mux source register two for indexed loads.
3. Execute stage:
   (a) Pre-shifter on ALU operand one for load effective address.
   (b) Mask generator on ALU for clear upper word.
   (c) Get correct memory size for indexed load.
4. Memory access stage:
   (a) Fix next program counter to avoid branch prediction bugs.

Fetch stage

Rocket's instruction cache is already capable of fetching up to eight bytes at a time (enough to sustain every single pair of instructions being fused continuously). The instruction cache provides a stream of data to the instruction buffer, which provides a ready/valid interface to the decode stage, allowing two instructions to be selectively decoded at a time when fusion opportunities appear.

Due to the need to handle misaligned instructions, caused by support for RISC-V compressed instructions, little additional hardware cost is caused by fetching two instructions at a time.

Although no modifications were made to Rocket's fetch stage during development, scope remains for changes in the future to allow fusion opportunities to be partially detected earlier in the pipeline, if necessary for performance reasons.

Decode stage

The existing implementation of the decode stage in Rocket is implemented as a large decode table, providing a mapping between instruction bit patterns and the corresponding control signals. These control signals specify whether a memory access should be performed, where the ALU should get its arguments from, and where its result should be written back to. While fusion detection could have been added directly to the decode table, due to the relatively small number of relevant instructions, a small amount of fusion-specific decode hardware was added instead, keeping a clear separation between new and existing code.

The added macro-fusion logic detects potential fusion pairs, and modifies the produced control signals to produce behaviour that is equivalent to the combined instruction pair. A number of new control signals were added to the CPU core, allowing masks to be generated in the execute stage and for shifts to be performed on the ALU's input.

As previously mentioned, depending on future requirements, decode-stage logic used to detect fusion opportunities could be moved to the fetch stage as necessary.

Execute stage

The majority of modifications to the execute stage involved ALU modifications, introducing a pre-shifter for operand one and the option to set op2 to a mask necessary for clear upper word fusion.

Additionally, modifications were made to the branch prediction hardware, preventing instructions from being killed due to an unexpected program-counter value causing the pipeline to be repeatedly flushed.

ALU modifications

Changes to the ALU included the addition of an optional shift operation to the first input operand, controlled by `op1_shamt` and generated in the decode stage. Additionally, masks were generated that can be selectively used as the second input operand to implement clear upper word. In both cases, it is possible that the critical path of the CPU core may be negatively affected, causing a potential decrease in maximum clock speed. Possible mitigations for this effect include moving mask generation to the decode stage. However, pre-shifting operand one must be done in the execute stage, as it relies on register values fetched in the decode stage. Ultimately, a possible reduction in clock speed must be traded off against reductions in effective instruction count. Analysis of the clock speed impact of macro-op fusion will be discussed alongside core area in §5.1.4.

Branch predictor modifications

As mentioned in Chapter 2, Rocket, like other high-performance CPU cores, relies on branch prediction to reduce pipeline stalls in the event of indirect or conditional branches. To detect mis-predicted branches, Rocket relies on the execute stage to produce the 'correct' next value of the program counter, after branch targets have been computed and conditions evaluated. However, this method conflicts with the need to skip program counter values when macro-fusion is enabled. Therefore, the implementation of macro fusion in Rocket multiplexes in the correct program counter value when fusion opportunities are detected, as illustrated in Fig. 4.5. Since the fused instructions do not include any branches, these modifications are essentially trivial and have been tested using Rocket's assembly tests.

Memory-access stage

To correctly handle add/load fusion opportunities loading different width values from memory (i.e., ld,lw,lh,lb,lwu,lbu,lhu), a single trivial modification is made to

the memory access stage. This modification ensures that the width of the memory operation performed accurately reflects the load instruction fused.

Write-back stage

As required by the RISC-V specification, Rocket implements precise exception handling. As described previously in §2.2.4, this feature ensures that the architectural state during an exception handler is exactly as it was after the previous instruction executed, in effect allowing the faulting instruction to be emulated in software by a trap handler.

In the case of the three instruction fusion pairs implemented in Rocket, the need to handle exceptions precisely only causes a significant issue in the case of 'indexed load', where the load instruction may fault due to a misaligned load. Since misaligned loads are required by the Linux ABI and not supported in hardware by Rocket, misaligned loads must be emulated in hardware. Therefore, it is important, that in the event of a misaligned load, the architectural state during the trap handler reflects the expected state after the first instruction.

Conveniently, in the case of Rocket, the solution is to write-back the intermediate result to the register file in the event of a misaligned load exception. Later implementations of macro-op fusion with RISC-V in a more complex pipeline may make precise exception handling more difficult to achieve.

Testing

In addition to Rocket's built in ISA tests and benchmarks, additional tests were added to ensure that misaligned loads were correctly handled in fusible add/load pairs.

4.1.3 Compiler optimisation implementation

An optimisation pass for macro-op fusion was added to LLVM using the built in `MachineScheduler` and `createMacroFusionDAGMutation` function. This function takes a new function as an argument, called `shouldScheduleAdjacent`, which tests whether a pair of instructions are fusible as part of a similar algorithm to that described in §4.1.1. Internally, LLVM iterates over data-dependent pairs of instructions, calling `shouldScheduleAdjacent` on each pair, and modifying a directed-acyclic graph (DAG) of instructions as it goes along. When a fusible pair is encountered, the pair is noted in the DAG, along with the reduced latency available by fusing this pair of instructions. LLVM's `MachineScheduler` can then schedule instructions optimally.

The resulting modification is used in the backend of the Clang C compiler, allowing benchmarks to be compiled with or without the optimisation enabled, and is Cvaluated in chapter 5.

4.2 Dynamic approaches

Extending the "static" macro-op fusion described previously, perhaps a natural question to ask is whether additional opportunities to reduce effective instruction count can be found if we know the runtime value of registers used by each instruction? To investigate this possibility, two experiments are described in this section: the first experiment simply aims to determine the proportion of instructions whose values are likely to provide opportunities for run-time optimisation.

The runtime values measured are just zero and one, and are chosen simply because they are likely to nullify the effects of many integer operations such as `add, sub, mul, div, and, or, xor`. The percentage of zeros and ones in the SPECint benchmark set are shown in Fig. 4.6, with an average of 3.2% of register values equal to zero and 1.7% equal to one. This already severely limits the potential range of optimisations, however if implemented sufficiently simply in hardware, this could be worthwhile from the perspective of the Kiss principle [2].

