MEng Individual Project

Imperial College London

Department of Computing

Better Than Registers: Designing an Instruction Set to Make O-o-O Microarchitecture Simpler and Faster

Author:
Stacey Wu

Supervisor:
Prof. Paul Kelly

Second Marker:
Dr. Holger Pirk

June 22, 2022

Abstract

In modern computer architecture, out-of-order superscalar processors use complex register rename logic to remove false data dependencies between instructions and improve instruction-level parallelism.

We design a novel architecture which encodes the data dependencies in the instruction at compile time and removes register renaming in the out-of-order pipeline. We design a new Instruction Set Architecture(ISA) and define the new instruction set in an ISA emulator. We also provide compilation, assembler and linker support to compile the program from C source code to executable file.

We modify the conventional out-of-order processor to remove register renaming and simulate the architecture with compiled C program in a cycle-accurate architecture modelling tool. Throughout the project, it is found that the performance benefits come from more efficient misprediction recovery mechanism.
We run some experiments and demonstrate that the run-time performance improves by up to 3.5% from removing register renaming from the reduced pipeline stages which reduces the branch misprediction penalty.

We also explore the microarchitectural benefits of the new architecture from removing register renaming and provide a qualitative analysis on other benefits arising from hardware simplification. This architecture also provides further performance benefits to exploit its statically constructed data dependence for selective recovery to furthuer reduce its mis-speculation penalty.

Acknowledgements

First and foremost, I would like to thank Prof. Paul Kelly for his continuous support and mentorship throughout the project, especially agreeing to schedule last minute meetings and replying emergency emails at random hours of the day.

My gratitude goes to Prof. Hidetsugu Irie and his research group from University of Tokyo for providing the source code of STRAIGHT backend.

I would also like to thank Ph.D. candidate Luke Panayi for helping me understand LLVM backend and providing me with numerous support in this project.

I am grateful for the amount of support and help I receive from my friends, Barbara, Gowoon, Tejas, Jason and many more throughout my project and degree. Without them, I would not have been here.

Last but not least, I would like to thank Tilman Roeder for the help in last minute running experiments, plotting diagrams and countless emotional support during the many breakdowns throughout the project.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

As the performance of CPU improved significantly in the past decades, superscalar architecture has gained a lot of traction. By executing more than one instruction per clock cycle, the architecture can simultaneously dispatch multiple instructions to different execution units in the processor. It, therefore, improves the throughput of the program run on a single processor. Register renaming, a technique to abstract logical registers from physical registers, is the most common technique used to solve the data dependency issues that arise from superscalar, out-of-order processor. When an instruction refers to a logical register, the processor will look the physical register up on the Register Mapping Table and map the logical register to the physical register.
However, register renaming has proven to be one of the biggest performance bottlenecks in the architecture design. In addition, due to the multi-ported nature of the register rename hardware, it also consumes a significant amount of power.
A novel idea of getting rid of register renaming is proposed. Instead of using register renaming to keep track of data dependency and control flow, each instruction can explicitly state the source operands in the instruction. The source operands can be indicated as a destination operand at a known distance from the current instruction.

1.1 Objectives

The objective of this project is to investigate the effect of eliminating register renaming by keeping track of data dependencies at compile time statically on execution performance and architectural complexity.
It can be achieved by designing a new Instruction Set Architecture(ISA) that allows each logical register to be written only once and discarded in a fixed period. Data dependency can be tracked in the instruction where the result of each instruction is written to a new register and source operand field encodes the distance between the consumer instruction and the producer instruction.
To investigate the performance of this architecture, a full instruction set needs to be designed to provide full architectural functionality. Binary and compilation support needs to be provided to run binary tests and compile source program to executables to run on an ISA simulator. Furthermore, to understand the performance of the new architecture, we also need to simulate the architecture in fine-grained details and compare it against a similar conventional ISA. This project will further explore the simplicity of the architectural design and its benefits compared to conventional RISC architecture.

1.2 Contributions

A past attempt in creating such a new ISA that skips register renaming by defining data dependency in instructions by a research group from University of Tokyo[10]. The project aims to use this novel architecture idea and try to use some of their solutions[9] as a point of inspiration.
The project has the following contributions:

1. Design a new ISA to realise the idea. The new ISA needs to be similar to conventional RISC ISA to create a commond ground for benchmarking peformance.

2. Describe the new ISA semantics in SAIL, a ISA description language and simulate instructions in their OCaml sequential emulator as an early stage concept prover.
3. Develop an assembler and a static linker to generate an executable in ELF format from assembly code
4. Model the microarchitecture in both single-cycle processor and out-of-order processor in gem5, an established platform for computer-system architecture research.
5. Simulate the new architecture in benchmark programs to evaluate the performance and test the hypothesis of effect of removing register renaming on branch misprediction penalty.
6. Analyse other microarchitectural benefits of removing register renaming and simplifying the hardware.

1.3 Conclusion

Simulation of STRAIGHT architecture in gem5 uncovers significant performance improvement, up to 3.5% from reducing the pipeline stages in the out-of-order processor. The hardware simplification – removing register rename logic, also brings power consumption benefits. Furthermore, by removing this hardware on the critical path, the latency of the critical path could be reduced further to allow over-clocking and scaling to larger instruction window size.
Furthermore, it is found that static computation of data dependency also brings potential benefits of further reducing misprediction penalty. With data dependence readily available at compile time, it makes selective recovery from mis-speculation easier and more scalable, an idea researched in EDGE instruction set architecture[8]. This project could serve as a base to study selective recovery scheme beyond current work.