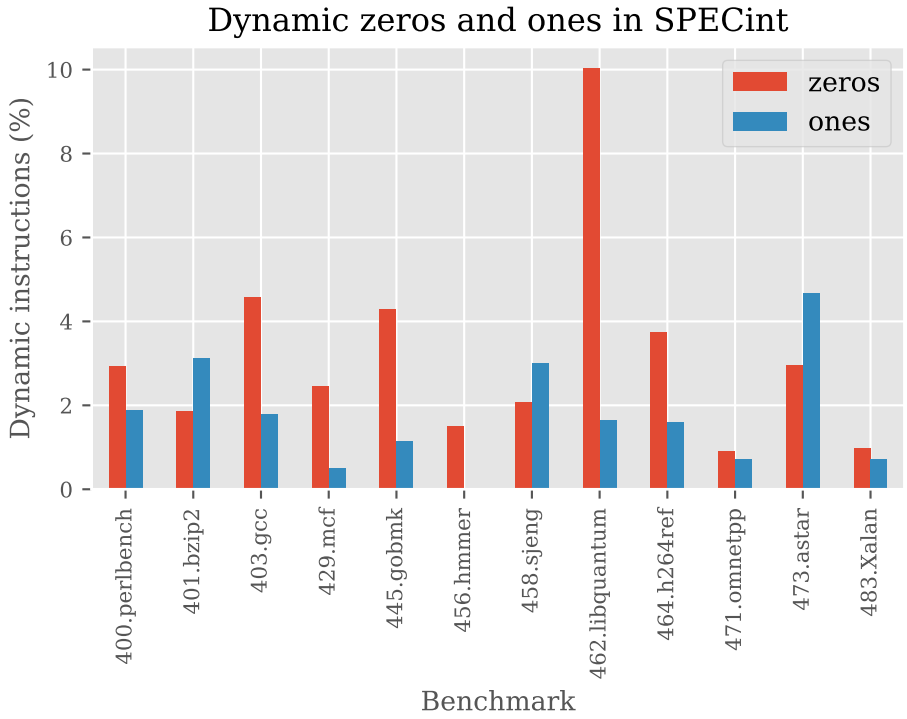


Figure 4.6: Instructions with arguments that are either zero or one in SPECint, compiled with GCC -O2.

The second experiment took a more direct approach, looking to reproduce the results of Yi and Lilja [1], in which "trivial computations" are bypassed or simplified. This includes integer addition by a runtime value of zero, subtraction by zero, multiplication by one, and equivalent identities for bitwise operations. Crucially, to be 'trivially bypassable', this instruction must overwrite the source register, a move instruction must be executed, effectively nullifying any theoretical improvement in the Rocket core. The cumulative number of opportunities are shown in Fig. 4.7. The average number of trivially bypassable instructions in SPECint was 0.57 %.

Ultimately, both experiments determined that such optimisations were not worthwhile with a Rocket-like CPU and the SpecINT set of benchmarks. Nevertheless, these studies are described to demonstrate an exploration of the potential design space that lead to the consideration of partial dual issue.

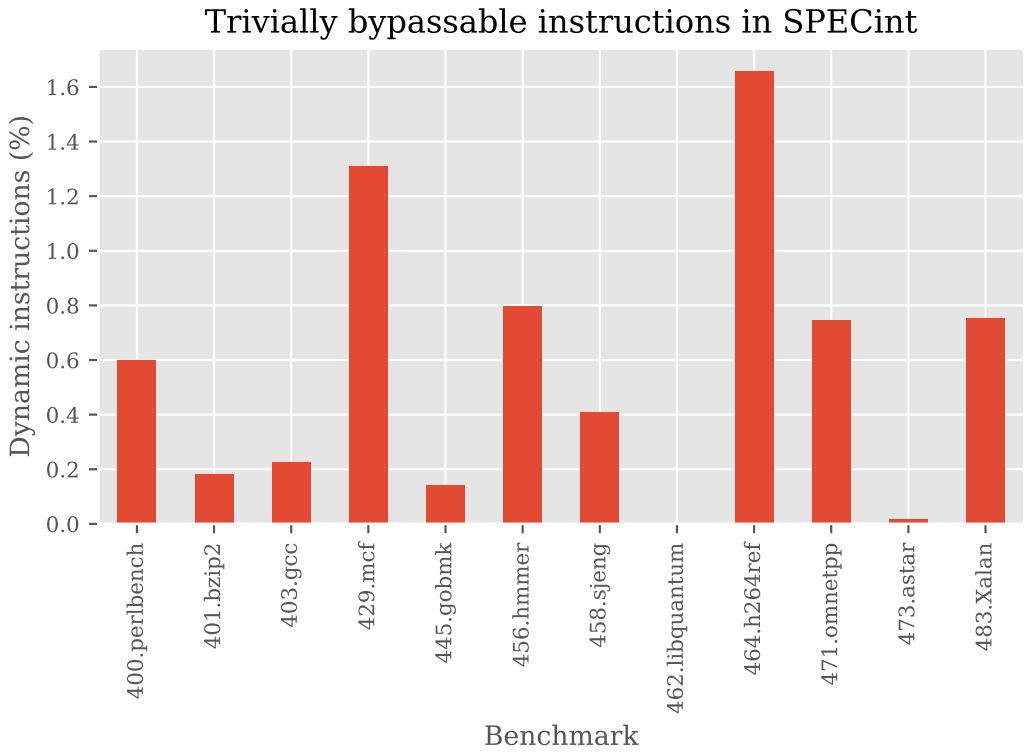


Figure 4.7: Trivially bypassable instructions in SPECint, compiled with GCC -O2.

4.2.1 Hybrid approach

A compromise approach to dynamic fusion combines the approaches of Yi and Lilja [1] and Celio et al. [6] and enables the use of arithmetic simplification by pairing trivial computations with dependent instructions.

```
add x1, x1, x2 // x2 == 0 at runtime
add x1, x1, x3

// effectively becomes

add x1, x1, x3
```

However, there are no examples of this pattern in the top twenty dependent instructions shown in Fig. 4.2 that are not already optimisable by macro-op fusion, and hence this hybrid approach is not considered in any greater detail.

4.3 Dual issue

To push the limits of complexity-effective hardware optimisations, the final stage of investigations focusses on the addition of variations of limited dual-issue to the classic RISC five-stage pipeline. The goal of these experiments is to determine the available additional performance by occasionally executing two independent instructions simultaneously, where available hardware permits. The amount of additional hardware introduced allows a future micro-architect to make an informed trade-off between the simplest hardware implementation (Rocket with the bare minimum hardware added for dual-issue), all the way up to a full dual-issue core with two ALU's and extra ports on the register file.

Although these experiments should be applicable to a wide range of five-stage RISC pipelines, they are primarily motivated by the design of Rocket and pre-existing parallelism, as shown in Fig. 2.4.

An initial examination of the Rocket core shows that the only pre-existing op-

portunity for parallelism is between the floating-point unit and the integer ALU, which each have separate ALU's and entirely separate register files. An additional opportunity exists for parallelism between the integer ALU and the integer multiplication/division unit, which share a register file and would, therefore, require an additional or banked register file.

There are two major considerations when implementing dual issue:

1. What opportunities exist for parallel execution? (e.g., independent functional units)

2. What features allow a core to read enough input arguments, and write back enough results?

4.3.1 Parallel execution

Rocket's modular design, combined with the increased latency of multiplication/division and floating point operations has lead to a natural separation of hardware between the integer ALU, FPU, and multiplication and division unit. These functional units present natural opportunities for parallelism, especially when integer operations are interspersed with floating point operations. However, for the common case (especially outside of highly numerical workloads), little opportunity is available to dual-issue independent integer instructions (excluding multiplications and divisions).

Similarly, in numerical workloads, where many FP instructions are executed sequentially, limited parallelism is available due to the single FPU. Therefore, there exist a number of natural stages between the baseline Rocket core and a full superscalar core that have a large number of parallel functional units. In the experiments described in §4.3.3, a simple progression from (i) a single ALU, to (ii) an ALU plus branch hardare, to (iii) a second ALU is demonstrated.

4.3.2 Parallel data access

As well as the need to compute values independently, sufficient register values are required to keep the ALU's fed with data, along with the capability to write those values back to the register file or forward them to following instructions. Due to the additional hardware cost associated with extra read and write ports, this section considers a number of alternative approaches that allow register file accesses to be reduced, potentially enabling future compiler optimisations.