1.4 Report Structure

In Chapter 2, we provide a review of relevant literature that forms the background knowledge related to this project.
In Chapter 3, we discuss the related work undertaken by other researchers and approaches in reducing the architectural complexity of register renaming logic and misspeculation penalty.
In Chapter 4, we design a new ISA from modifying the RISC-V ISA to realise the idea of STRAIGHT architecture.
In Chapter 5 and 6, we discuss the implementation details of ISA definition in SAIL emulator, static linker and assembler for assembly code compilation, and microarchitecture modelling in gem5.
In Chapter 7, we discuss how we run testbenches in the architectural simulator to test the hypothesis of removing register renaming.
In Chapter 8, we analyse benefits of architectural simplification in the aspects of performance, hardware and clock rate.
In Chapter 9, we conclude the project, limitations in the current design roundtrip including compiler and gem5 simulator, and possible future work.

Chapter 2

Background

This chapter will provide a background review of all necessary knowledge requried to understand the project, as well as review/introduction on softwares/tools used in the project.

2.1 Superscalar Architecture

Superscalar architecture[11] is a modern architecture design which contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. This architecture exploits instruction-level parallelism(ILP) by allowing more throughput – the number of instructions that can be executed in a unit of time. In a superscalar CPU shown in Figure 2.1, the dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching each to one of the several functional units inside a single CPU. The superscalar architecture overcomes the limitations of pipelined simple-scalar in performance.

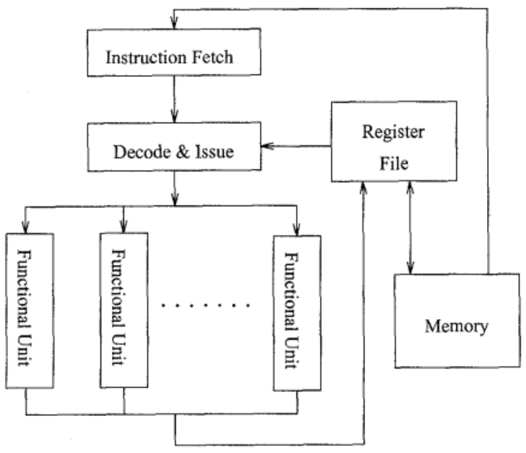


Figure 2.1: Basic Superscalar Architecture, Taken from [1]

2.1.1 Out-of-Order Execution

Out-of-Order(OoO) execution is a process where a computer fetches the instructions and determines which data is required and executes the instructions. The processor executes the instructions in an order of availability of data instead of original order of the instructions. By out-of-order execution, it efficiently fetches and executes instructions to prevent the processor getting into idle stage, waiting on generation of dependent data. It has become a common technique to improve the performance of superscalar architecture through occupying the functional units more efficiently.

When more than one instruction reads/writes to a particular location as an operand, executing those instructions in an order different from the original program order will lead to data dependency hazards[2]. There are three types of data dependency hazard to deal with, namely:

1. Read-after-write (RAW): true dependence where a read from a register is followed a previous write to the register
2. Write-after-read (WAR): anti dependence where a read from a register must return the value before a new value is written to the register
3. Write-after-write (WAW): false/output dependence where two successive writes to one register leave the location containing the result of the later write

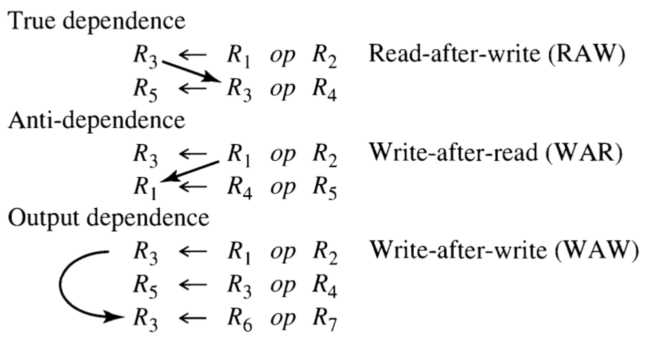

True dependence
$R_3 \leftarrow R_1$ op R_2
$R_5 \leftarrow R_3$ op R_4
Read-after-write (RAW)
Anti-dependence
$R_3 \leftarrow R_1$ op R_2
$R_1 \leftarrow R_4$ op R_5
Write-after-read (WAR)
Output dependence
$R_3 \leftarrow R_1$ op R_2
$R_5 \leftarrow R_3$ op R_4
$R_3 \leftarrow R_6$ op R_7
Write-after-write (WAW)

Figure 2.2: Data hazards, Taken from [2]

2.1.2 Register Renaming

Register renaming [1], is a technique that solves the false and anti data dependence hazards dynamically. As shown in Figure 2.3, register rename logic translates logical register file into physical register file by accessing a map table indexed by logical register. In addition to the map table, dependence check logic is required to detect cases where the logical register being renamed is written by an earlier instruction in the current group of instructions being renamed.
When an instruction that produces a result is decoded, the renaming logic allocates a free physical register. The logical destination register is mapped to the physical register and updated in the map table. Subsequent data-dependent instructions rename their source registers to access the physical register by register name lookup in the map table[3]. Register renaming is a critical technique in exploiting the additional ILP by solving data dependency hazards, yielding higher performance.

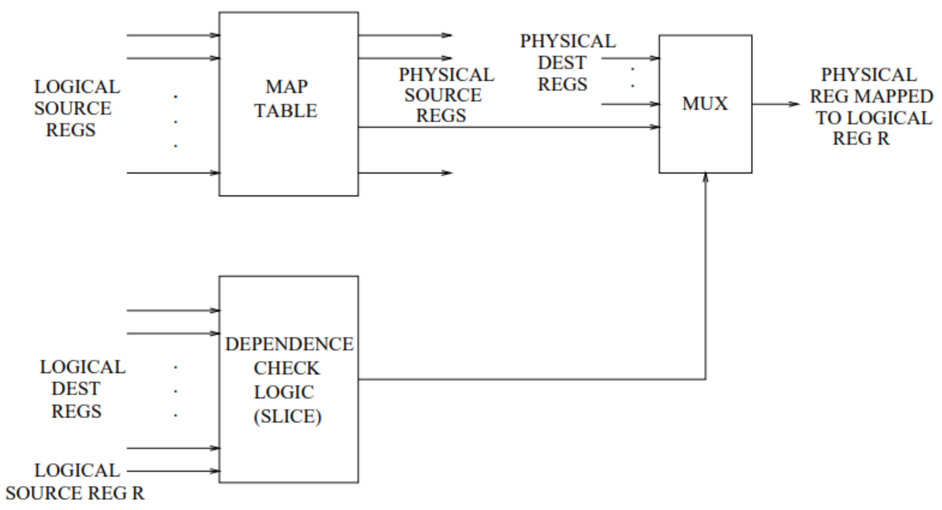
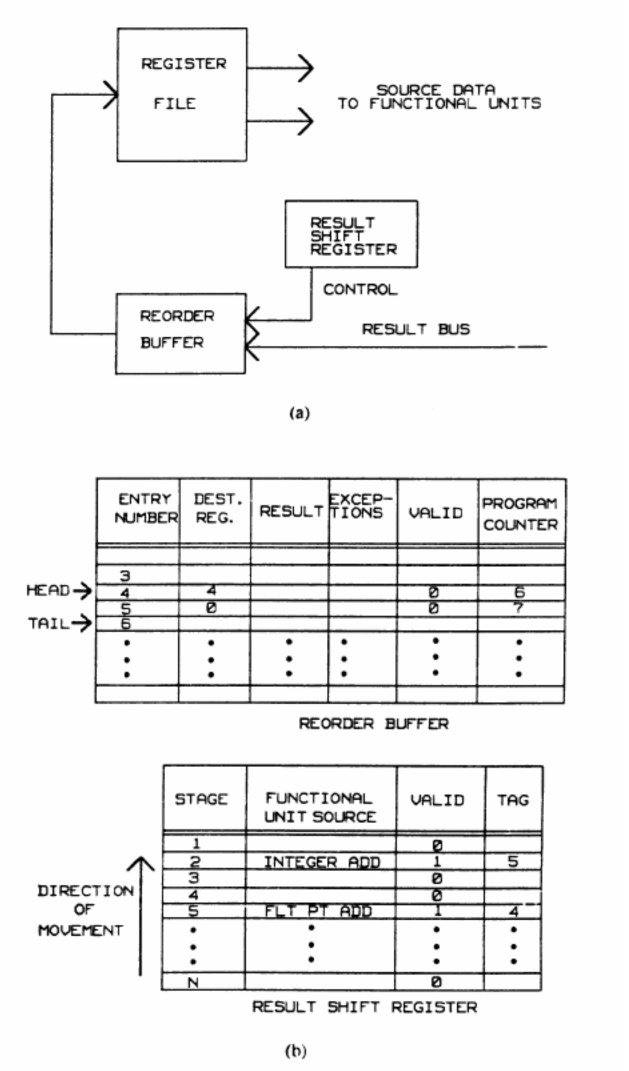


Figure 2.3: Register Rename Logic, Taken from [3]

2.1.3 Reorder Buffer

Register renaming allows instructions to execute out of order but to force them to commit in order, an extra commit phase is needed with an additional set of buffer to hold the results of instructions in order to prevent any irrevocable side effects when a branch misprediction or mis-speculation occurs.
Reorder Buffer(ROB)[12] is a circular queue with head and tail pointers, as shown in Figure 2.4. ROB tracks the state of all inflight instructions in the pipeline. After instructions are decoded and renamed, an instruction is dispatched and assigned an entry at the tail of the ROB. At the end of execution, the result is put back to the instruction's reorder buffer's position. When the register reaches the head of the ROB, the instruction will be committed and register file will be updated with the result. After that, ROB removes the instruction entry by moving the head pointer.
If a branch misprediction occurs, all instructions following the mispredicted branch are flushed in the ROB. Rename map table must also be reset to represent the true, non-speculative committed state. Execution is restarted at the correct successor of the branch.



ENTRY NUMBER	DEST. REG.	RESULT	EXCEP- TIONS	VALID	PROGRAM COUNTER
3					
4	4			0	6
5	0			0	7
6					
⋮	⋮	⋮	⋮	⋮	⋮

STAGE	FUNCTIONAL UNIT SOURCE	VALID	TAG
1		0	
2	INTEGER ADD	1	5
3		0	
4		0	
5	FLT PT ADD	1	4
⋮	⋮	⋮	⋮
N		0	

Figure 2.4: Reorder Buffer, Taken from [4]

2.1.4 Performance Bottleneck

Out-of-order execution realised by register renaming quickly saturates the performance improvement as register renaming becomes one of the hotspots[13] in terms of both power density and power consumption due to the Register Mapping Table(RMT) accesses. Each instruction requires three reads and one write per cycle and the required number of read/write ports is dependent on the number of fetch width. The Register Mapping Table(RMT) being rendered is a complicated multi-ported table in the processor. The multi-ported access slows down the clock frequency since renaming of the next fetch group cannot start unless RMT updates are finished. There have been multiple studies[3][13] in the past years checkpointing this technique and researching implementation of register renaming to reduce the power consumption and latency.

2.2 Sail

Sail is a programming language for describing the instruction-set architecture (ISA) semantics [5]. It provides a faster and simpler development cycle for developing new ISA. Architecture descriptions and emulations[14] are conventionally expressed in a combination of prose and pseudocode as if it is written in a sequential imperative language. It is relatively large with pages of instruction descriptions. Sail, an ISA domain specific language, allows instruction descriptions to be expressed in a compact and readable format for researchers.