With no additional modifications to Rocket (two read ports and one write port in the integer register file), the only opportunity for parallel register accesses is between the floating point and the integer register files. This separation also allows for parallel writes without the need for arbitration. Moreover, the following additions represent increasing levels of complexity, and hopefully correspondingly higher performance:

1. *Shared reads*: allow two adjacent instructions to be issued simultaneously, providing the combined number of unique register reads is less than or equal to the number of register file read ports.

2. *Register forwarding*: allows two registers to be read from the register file, as well as additional registers, if available from the last three instructions due to the forwarding network.

3. *Additional read port(s)*: potentially a more significant increase in hardware than the first two suggestions, this modification allows a third (or fourth) register to be read from the register file. However, without an additional write port, dual-issue opportunities to ALU + branch, or ALU + FPU are restricted.

4. *Additional write port(s)*: allows two integer results to be written back to the register file simultaneously, but would likely cause a significant increase in complexity due to the additional increase in forwarding logic, on top of the write ports.

5. *Extended forwarding path*: register forwarding typically allows the three most

recent results to be forwarded, before they are written back to the register file. However, by queueing an additional n results after the write-back stage, additional recent register writes can be obtained in the decode stage without the need to read them from the register file.

4.3.3 Dual-issue experiments

The following section describes the experiments performed to demonstrate the potential speedup available by extending Rocket in a number of simple ways to produce variations of dual-issue in-order cores. This simulation relies on a very simple pipeline model, in which either one or two instructions are taken from the instruction queue. The number of simultaneously issued instructions will depend on two factors:

1. Availability of computation resources: if both instructions require the use of a single functional unit, then they will be unable to dual-issue.
2. Availability of source operands: only a finite number of registers can be accessed at a time from the register file. In the event that too many registers are required, only a single instruction is issued. Note that the results of any prior instructions are otherwise assumed to have been forwarded from later stages in the pipeline as necessary.

Similarly to previous experiments, this simulation relied on program-counter histograms generated by Spike, and does not model the impact of memory latency or less predictable features such as instruction caches and branch predictors. Therefore, these results represent an upper limit to the speedup available across the SPEC integer benchmark set. Future work to develop a more complete pipeline model is suggested in §6.2.

Table 4.1: Average percentage of instructions that can be dual-issued in SPECint

Configuration	ALU	ALU + branch	2 x ALU
baseline	0.198602	0.217500	1.009812
share	0.207159	0.507032	1.682082
share + forward	0.216630	3.203618	5.377526
share + forward + read port	0.216635	4.030010	6.954868
share + forward + write port	0.242378	3.225111	19.345924
share + forward + write port + 1 forwarding cycles	0.242378	3.385249	19.791810
share + forward + write port + 2 forwarding cycles	0.242378	3.680508	20.207363
share + forward + write port + 3 forwarding cycles	0.242383	3.797826	20.615938
share + forward + write port + 4 forwarding cycles	0.242383	3.834868	20.840663
share + forward + write port + 5 forwarding cycles	0.242383	3.898161	20.914805
share + forward + write port + 6 forwarding cycles	0.242383	3.928254	21.081151
share + forward + read port + write port	0.242383	4.051505	21.992404
share + forward + 2 read ports + write port	0.242383	4.195988	22.368803

SPEC 2006 integer benchmark, version 1.1, compiled with GCC -O2.

Chapter 5

Evaluation

This chapter evaluates the development made in Chapter 4, in particular describing the methodology, benchmarks used and results obtained. The key areas requiring evaluation are the hardware implementation of macro-op fusion in Rocket and the optimisations added to LLVM to improve the proportion of available macro-op fusion.

5.1 Macro-op fusion in Rocket

To accurately demonstrate that the theoretical speedup highlighted by simulated experiments in Celio et al. [6], and simulated experiments, as reproduced in §4.1.1, translates into a real hardware implementation with a real memory hierarchy, branch prediction and a (some) non-determinism, the Rocket core must be simulated accurately. This simulation requires a realistic set of benchmarks, such that hte results can be compared with those obtained using the Spike ISA simulator. Since the 2016 paper has already demonstrated the case for macro-op fusion in RISC-V, this evaluation stage aims to demonstrate that the speedup promised by Celio et al. [6] translates to a real Rocket core and that this optimisation only results in a small increase in chip area.

5.1.1 Simulation

There are two practical ways to simulate Rocket without taping out physical hardware: using a cycle-accurate software simulator, using Verilator, or to synthesise Rocket for a field-programmable gate array (FPGA). The software simulator is flexible, and quick to set-up, but significantly slower to run benchmarks, simulating just a few thousand cycles per second. Booting Linux or running the full suite of SPECint benchmarks would be prohibitively time consuming. Although running Rocket on an FPGA may solve these problems and allow the processor to be simulated at several tens of megahertz, it may not be able to accurately reproduce the memory hierarchy of a real Rocket core taped out to silicon. Ultimately, due to time constraints and the need to quickly iterate, Rocket was simulated in software during this project, which limited benchmark choices, as described in the following section.

Two different configurations of Rocket were chosen for the benchmarking process: one with macro-fusion disabled and one with macro-fusion enabled. These cores implemented RV64GC with large L1 caches and a branch-target buffer, and therefore offer the opportunity to observe the effect of other optimisations on macro-op fusion.

5.1.2 Benchmarks

Ideally, to be consistent with prior work, and make use of real-world applications for benchmarking, SPECint would have been used. However, it would have been prohibitively time-consuming due to the large number of cycles required. As a result, a smaller group of benchmarks, taken from 'riscv-tests' and 'rv8-bench', were used, which would allow the potential speed-up promised by simulated experiments with Spike to be compared to the exact number of cycles used by Rocket in a real setup.

5.1.3 Performance results

This section describes the Rocket core performance results from three perspectives: (i) comparing the small and large baseline versions of Rocket to Rocket with macro-op fusion enabled, (ii) comparing Rocket with macro-op fusion enabled to macro-op fusion results obtained with Spike and (iii) raw Dhrystone and CoreMark results for baseline Rocket, compared to Rocket with macro-op fusion enabled.

Rocket vs Rocket with macro-op fusion

This experiment aimed to demonstrate a statistically significant difference between the performance of a baseline Rocket system, and a Rocket system with macro-op fusion enabled. This performance was evaluated using the 'riscv-tests' and a simplified version of the 'rv8-bench' benchmark set (the number of iterations of each benchmark was reduced to speed up the simulation process), compiled with a baseline version of GCC.

Since Rocket is non-deterministic, each experiment was repeated eleven times and Student's t-test was used to show a statistically significant difference in performance separately for each benchmark (if applicable).

rv8-bench benchmark results

The 'rv8-bench' benchmark was initially developed to test the performance of the rv8 dynamic binary translating simulator. These benchmarks were modified to execute quickly on the Rocket simulator, allowing a full suite of benchmarks to be run. The results are described in Tbl. 5.2, and demonstrate a small, consistent improvement in performance for all benchmarks. These results are less impressive than may have been expected from a Spike simulation, possibly due to caching and branch prediction behaviour; a more complete set of benchmarks are needed to more accurately demonstrate possible speed-up.

Table 5.1: Comparison of Rocket with and without macro-op fusion enabled using the riscv-tests benchmark set.

Benchmark	Predicted Speedup[††](%)	Cycles		Measured Speedup (%)	p value[‡]
		Baseline	With fusion		
dhrystone	0.24	484 035	470 416	−2.81 [†]	0.00[*]
median	0.08	162 420	162 317	−0.06 [†]	0.65
mm	0.02	762 213	761 985	−0.03 [†]	0.72
mt-matmul	0.06	273 089	273 170	0.03	0.71
mt-vvadd	0.08	742 465	742 566	0.01	0.85
multiply	0.02	169 822	169 884	0.04	0.85
pmp	0.00	1 640 975	1 595 460	−2.77 [†]	0.00[*]
qsort	0.00	723 576	724 181	0.08	0.34
rsort	3.24	862 643	859 013	−0.42 [†]	0.14
spmv	2.94	895 036	888 146	−0.77 [†]	0.04[*]
towers	0.06	120 348	120 436	0.07	0.75
vvadd	0.07	169 574	169 369	−0.12 [†]	0.26

[*] $p < 0.05$, so improvement is statistically significant.
[†] Average improvement in performance.
[‡] Independent Student's t-test comparing raw cycle counts with and without fusion. Raw data reproduced in Appendix A.
[**] All benchmarks compiled with GCC using -O3.
[††] Compiled with GCC, run using Spike, similarly to data collected in §4.1.1

Table 5.2: Performance behaviour of Rocket with macro-op fusion enabled in rv8-bench.

Benchmark*	Cycles		Measured Difference (%)
	Baseline	With fusion	
aes	7 255 368	7 141 584	−1.57 %
dhrystone	5 861 828	5 838 460	−0.41 %
miniz	7 751 000	7 717 188	−0.44 %
norx	6 709 780	6 689 724	−0.30 %
primes	6 553 680	6 407 480	−2.23 %
qsort	5 583 504	5 556 308	−0.49 %
sha512	6 527 544	6 520 292	−0.11 %

* All benchmarks compiled with GCC using -O3.

Table 5.3: Dhrystone and CoreMark scores using Rocket, with and without macro-op fusion.

Core configuration	Compiler	Dhrystone*(DMIPS/MHz)	CoreMark†(CM/MHz)
DefaultConfigWithFusion	gcc	4.67	2.03
DefaultConfig	gcc	4.67	2.01

* Compiled using `riscv64-unknown-linux-gnu-gcc -O2 -fno-inline -Wl,-Ttext-segment,0x10000 dhrystone.c -o dhrystone`, GCC version 8.2.0.
† Compiled using `riscv64-unknown-elf-gcc -O2`, GCC version 7.2.0.

Dhrystone and CoreMark

Note that the options used to compile Dhrystone are taken from the RISC-V foundation website[1].

[1] https://riscv.org/2014/10/about-our-dhrystone-benchmarking-methodology/, and following ARM's instructions on Dhrystone benchmarking (http://infocenter.arm.com/help/topic/com.arm.doc.dai0273a/DAI0273A_dhrystone_benchmarking.pdf).

Table 5.4: Synthesis results for Rocket

Core configuration	Core area* (μm^2)	Core area† (relative)
DefaultConfig	42016.8548	1
DefaultConfigWithFusion	42651.2649	1.0151

* Generated using a low-leakage 40 nm technology with Synopsys tools.
† Relative to DefaultConfig

5.1.4 Core area differences

The same four configurations of Rocket were synthesised for a low-leakage 40 nm technology in order to determine the effect of macro-op fusion on core area and clock frequency, targeting a clock frequency of 350 MHz. This synthesis resulted in the core areas and max frequencies described in Table 5.4.

5.2 Macro-op fusion optimisation in LLVM

This section describes the evaluation of the optimisation pass added to the LLVM compiler, which is intended to improve the performance of programs running on macro-op fusion enabled targets such as the modified Rocket core evaluated previously. The purpose of this optimisation pass is primarily to improve performance, so the key metric is to determine the percentage decrease in effective instruction count, rather than the percentage increase in macro-op fusion opportunities.

A modified version of LLVM was used to compile the SPEC integer benchmark set, both with and without macro-fusion optimisations enabled, and was simulated using Spike to produce the results shown in Table 5.5.

Table 5.5: Results of evaluating LLVM optimisations using Spike. Full raw results are listed in appendix B

Benchmark	Baseline			Optimised*		
	Instruction count	Fusion opportunities (%)	Potential opportunities (%)	Instruction count	Fusion opportunities (%)	Potential opportunities (%)
400.perlbench	1.13×10^9	1.06	1.49	1.13×10^9	1.11	1.49
401.bzip2	3.99×10^{10}	4.98	10.81	3.99×10^{10}	10.56	10.81
403.gcc	6.77×10^9	2.46	2.86	6.77×10^9	2.56	2.86
429.mcf	3.28×10^9	1.49	1.80	3.28×10^9	1.49	1.80
445.gobmk	1.86×10^{11}	3.05	3.53	1.86×10^{11}	3.05	3.53
456.hmmer	6.79×10^{10}	2.01	3.89	6.79×10^{10}	2.50	3.89
458.sjeng	3.54×10^{10}	4.27	5.51	3.54×10^{10}	4.84	5.51
462.libquantum	4.95×10^8	0.87	1.94	4.95×10^8	0.88	1.94
464.h264ref	1.41×10^{11}	2.89	6.27	1.41×10^{11}	5.78	6.27
471.omnetpp	3.23×10^9	1.34	2.11	3.23×10^9	1.37	2.11
473.astar	3.87×10^{10}	2.52	2.78	3.87×10^{10}	2.57	2.78
483.Xalan	5.89×10^8	0.82	1.11	5.89×10^8	0.87	1.10
Arithmetic mean	–	2.31	3.67	–	3.13	3.67

* LLVM IR compiled using `llc -enable-misched=true -riscv-macro-fusion=true -misched-fusion=true -misched-postra=true -enable-post-misched=true`

Chapter 6

Summary and Conclusions

This chapter summarises the experiments undertaken throughout this project, which were motivated by prior work described in Chapter 3, based on design work presented in Chapter 4 and evaluated in Chapter 5. The conclusions drawn from each experiment are addressed here in turn, followed by a description of possibilities for future work, which are either outside the scope of this project, or were not addressed due to time constraints.

6.1 Summary

The three groups of experiments presented in this report were inspired primarily by Celio et al. [6] in which they described potential speed-up in RISC-V CPU's due to macro-op fusion, and exploration of micro-architectural design space described by Azizi et al. [20].

The goals of this project were in three distinct areas: (i) macro-op fusion, in which two dependent instructions are fused and executed simultaneously at run-time based on the register values used; (ii) opportunities for dynamic optimisations, which seek to either discard trivial computations based on mathematical identities or otherwise reduce work based on the runtime values of specific registers; and (iii) a series of limit studies to investigate the potential speed-up in an in-order core,

such as Rocket, with a small amount of additional hardware to provide partial dual issue.

6.1.1 Macro-op fusion

Celio et al. [6] demonstrated a potential 2.25 % speed-up on the SPEC integer benchmark set when three idiomatic pairs of dependent instructions were fused in a RISC-V in-order pipeline: load-effective address, indexed load, and clear upper-word. This speed-up increased to ∼5 % in the presence of potential compiler optimisations using manually-acquired data.

This project extended upon Celio et al. [6] in a number of ways: beginning by successfully reproducing their results, then automating the limit study, and crucially implementing macro-op fusion as an optimisation in UC Berkeley's Rocket in-order CPU. Additional work was performed to add compiler optimisations to the LLVM compiler framework, to more effectively take advantage of the macro-op fusion hardware added to Rocket.