Sail, shown in Figure 2.5 was written in OCaml, a functional programming language with an advanced type system for parametric polymorphism and type inference. It allows a carefully designed lightweight type system for checking vector bit length and numeric data types, which generates C and OCaml emulation executable automatically and theorem-prover definitions for Isabelle, HOL4 and Coq, and facilitate integration with RMEM tool for concurrency semantic. Many major instruction-set architectures, such as, RISC-V, ARM, x86 have full sets of definitions in Sail. It is the main tool used when designing the new instruction set architecture. Due to the dependent type system and pseudocode-like language, the tool will reduce the development cycle of emulating the new instruction set.

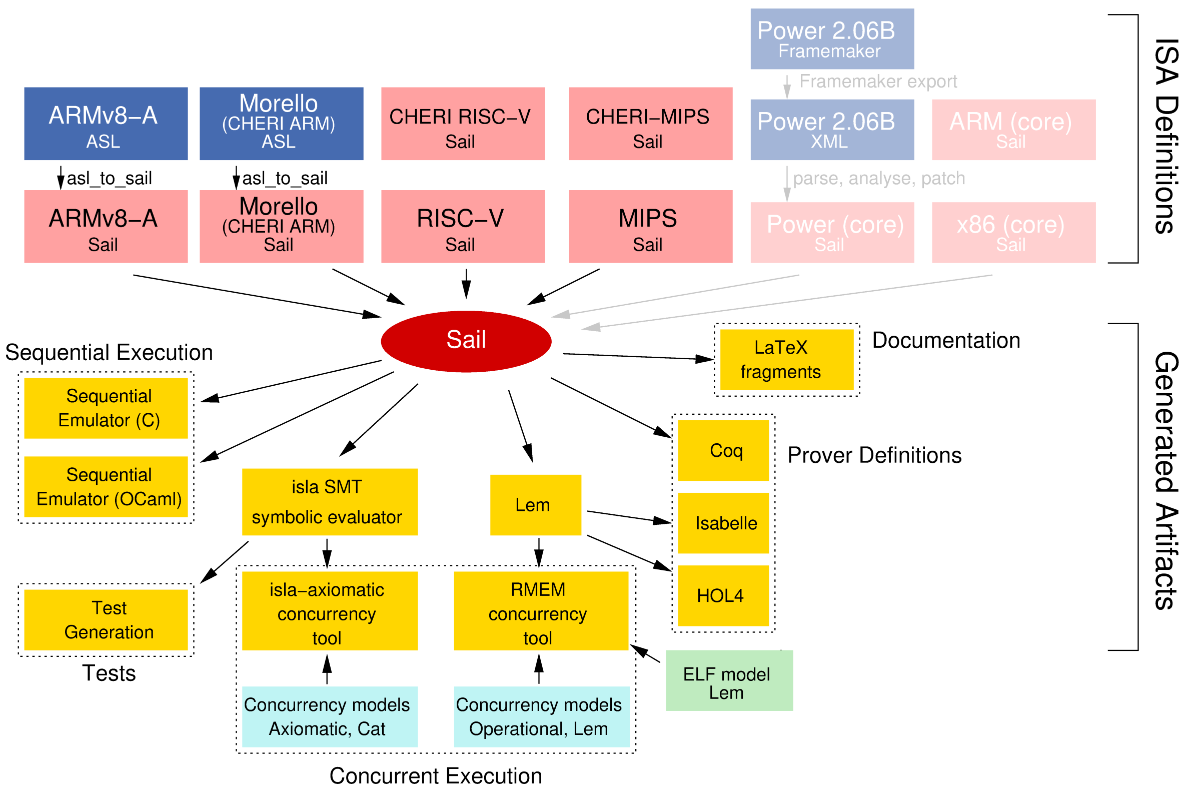


Figure 2.5: Sail Ecosystem Diagram[5]

A Sail specification will typically define an abstract syntax type (AST) of machine instructions, a decode function that takes binary values and decodes to AST values, and an execute function that describes how each instruction was executed at runtime, together with auxiliary functions and types are needed, i.e. update next Program Counter(PC) value.

2.3 LLVM

LLVM[15] is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture. The front end reads the source program in different programming languages, divides it into core parts and checks for lexical, grammar and syntax errors. It generates an Intermediate Representation of the source which is passed to the back end of the compiler to generate machine code of different ISAs.

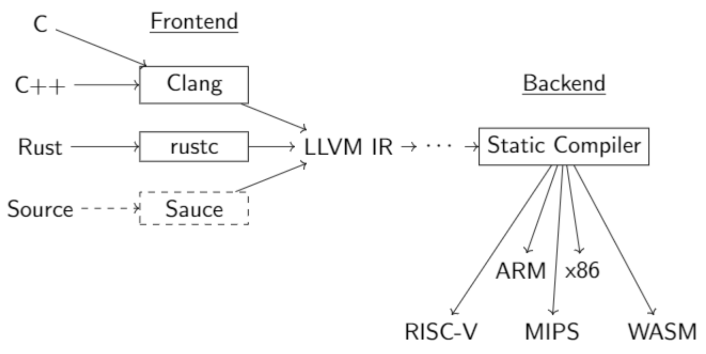


Figure 2.6: LLVM Architecture

2.3.1 Intermediate Representation(IR)

LLVM IR[16] is an intermediate representation, independent of language and machine architecture, similar to assembly and designed to host mid-level analyses and transformations. LLVM IR is generated based on the assumption that a register machine has unlimited number of virtual registers. It adopts Static Single Assignment(SSA) which means each variable is assigned exactly once. When multiple candidates of values are possible by conditional branches, phi-instruction is used to select value according to the control flow.
Listings 4 2 present a compiler front end work, translating source code to LLVM IR. The sum function in C is compiled to LLVM IR with a default -O0 optimisation flag. The alloca instructions on line 2 and 3 reserve space on the stack frame for variable a and b. Since they are both of type int, it reserves a 4-byte alignment. The store instructions on line 3 and 4 store the pointer to the stack element a and b, followed by a load where the two variables are loaded back from the same memory locations. Finally, the result of addition is written to a new variable which is returned by the function.

```c
int sum(int a, int b){
    return a+b;
}
```

Listing 1: Function Sum() in C

This format is similar to the new Instruction Set we are going to design and investigate. The project will explore the possibilities of compilation path from LLVM IR to the new assembly code for compilation flow.