Results

The work of Celio et al. was reproduced using a range of compilers, including the configuration used by the original authors, demonstrating a slightly higher 4.28 % baseline speed-up using a more recent compiler version and taking into account the literal generation idiom, with a theoretical 7.08 % reduction in effective instruction count available with perfect compiler optimisations. A similar range of potential fusion opportunities were found, compared to prior work, and hence three macro-fusion opportunities were considered: load effective address, indexed load, and clear upper word. However, unlike Celio et al. [6], this work also demonstrated that literal generation using `lui`/`addi` would be a worthwhile addition.

Macro-op fusion was added to the Rocket CPU core, resulting in a 1.51 % increase in core area compared to an equivalent baseline system, with no impact on critical path length. It was further demonstrated in simulation that for a subset of the

rv8-bench benchmark set, a statistically significant improvement in performance was enabled, with no statistically significant performance regressions from the remaining benchmarks (to a 95 % confidence level). However, the speed-up observed was much smaller than the speed-up predicted by Spike simulations, most likely due to the impact of caching and branch-prediction, which are not modelled by Spike.

An additional compiler optimisation pass was added to the RISC-V back-end of LLVM, enabling a 36 % increase in available fusion opportunities. This optimisation lead to an overall potential speed-up of 3.13 %, compared to the 3.67 % predicted by the prior limit study, and 2.31 % available without any modifications at all.

Conclusion

The macro-op fusion experiment results demonstrate that with only a 1.51 % increase in core area, a statistically significant improvement in performance is available for the 'rv8-bench' benchmark set and for a subset of the smaller 'riscv-tests' benchmark set. Due to simulation limitations, a more complete investigation using the SPEC integer benchmark set was not completed, and hence it is unclear how other parts of the CPU core may interact with macro-op fusion in a more realistic benchmark. However, the small amount of testing performed on 'rv8-bench' demonstrated on average 0.79 % speed-up, including up to 2.23 % on one paritcular benchmark. Overall benchmark results, in combination with the small increase in core area, seem promising, and macro-op fusion appears to be a worthwhile optimisation for an in-order 5-stage pipeline, such as Rocket, given additional refinement and testing.

6.1.2 Dynamic opportunities

Inspired by the work of Yi and Lilja [1], as well as a desire to explore additional design space in the field of macro-op fusion, the second stage of this project involved a number of limit studies investigating potential performance gains due to register

values known at run-time (e.g., where both registers are zero, one, or negative one).

Two studies were performed: the first recreated the results of Yi and Lilja [1] in the context of RISC-V, demonstrating that the reported 8 % speed-up in the architecture used by the authors does not translate well to a RISC pipeline, such as Rocket. Only a 0.50 % instruction count reduction was achived when trivial computations were discarded. This poor performance is likely due to the inability to make use of trivial operations such as `add r1,r2,r3` when `r3 = 0`, since a move instruction is a pseudo instruction for `addi` instruction in RISC-V, taking the same path through the Rocket pipeline.

The suggestion, which was considered too late in the project for investigation, was that a similar approach could be taken to normal macro-op fusion, i.e., simplifying a dependent pair of instructions in which the first is trivial by forwarding the trivial result of the first to the operand of the second. This potential avenue of investigation is left for future work.

Finally, to push the boundaries of dynamic opportunities, a simple limit study was used to demonstrate the limiting number of potential trivial operations using the RISC-V ISA simulator 'Spike'. This study demonstrated that fewer than 3.20 % of dynamically-executed instructions had operands equal to zero, and fewer than 1.70 % equal to one; however, this result is highly benchmark-dependent.

Conclusion

Following the results obtained by both the reproduction of Yi and Lilja [1], and a novel exploration of runtime values, it was concluded that dynamic optimisations were unlikely to make a significant impact on the performance of simple processors, such as Rocket; especially given the additional hardware likely required to demonstrate these opportunities early enough in the pipeline. In addition, the proportion of 'trivial' operations is likely to only decrease over time as compiler optimisations become more sophisticated and able to predict the runtime values of variables in a program.

6.1.3 Dual-issue

To push the boundaries of complexity-effective hardware, the final section of this project involved a fine-grained limit study of partial dual-issue approaches in a RISC-V pipeline, such as Rocket. These experiments were all performed in simulation using a very simple pipeline model, and hence represent a best-case scenario. However, these experiments uniquely demonstrate a large range of the potential design space, making use of hardware already implemented in Rocket, and show what performance could be gained if a small amount of extra hardware were added.

The dual-issue limit study considered a cross-section of potential cores, which were each modified according to the amount of processing resources added (whether the core was restricted to a single integer ALU, an integer ALU with additional hardware for branching or two ALU's) and the amount of additional hardware added to the register file and forwarding network. Each processor configuration was evaluated with every benchmark in the SPEC integer benchmark set, leading to the full set of results reproduced in Appendix D. These results demonstrated that, as expected, a great deal of extra performance is available in full dual-issue cores. However, $\sim$5.38 % of dynamically executed instructions can be implemented without a significant increase in hardware, by allowing forwarded and shared register reads between independent pairs, and duplicating the integer ALU.

Conclusion

The results produced demonstrated that by implementing dual-issue, a significant speed-up is available, for a range of complexity points. $\sim$5 % additional performance is available simply by dual-issuing ALU instructions and branches without an additional register read port or write port. Unfortunately, this work was restricted to a simulated limit study, leaving additional future work, including a flexible hardware implementation able to more thoroughly explore the complexity and power consumption. A more complete pipeline model of Rocket that can more accurately predict performance improvements, without the need for RTL-level simulations and hardware implementations, could also be explored in the

future.

6.2 Future Work

Following the various experiments performed in this project, a number of opportunities for future work can be highlighted:

- Consider other hardware trade-offs in macro-op fusion. What additional benefit could be obtained with a small increase in core size by adding additional register read ports, forwarding or register sharing? These additions would allow more complex idioms, requiring three register reads or two register writes to be fused.
- More thoroughly investigate the hybrid approach of trivial computations combined with macro-op fusion: a pair of instructions, beginning with a trivial operation, followed by a dependent instruction can be fused if the intermediate result is overwritten by the second instruction.
- Extend Rocket, or another similar core, to a more complete, fully configurable core to allow a parameter sweep of partial dual issue approaches without the need for simulation. Similar to Azizi et al. [20], this would allow the sweet-spot between complexity, power consumption and performance to be found in a far more fine-grained manner. Macro-op fusion, and possibly trivial computation optimisations, could then be added to see how far the performance of an in-order core can be extended.
- Consider very simple OOO models, similar to Patsidis et al. [16], possibly using limited forms of register renaming.
- Develop a more accurate high-performance pipeline model for Rocket, and for in-order cores in general. When compared to Spike, this model would allow a greater degree of reliability that ideas developed in simulation would translate well to real hardware.
- In the context of dual-issue, consider the possibility of register banking to

effectively increase the number of register read ports, based on the non-uniform behaviour of RISC-V's register accesses.

Appendix A

Raw riscv-tests data

The raw data used to evaluate Rocket's performance, with and without macro-op fusion enabled is shown in the following table. This simply lists the number of cycles, as reported by the Rocket simulator for eleven different runs (allowing the performance of Rocket to be evaluated, despite its non-determinism).

Benchmarks were based on the riscv-tests open source project[1]. These were compiled with GCC using `-O3`.

These benchmarks were run using Rocket's built-in simulator based on generated Verilog converted to C++ source code using Verilator [26], and run in parallel using a Python script and the Python 'multiprocessing' library. Statistical testing was performed using the SciPy statistics library [27, 28] and Student's t-test. Processed results are shown in table 5.1.

```
emulator-freechips.rocketchip.system-DefaultConfig -m 100000000 -c pk
$BMK
```

```
emulator-freechips.rocketchip.system-DefaultConfigWithFusion -m 100000000
-c pk $BMK
```

[1] https://github.com/riscv/riscv-tests

		Cycles										
Benchmark	Config	0	1	2	3	4	5	6	7	8	9	10
dhrystone	Baseline	490992	492360	491772	491136	470828	491988	470120	469856	492588	492996	469748
	Fusion	470564	470240	470012	470564	470684	470108	470072	469940	470948	470348	471092
median	Baseline	162148	162928	161848	163084	162100	162172	162124	162460	161968	161944	163840
	Fusion	162256	162244	161956	161956	162412	163480	162388	162040	162256	162316	162184
mm	Baseline	762004	761584	761500	762332	760556	762476	764620	761440	762208	762568	763060
	Fusion	761228	760964	761132	764876	760928	761216	765776	761096	763028	760904	760688
mt-matmul	Baseline	273240	273120	272724	274308	272412	273660	272880	273936	272640	272472	272592
	Fusion	273504	272700	273000	273180	273576	273132	272736	273288	272856	273516	273384
mt-vvadd	Baseline	743260	744028	743320	741340	740944	741124	742900	744124	741256	743644	741172
	Fusion	741352	745180	742804	740848	743548	742984	742468	741088	742012	742816	743128
multiply	Baseline	170336	169220	170348	170888	169388	171320	169232	169232	169244	169568	169268
	Fusion	169460	169724	169904	169832	169784	169412	169448	169388	169688	172220	169868
pmp	Baseline	1651976	1651916	1652612	1653836	1631368	1631116	1651868	1632676	1631188	1631200	1630972
	Fusion	1589756	1589588	1589876	1590056	1589672	1610352	1589888	1589888	1610844	1589756	1610388
qsort	Baseline	724004	724664	722744	722744	724532	723920	723776	723848	723524	722720	722864
	Fusion	722540	723680	722852	724976	729056	725372	722456	723056	725372	723212	723416
rsort	Baseline	864104	863300	863480	862184	861248	863504	862676	862124	861176	862400	862880
	Fusion	865316	861368	861344	861788	861968	861608	847156	862148	840604	864824	861020
spmv	Baseline	894392	894476	895028	894176	896252	895076	896912	894524	894332	895208	895016
	Fusion	896108	893612	878188	878260	877984	878116	899312	899048	896024	873940	899012
towers	Baseline	120020	121028	120056	120104	120176	119924	121100	120116	120188	120212	120908
	Fusion	120368	119984	120476	120176	120104	120140	120140	120464	122744	120104	120092
vvadd	Baseline	169672	169636	169132	169216	168928	169084	168940	170128	170128	170224	170224
	Fusion	169108	169540	169216	169564	169216	169216	169264	169564	169804	169024	169540

Appendix B

Raw LLVM evaluation data

	bmk	cc	total_dynamic	overall	lea	ix_ld	cuw	lit
1	400.perlbench	clang	1.127254e+09	1.063330	0.106710	0.288380	0.300383	0.367857
2	400.perlbench	clang_fusion	1.127255e+09	1.108621	0.106738	0.325229	0.309933	0.366720
4	401.bzip2	clang	3.987893e+10	4.975978	0.055219	3.484149	1.035568	0.401043
5	401.bzip2	clang_fusion	3.987893e+10	10.564332	0.110436	7.568798	2.484055	0.401043
7	403.gcc	clang	6.765867e+09	2.463546	0.209959	1.043416	0.257287	0.952885
8	403.gcc	clang_fusion	6.765862e+09	2.560265	0.227034	1.116418	0.286775	0.930039
10	429.mcf	clang	3.277842e+09	1.489049	0.852130	0.617259	0.008036	0.011624
11	429.mcf	clang_fusion	3.277842e+09	1.489049	0.852130	0.617259	0.008036	0.011624
13	445.gobmk	clang	1.864057e+11	3.046387	0.280760	1.597126	0.120807	1.047695
14	445.gobmk	clang_fusion	1.864057e+11	3.052827	0.329887	1.655765	0.125508	0.941667
16	456.hmmer	clang	6.793973e+10	2.013813	0.094525	0.808597	0.588925	0.521765
17	456.hmmer	clang_fusion	6.793973e+10	2.498792	0.094525	1.293374	0.589128	0.521765
19	458.sjeng	clang	3.536627e+10	4.266232	0.525396	1.482426	1.015501	1.242908
20	458.sjeng	clang_fusion	3.536627e+10	4.844280	0.663786	1.811837	1.043286	1.325371
22	462.libquantum	clang	4.947220e+08	0.865573	0.293817	0.007903	0.351845	0.212008
23	462.libquantum	clang_fusion	4.947220e+08	0.875160	0.293817	0.007903	0.361431	0.212008
25	464.h264ref	clang	1.405516e+11	2.888405	0.096216	2.684568	0.067964	0.039657
26	464.h264ref	clang_fusion	1.405516e+11	5.775608	0.107976	5.553407	0.073411	0.040814
28	471.omnetpp	clang	3.229124e+09	1.342531	0.238156	0.203952	0.367590	0.532834
29	471.omnetpp	clang_fusion	3.229124e+09	1.374436	0.238156	0.203952	0.399495	0.532834
31	473.astar	clang	3.865405e+10	2.519341	0.015239	2.340304	0.150870	0.012928
32	473.astar	clang_fusion	3.865405e+10	2.572211	0.015481	2.343976	0.199826	0.012928
34	483.Xalan	clang	5.890704e+08	0.821091	0.047354	0.233399	0.261051	0.279287
35	483.Xalan	clang_fusion	5.891963e+08	0.868058	0.075646	0.238735	0.274534	0.279143

Appendix C

Raw CoreMark data

Raw output from CoreMark running on Rocket in simulation:

```
$ time emulator-freechips.rocketchip.system-DefaultConfig \
+max-cycles=10000000000 +cycle-count pk ./coremark.riscv  \
0x0 0x0 0x66 1000 7 1 2000
This emulator compiled with JTAG Remote Bitbang client. To
enable, use +jtag_rbb_enable=1.
Listening on port 42775
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 498021460
Total time (secs): 1.24505365
Iterations/Sec   : 803.1782
ERROR! Must execute for at least 10 secs for a valid result!
Iterations       : 1000
Compiler version : GCC7.2.0
Compiler flags   : -O2
Memory location  : Please put data memory location here
                        (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Errors detected
*** PASSED *** Completed after 501455932 cycles
```

```
 +max-cycles=10000000000 +cycle-count pk ./coremark.riscv
0x0 0x0 0x66 1000 7
42199.94s user 112.06s system 181% cpu 6:29:03.22 total
```

With macro-op fusion optimisation enabled:

```
$ time emulator-freechips.rocketchip.system-DefaultConfig \
+max-cycles=10000000000 +cycle-count pk ./coremark.riscv  \
0x0 0x0 0x66 1000 7 1 2000
This emulator compiled with JTAG Remote Bitbang client. To
enable, use +jtag_rbb_enable=1.
Listening on port 42775
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 498021460
Total time (secs): 1.24505365
Iterations/Sec   : 803.1782
ERROR! Must execute for at least 10 secs for a valid result!
Iterations       : 1000
Compiler version : GCC7.2.0
Compiler flags   : -O2
Memory location  : Please put data memory location here
                        (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xd340
Errors detected
*** PASSED *** Completed after 501455932 cycles
 +max-cycles=10000000000 +cycle-count pk ./coremark.riscv
0x0 0x0 0x66 1000 7
42199.94s user 112.06s system 181% cpu 6:29:03.22 total
```

Appendix D

Raw dual-issue data

Table D.1: Upper bound of speed-up available from dual-issue for each SPEC integer benchmark.