2.4 gem5

gem5[17] simulator is an open-source tool for architectural modelling. It provides a flexible, modular simulation system by offering a wide range of CPU models, system execution modes and memory system models. The Python script initialises, configures and simulates the microprocessor. The output generates a lot of key metrics for performance indicators, such as, total simulation time, number of instructions committed by CPU, number of simulated instructions per second. These should provide a general performance overview of the architecture.

```
define i32 @sum(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

Listing 2: Function Sum() in LLVM IR

2.4.1 ISA

This tool provides support for many commercial ISAs including RISC-V, ARM and MIPS. Evaluation of counterpart architecture can be easily available without much effort. Besides, user-defined ISA domain specific language(DSL) could be plugged into the generic models using the ISA parser to specify the instruction format. The DSL allows the specification of class templates that cover broad categories of instructions. Also it provides a decode tree that combines opcode decoding with specific class creation in the source code. For the new instruction set to be investigated, the new instruction semantics can be defined in the gem5 DSL.

2.4.2 CPU Model

For CPU model, gem5 supports O3 CPU model which is a pipelined, out-of-order model that simulates dependencies between instructions, functional units, memory accesses and pipeline stages. The pipeline stage of the O3 CPU includes fetch, decode, rename, issue, execute, write-back and commit. O3 can simulate a superscalar architecture, which will be an ideal candidate for simulating the new ISA under an out-of-order environment. However, potential concern that arises from the gem5 O3 CPU. As seen in Figure 2.7, the CPU model has default rename stage and reorder-buffer in the pipeline for register renaming.

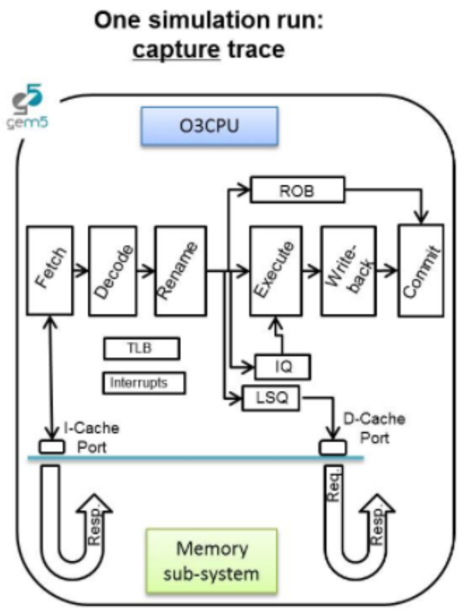


Figure 2.7: gem5 O3 CPU[6]

Either the CPU component needs to be modified or a new CPU component needs to be developed. The abstract class AbstractProcessor in gem5 gives users the flexibility to develop their own components to be compiled into gem5 library. More investigation is needed to confirm whether a processor without register renaming can be developed from the default O3 CPU configurations.

2.4.3 Architecture Simulation

It will be used as a tool for architecture simulation in this project due to the following reasons:

1. It is easy and quick to script and simulate a micro-architecture with different configurations, reducing the time risk at the simulation stage;
2. A large number of ISAs is supported so time could be saved when evaluating the performance of the architecture against a standard RISC architecture;
3. The simulation generates a number of numeric metrics for the performance evaluation.

2.5 Conclusion

After forming the background knowledge of the project, we are going to talk about relevant past literatures that aim to solve the same objectives we mention earlier.

Chapter 3

Related Work

In this chapter, we focus on past literature to remove the performance bottleneck from register renaming and reducing branch misprediction penalty. We also introduce the STRAIGHT architecture[9], a new ISA idea proposed by a research group from University of Tokyo and EDGE architecture, a more scalable ISA idea to statically encode data dependency.

3.1 Register Renaming

As mentioned in Chapter 2, register renaming is one of the major performance bottlenecks in a superscalar out-of-order processor due to the complexity in hardware implementation.

3.1.1 Complexity of Rename Logic

A study[18] shows that a superscalar out-of-order processor with register renaming assuming restricted resources has a complexity of n^k gates and $k^2 logn$ delay, where n is the cardinality of the instruction set(how many instructions are supported in the instruction set) and k is the issue width. This means the hardware complexity increases exponentially with the size of the instruction set and instruction issue width. The delay also increases quadratically as the processor issues more instructions in one cycle. Therefore, performance increase of out-of-order execution quickly saturates as the complexity of the hardware components for register renaming becomes the performance bottleneck.

3.1.2 Rename Logic Optimisation

Some previous studies have attempted to reduce the impact of register renaming on the power consumption and execution performance. One of the attempts[19] tried to implement a two-stage register renaming component, enabling the early termination of register mapping table reads to reduce power consumption. Another approach[20] dealt with caching renamed operands by introducing renamed trace cache(RTC). RTC caches and reuses renamed operands, therefore register renaming can be omitted on RTC hits.

3.2 Misprediction Recovery

In conventional instrucion recovery mechanism, all the instructions younger than the mispredicted branch are removed from the machine and processor is redirected to fetch the correct path. As the modern out-of-order processor pipeline deepens and instruction window becomes larger, the cost of instruction misprediction recovery becomes higher since more in-flight instructions need to be flushed in the pipeline. This has become one of the prominent factors in performance loss and energy consumption. There have been a lot of past literatures discussing ideas for reducing branch misprediction penalty by selective recovery mechanism.

3.2.1 Selective Recovery

This idea is proposed and researched in a few past literatures [21] [22] [7]. Control independence of a branch, shown in Figure 3.1, means the instruction is executed regardless of the outcome of that branch. Data independence of a branch, shown in Figure 3.2, means the source operands are dependent on the same producing instruction regardless of the branch outcome. In Figure 3.2, instruction 7 is data dependent since its data dependence lies on the branch while instruction 8 is data independent. By constructing the control flow diagram either statically or dynamically, control-independent and data-independent instruction of mis-speculated instrucion can be detected and do not need to be executed again in case of instrucion flushing.

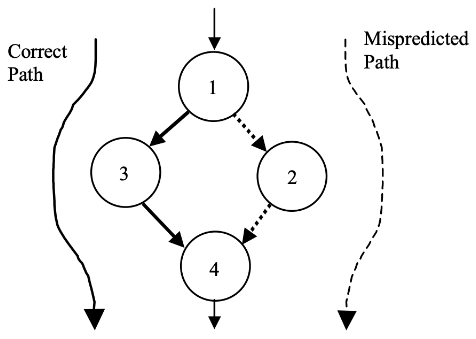


Figure 3.1: Example of Control Independent Block, Taken from [7]

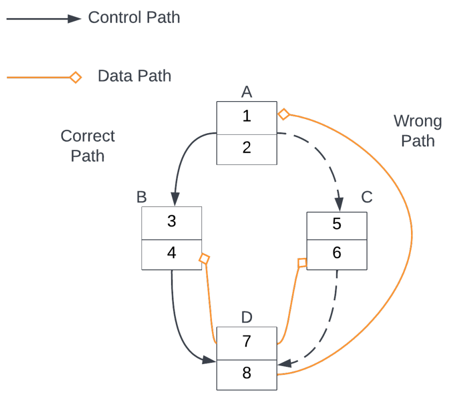


Figure 3.2: Example of Data Independent and Dependent Instruction

Register Integration

Register Integration is one of the implementation techniques proposed to incorporate speculative results directly into a sequential execution using data-dependence relationship. By using the data dependency, it allows squashed instruction reuse, which can be achieved by tracing the squashed path and examining each instruction as they are renamed. If an instruction whose data path is independent of the squashed instruction, the instruction and its results will be re-validated using simple update in the rename table. Register Integration mechanism reduces the wasteful instruction squashing and can reduce the number of instructions executed by up to 15% and improve the performance by up to 11.5%. This mechanism only requires manipulation in the rename table.

3.3 EDGE Architecture

EDGE(Explicit Dataflow Graph Execution) [8] architecture describes a new class of instruction set to direct instruction communication, where hardware delivers a producer instruction's output directly as an input to a consumer instruction, rather than writing it back to register file. In EDGE architecture, a producer with multiple consumers specifies all consumers explicitly in the instruction encoding. The instruction set encodes the physical location of the consumers directly into the bits of each producer instruction. After constructing the control flow graph, the compiler unrolls the loop, predicates basic blocks of instructions and combines single blocks into hyperblcoks. The compiler then generates the code, allocates registers and statically assign each instruction to ALUs. The blocks execute according to Static Placement, Dynamically Issue(SPDI) protocol to increase instruction-level parallelism.

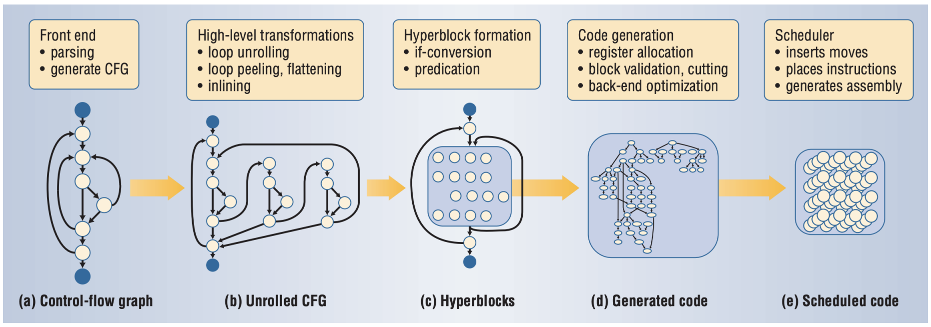


Figure 3.3: EDGE Compiles Program with Control Flow and Schedules Code Statically, Taken [8]

3.3.1 Construction of DFG

Here is an example of how the compiler converts a simple C code in Listing 3 into a Data Flow Graph(DFG) in Figure 3.4, where both variables x and z are dependent on the variable y. The dotted lines in Figure 3.4 show the control dependent instructions. The compiler then optimises them according to their data flow and schedule them statically.