		ALU	ALU + branch	2 x ALU
400.perlbench	baseline	0.24%	0.25%	2.41%
	share	0.24%	1.48%	3.83%
	share + forward	0.24%	3.73%	7.33%
	share + forward + 2 read ports + write port	0.27%	4.90%	22.25%
	share + forward + read port	0.24%	4.72%	8.40%
	share + forward + read port + write port	0.27%	4.76%	22.02%
	share + forward + write port	0.27%	3.77%	20.68%
	share + forward + write port + 1 forwarding cycles	0.27%	4.10%	20.74%
	share + forward + write port + 2 forwarding cycles	0.27%	4.25%	20.94%
	share + forward + write port + 3 forwarding cycles	0.27%	4.40%	21.09%
	share + forward + write port + 4 forwarding cycles	0.27%	4.43%	21.24%
	share + forward + write port + 5 forwarding cycles	0.27%	4.44%	21.28%
	share + forward + write port + 6 forwarding cycles	0.27%	4.56%	21.31%

401.bzip2	baseline	0.00%	0.00%	0.29%
	share	0.00%	0.02%	0.35%
	share + forward	0.00%	4.61%	6.00%
	share + forward + 2 read ports + write port	0.00%	5.64%	27.42%
	share + forward + read port	0.00%	5.34%	8.97%
	share + forward + read port + write port	0.00%	5.34%	27.33%
	share + forward + write port	0.00%	4.61%	23.86%
	share + forward + write port + 1 forwarding cycles	0.00%	4.73%	24.39%
	share + forward + write port + 2 forwarding cycles	0.00%	5.10%	24.72%
	share + forward + write port + 3 forwarding cycles	0.00%	5.10%	26.61%
	share + forward + write port + 4 forwarding cycles	0.00%	5.23%	26.97%
	share + forward + write port + 5 forwarding cycles	0.00%	5.50%	27.01%
	share + forward + write port + 6 forwarding cycles	0.00%	5.51%	27.03%
403.gcc	baseline	0.01%	0.01%	0.56%
	share	0.01%	0.39%	3.40%
	share + forward	0.01%	3.27%	6.84%
	share + forward + 2 read ports + write port	0.01%	4.09%	22.66%
	share + forward + read port	0.01%	3.96%	8.10%
	share + forward + read port + write port	0.01%	3.96%	22.31%
	share + forward + write port	0.01%	3.27%	20.29%
	share + forward + write port + 1 forwarding cycles	0.01%	3.45%	20.66%
	share + forward + write port + 2 forwarding cycles	0.01%	3.57%	20.90%
	share + forward + write port + 3 forwarding cycles	0.01%	3.70%	21.04%
	share + forward + write port + 4 forwarding cycles	0.01%	3.75%	21.20%
	share + forward + write port + 5 forwarding cycles	0.01%	3.77%	21.25%
	share + forward + write port + 6 forwarding cycles	0.01%	3.80%	21.37%

429.mcf	baseline	0.00%	0.01%	0.14%
	share	0.00%	0.05%	0.39%
	share + forward	0.00%	1.71%	3.67%
	share + forward + 2 read ports + write port	0.00%	2.65%	15.31%
	share + forward + read port	0.00%	2.53%	5.03%
	share + forward + read port + write port	0.00%	2.53%	15.27%
	share + forward + write port	0.00%	1.71%	11.07%
	share + forward + write port + 1 forwarding cycles	0.00%	2.19%	11.75%
	share + forward + write port + 2 forwarding cycles	0.00%	2.19%	11.84%
	share + forward + write port + 3 forwarding cycles	0.00%	2.21%	12.12%
	share + forward + write port + 4 forwarding cycles	0.00%	2.22%	12.16%
	share + forward + write port + 5 forwarding cycles	0.00%	2.22%	12.25%
	share + forward + write port + 6 forwarding cycles	0.00%	2.31%	12.26%
445.gobmk	baseline	0.02%	0.02%	1.28%
	share	0.02%	0.33%	2.39%
	share + forward	0.02%	3.47%	7.50%
	share + forward + 2 read ports + write port	0.09%	4.37%	26.06%
	share + forward + read port	0.02%	4.25%	9.44%
	share + forward + read port + write port	0.09%	4.27%	25.83%
	share + forward + write port	0.09%	3.49%	23.01%
	share + forward + write port + 1 forwarding cycles	0.09%	3.71%	23.46%
	share + forward + write port + 2 forwarding cycles	0.09%	3.95%	23.74%
	share + forward + write port + 3 forwarding cycles	0.09%	3.98%	23.92%
	share + forward + write port + 4 forwarding cycles	0.09%	4.07%	24.06%
	share + forward + write port + 5 forwarding cycles	0.09%	4.10%	24.11%
	share + forward + write port + 6 forwarding cycles	0.09%	4.14%	24.42%

456.hmmer	baseline	1.08%	1.08%	2.66%
	share	1.08%	1.08%	2.66%
	share + forward	1.17%	3.89%	6.70%
	share + forward + 2 read ports + write port	1.17%	4.95%	20.07%
	share + forward + read port	1.17%	4.86%	7.87%
	share + forward + read port + write port	1.17%	4.86%	19.63%
	share + forward + write port	1.17%	3.89%	18.78%
	share + forward + write port + 1 forwarding cycles	1.17%	3.98%	18.87%
	share + forward + write port + 2 forwarding cycles	1.17%	3.98%	19.76%
	share + forward + write port + 3 forwarding cycles	1.17%	4.77%	20.03%
	share + forward + write port + 4 forwarding cycles	1.17%	4.77%	19.94%
	share + forward + write port + 5 forwarding cycles	1.17%	4.86%	20.12%
	share + forward + write port + 6 forwarding cycles	1.17%	4.86%	20.29%
458.sjeng	baseline	0.00%	0.19%	1.17%
	share	0.00%	0.34%	1.74%
	share + forward	0.00%	3.54%	6.07%
	share + forward + 2 read ports + write port	0.00%	4.76%	25.96%
	share + forward + read port	0.00%	4.73%	8.27%
	share + forward + read port + write port	0.00%	4.73%	25.91%
	share + forward + write port	0.00%	3.54%	23.38%
	share + forward + write port + 1 forwarding cycles	0.00%	3.58%	23.85%
	share + forward + write port + 2 forwarding cycles	0.00%	3.81%	24.07%
	share + forward + write port + 3 forwarding cycles	0.00%	3.85%	24.58%
	share + forward + write port + 4 forwarding cycles	0.00%	3.97%	24.95%
	share + forward + write port + 5 forwarding cycles	0.00%	4.09%	25.03%
	share + forward + write port + 6 forwarding cycles	0.00%	4.14%	25.22%