```
    // y, z in registers
    x = y * 2;
    if (x > 7){
        y + = 7;
        z = 5;
    }
    x + = y;
    // x, z are live registers
```

Listing 3: C Code Snippet to Showcase EDGE Architecture

3.3.2 Register Rename Simplification

Statically constructing data flow graph gives another advantage to the architecture since the processor does not need to write temporary values produced and consumed within a block to the register file. It reduces register file accesses and rename table lookups by 70 percent.

3.3.3 Selective Recovery

This architecture exploits the feature of distributed selective re-execution(DSRE) of data misspeculation, where it can scale to window sizes of thounsands of instructions with high performance[23]. It solves the bottleneck that most selective recovery/re-execution scheme in conventional RISC architecture where selective recovery becomes progressively more difficult when instruction window size becomes larger.

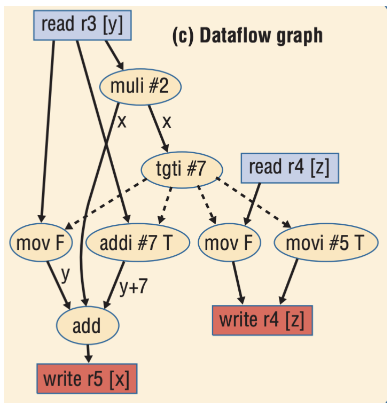


Figure 3.4: Data Flow Graph(DFG) of C Code Snippet in Listing 3, Taken from [8]

In EDGE architecture, when a load mis-speculates, the Data Flow Graph(DFG) sub-tree nodes get incorrect values. According to DSRE protocol, the correct value is sent to the incorrect instruction's node and the instruction is re-executesd sending its new output value to its dependent children nodes. The chilrden, dependent on the mis-speculated instruction, subsequently are re-executed and eventually the entire DFG subtree dependent on the faulting instruction in re-executed. DSRE enables multiple waves of speculative execution to traverse the dataflow graph simultaneously, enabling more scalable selective re-execution. It is reported that DSRE protocol can speedup the best dependence predictor by an average of 17%.

3.4 STRAIGHT Architecture

STRAIGHT [9][24][10] is a new architecture idea proposed by a research group in University of Tokyo to skip the performance bottleneck of register renaming and gain performance advantage from rapid miss-recovery. The new ISA presented a feature that each logical register is written once and only once before being discarded in a fixed period, being made possible by assuming large space of logical registers. Each instruction is mapped to a fixed register number, storing the result value of the instruction into the register. Register numbers of the source operands are indicated by calculating the distance from the instruction that produces the value. The fetched instructions can be directly dispatched to the scheduler. The register file was built as a simple key-value store that is easy to scale. This technique characterises the data flow of programs at runtime, therefore eliminating data hazard and the need for register renaming. Logical values were wrapped around, meaning there is a fixed distance to which each instruction could access.

3.4.1 Instruction Set Architecture(ISA)

The STRAIGHT instruction set shown in Figure 4.2 was based on a RISC ISA and designed by modifying the MIPS instruction set. Destination register field, which becomes unnecessary, was removed to accommodate for the increase in bit length for source operands due to an increase in register file size. 10 bits were allocated to index the register, therefore allowing a total register file size of 1024. This architecture was able to execute a wide range of operations to achieve the full functionality as RISCV32IM.

For example, to compute an integer addition, the following instructions would be needed. Instruction 1 and 2 load integer 1 and 2 respectively into logical register 1 and 2, with respect to instruction 1 and 2. The next instruction performs an addition with source operands taken from results of instructions with distance of 1 and 2 away from current one. In this case, it is an addition of logical registers 1 and 2 from instruction 1 and 2. In instruction 4, source operands with a distance of 1 and 2 instructions are logical register 2 and 3 respectively. It, therefore, performs an addition of integer stored in register 2 and 3, which gives a result of 5.

Below presents a few special features both in the instruction set and compiler technology unique that gives the new ISA full functionality.

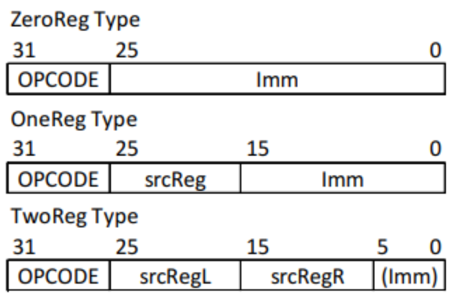


Figure 3.5: STRAIGHT Instruction Set, Taken from [9]