462.libquantum	baseline	0.10%	0.10%	0.12%
	share	0.10%	0.12%	0.14%
	share + forward	0.10%	1.29%	1.32%
	share + forward + 2 read ports + write port	0.10%	1.48%	15.42%
	share + forward + read port	0.10%	1.39%	1.42%
	share + forward + read port + write port	0.10%	1.39%	15.24%
	share + forward + write port	0.10%	1.29%	14.58%
	share + forward + write port + 1 forwarding cycles	0.10%	1.30%	14.59%
	share + forward + write port + 2 forwarding cycles	0.10%	1.30%	15.13%
	share + forward + write port + 3 forwarding cycles	0.10%	1.30%	15.14%
	share + forward + write port + 4 forwarding cycles	0.10%	1.30%	15.14%
	share + forward + write port + 5 forwarding cycles	0.10%	1.30%	15.14%
	share + forward + write port + 6 forwarding cycles	0.10%	1.30%	15.14%
464.h264ref	baseline	0.01%	0.01%	0.14%
	share	0.01%	0.04%	0.42%
	share + forward	0.01%	0.87%	1.93%
	share + forward + 2 read ports + write port	0.02%	2.76%	31.99%
	share + forward + read port	0.01%	2.40%	4.71%
	share + forward + read port + write port	0.02%	2.40%	29.97%
	share + forward + write port	0.02%	0.87%	23.06%
	share + forward + write port + 1 forwarding cycles	0.02%	1.04%	25.02%
	share + forward + write port + 2 forwarding cycles	0.02%	2.23%	26.52%
	share + forward + write port + 3 forwarding cycles	0.02%	2.28%	27.74%
	share + forward + write port + 4 forwarding cycles	0.02%	2.29%	28.71%
	share + forward + write port + 5 forwarding cycles	0.02%	2.30%	28.57%
	share + forward + write port + 6 forwarding cycles	0.02%	2.32%	29.59%

471.omnetpp	baseline	0.78%	0.78%	2.47%
	share	0.87%	1.49%	3.29%
	share + forward	0.87%	3.49%	5.97%
	share + forward + 2 read ports + write port	1.03%	4.60%	23.67%
	share + forward + read port	0.87%	4.25%	8.01%
	share + forward + read port + write port	1.03%	4.41%	23.35%
	share + forward + write port	1.03%	3.66%	19.22%
	share + forward + write port + 1 forwarding cycles	1.03%	3.85%	19.48%
	share + forward + write port + 2 forwarding cycles	1.03%	3.98%	19.95%
	share + forward + write port + 3 forwarding cycles	1.03%	4.12%	20.11%
	share + forward + write port + 4 forwarding cycles	1.03%	4.12%	20.42%
	share + forward + write port + 5 forwarding cycles	1.03%	4.22%	20.73%
	share + forward + write port + 6 forwarding cycles	1.03%	4.22%	20.79%
473.astar	baseline	0.04%	0.04%	0.14%
	share	0.04%	0.12%	0.22%
	share + forward	0.05%	3.25%	4.59%
	share + forward + 2 read ports + write port	0.05%	4.34%	16.89%
	share + forward + read port	0.05%	4.26%	5.71%
	share + forward + read port + write port	0.05%	4.26%	16.80%
	share + forward + write port	0.05%	3.25%	16.17%
	share + forward + write port + 1 forwarding cycles	0.05%	3.26%	16.44%
	share + forward + write port + 2 forwarding cycles	0.05%	4.24%	16.45%
	share + forward + write port + 3 forwarding cycles	0.05%	4.24%	16.46%
	share + forward + write port + 4 forwarding cycles	0.05%	4.24%	16.49%
	share + forward + write port + 5 forwarding cycles	0.05%	4.32%	16.58%
	share + forward + write port + 6 forwarding cycles	0.05%	4.33%	16.59%

483.Xalan	baseline	0.10%	0.11%	0.75%
	share	0.12%	0.63%	1.34%
	share + forward	0.13%	5.31%	6.61%
	share + forward + 2 read ports + write port	0.16%	5.82%	20.74%
	share + forward + read port	0.13%	5.67%	7.53%
	share + forward + read port + write port	0.16%	5.71%	20.26%
	share + forward + write port	0.16%	5.35%	18.05%
	share + forward + write port + 1 forwarding cycles	0.16%	5.44%	18.27%
	share + forward + write port + 2 forwarding cycles	0.16%	5.58%	18.46%
	share + forward + write port + 3 forwarding cycles	0.16%	5.61%	18.55%
	share + forward + write port + 4 forwarding cycles	0.16%	5.62%	18.81%
	share + forward + write port + 5 forwarding cycles	0.16%	5.64%	18.92%
	share + forward + write port + 6 forwarding cycles	0.16%	5.65%	18.97%

Bibliography

[1] Joshua J Yi and David J Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 462–465. IEEE, 2002.

[2] Anant Agarwal and Markus Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, pages 750–753. ACM, 2007.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[4] Bob Wheeler. SiFive Raises RISC-V Performance. *The Linley Group MICROPROCESSOR Report (November 2016)*, 2018.

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[6] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V. *arXiv preprint arXiv:1607.02318*, 2016.

[7] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[8] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: Base user-level isa. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[9] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[11] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[12] Michael Clark and Bruce Hoult. rv8: a high performance RISC-V to x86 binary translator. 2017.

[13] James Balfour, Richard Harting, and William Dally. Operand registers and explicit operand forwarding. *IEEE Computer Architecture Letters*, 8(2):60–63, 2009.

[14] William J Dally, James Balfour, David Black-Shaffer, James Chen, R Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, 2008.

[15] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[16] Karyofyllis Patsidis, Dimitris Konstantinou, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed Instruction Set Extension. *Microprocessors and Microsystems*, 61:1 – 10, 2018. ISSN 0141-9331. doi: https://doi.org/10.1016/j.micpro.2018.05.007. URL http://www.sciencedirect.com/science/article/pii/S0141933118300048.

[17] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[18] Christopher Celio. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html.

[19] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[20] Omid Azizi, Aqeel Mahesri, Benjamin C Lee, Sanjay J Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 26–36. ACM, 2010.

[21] Christohper Celio, Krste Asanovic, and David Patterson. ISA Shootout: Comparing RISC--V, ARM, and x86 on SPECInt 2006. 2016 RISC-V Workshop 4, 2016. URL https://riscv.org/wp-content/uploads/2016/07/Tue1130celio-fusion-finalV2.pdf.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[23] Bandic Z. Zvonimir, Robert Golla, and Dejan Vucinic. CPU Project in Western Digital: From Embedded Cores for Flash Controllers to Vision of Datacenter Processors with Open Interfaces. December 2018 RISC-V Workshop, 2018. URL https://content.riscv.org/wp-content/uploads/2018/12/13.10-Bandic-Golla-Vucinic-CPU-Project-in-Western-Digital-From-Embedded-Cores-for-F
pdf.

[24] Rezaur Rahman. *Xeon Phi Core Microarchitecture*, pages 49–64. Apress, Berkeley, CA, 2013. ISBN 978-1-4302-5927-5. doi: 10.1007/978-1-4302-5927-5_4. URL https://doi.org/10.1007/978-1-4302-5927-5_4.

[25] LowRISC. Rocket core overview. URL https://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/.

[26] Verilator. Intro - Verilator - Veripool. URL https://www.veripool.org/projects/verilator/wiki/Intro.

[27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed 28th May 2019].

[28] Damir Kalpić, Nikica Hlupić, and Miodrag Lovrić. *Student's t-Tests*, pages 1559–1563. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_641. URL https://doi.org/10.1007/978-3-642-04898-2_641.