```
    ADDI $ZERO, 1 # 1
    ADDI $ZERO, 2 # 2
    ADD [1][2]    # 3
    ADD [1][2]    # 5
```

Listing 4: Instruction Code

Register Pointer(RP)

Register Pointer(RP) is a special register introduced to the new architecture. It acts as a translator to provide the destination register number in each instruction. It is incremented rigidly by one for every instruction to keep record of where the destination register of the current instruction is at and therefore the source registers can be calculated by subtracting the distance from the RP. *MAX_ RP* determines the number of physical registers available and RP returns to 0 when it exceeds the *MAX_ RP*, forming a physical register wrap-around.

Stack Pointer(SP) & SPADD Instruction

Stack pointer(SP) is another special register introduced to the new architecture. *SPADD* instruction is a SP-related operation which reads and modifies the SP by adding the given immediate value and writing the result to the respective destination register. The succeeding load and store instructions can use the SP value by indicating the distance from the *SPADD* instruction. *SPADD* is used for more complicated control flows such as function calls and generated at the entrance and exit of the function.

3.4.2 Compilation Flow Part 1 – Calling Convention

Since there is no fixed register, STRAIGHT architecture stores arguments and returns values in registers. Instructions that generate arguments and return values are arranged in a fixed order and to place the producer of argument prior to jump-and-link(*JAL*) instruction. By following a fixed calling convention, the distances between an instruction in the callee to producers of the argument are fixed. As shown in the following Figure 3.6, *ADD [3] [4]* takes source operands at a distance of 3 and 4 instructions away respectively. Therefore, it will always refer to arg0 and arg1. The same logic applies to return values as well. In instruction *ADDI [3] 1*, source operand is referenced to producer of retval0 3 instructions above the addi instruction.
At the instruction, *JAL #callee*, the destination register is $PC + 4$. At the exit of the function call, return address is passed to the callee by the *JAL* instruction that writes its $PC + 4$ to its destination register.

3.4.3 Compilation Flow Part 2 – Code Generation

The code generation mainly deals with translating from LLVM intermediate representation stage to assembly code, which is a LLVM back-end compiler.The compiler has 3 stages – operation translation, distance fixing and distance bounding.

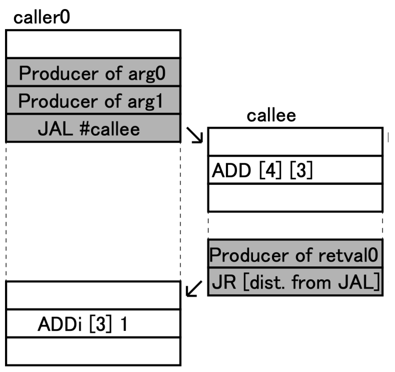


Figure 3.6: Function Call, Taken from [9]

Operation translation

Operation translation is equivalent to LLVM back-end in a conventional RISC architecture, translating IR to assembly code.

Distance fixing

Distance fixing is the step that the compiler calculates and adjusts distance from a consumer instruction to producer instructions to ensure compatibility with STRAIGHT instruction set in case of merging control blocks. When an operand has multiple producer candidates or there are multiple paths from producer instruction to its consumer, liveness analysis in LLVM backend can capture such information. Register move, *RMOV*, instructions are added to the end of merging basic blocks so to fix the distance regardless of the control flow. Figure 3.7 illustrates how register move instruction would help fix distance between producer and consumer instruction in case of merging block.

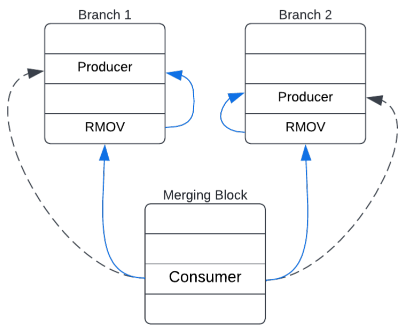


Figure 3.7: Distance Fixing On Merging Block, Taken From [9]

Figure 3.8 shows an example implementation of simple loop in STRAIGHT. *RMOV* instructions need to be inserted right before the conditional jump to make sure the producers' positions of arguments 1 and 2 are fixed and known to the loop body.

Distance bounding Distance bounding is an additional step to ensure register encoding is bounded by the maximum distance of source registers. *RMOV* instruction will be added to relay the values when any distance exceeds the maximum.

3.4.4 Performance Evaluation

A softcore processor[10] in FGPA is developed to simulate the performance of the new architecture. The new architecture is evaluated against an OoO RISC-V counterpart with almost identical configurations in cache, re-order buffer, fetch and commit width.

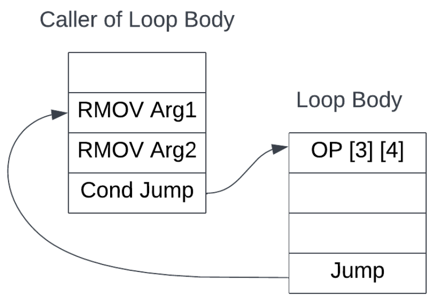


Figure 3.8: Distance fixing on a Loop Implementation

Compared to a OoO RISC-V soft processor, the STRAIGHT softcore processor consumes approximately 17% fewer LUTs and 10% fewer FlipFlops and achieves 15% higher performance in CoreMark.

3.5 Conclusion

This project will use the features from above in STRAIGHT architecture as the main inspiration for a new instruction set to study the effect of removing register rename logic on instruction recovery penalty and overall processor performance.

Chapter 4

Design

This chapter will discuss the main design concepts on designing a new Instruction Set Architecture(ISA), compilation flow, assembly and linking, and architecture modelling. It introduces the high-level abstract ideas of the new ISA. The discussion will be focused on design choices and how it could achieve the goal of improving performance and simplifying architectural design.

4.1 Aim

The aim of the design flow is to find the reliable and open-source tools available to modify with respect to the new STRAIGHT architecture with high accuracy and agility within a short development cycle to be able to make evaluation and informed decision of the effectiveness of the new architecture. A lot of ideas were taken from original STRAIGHT[9] paper. However, one of the drawbacks of the old STRAIGHT architecture was that they didn't have one-to-one mapping of instructions between conventional RISC instruction set and the new ISA. To create a common ground for architecture simulation comparison with well-established, the approach of the project was to build on the basis of a well-established instruction set and modify the architecture to fulfill the requirements of the new STRAIGHT architecture. Readily available toolchains and architecture models would be used for modification so the new architecture could be modelled with a good level of confidence.

4.2 Instruction Set Design

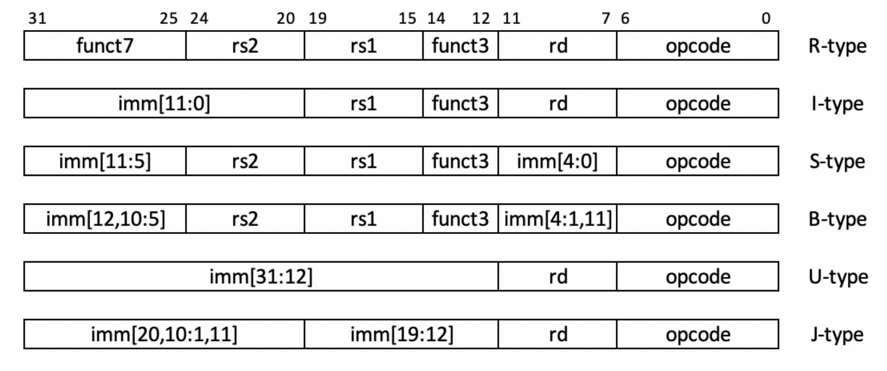
RISC-V is an open standard instruction set architecture(ISA) based on established RISC principles. In the new instruction set design, generic opcode, funct3/funct7 and immediate fields remain unchanged. Since destination operands are mapped to a fixed position in the register file, the destination operand fields would be *don't cares(xxx)*.
Furthermore, to facilitate a longer bit length for larger register file, the instruction length has been extended from 32 bits to 64 bits. The new modified instruction set has a 10-bit field for each source operand. The original source operand field – 5 bits used for the lower half bits of the new source operand. The upper half bits of the source operand will be taken in the extended 32 bit field. This design maximises the similarity of the new ISA to the original RISC-V ISA, which paves the way for the accuracy and convenience of architecture modelling.
There is also bit field left as *don't cares(xxx)*. It can be used for future investigation of this instruction set. For example, it gives more flexibility over extending the size of the register file and studying the effects on the architecture performance of a larger register file.

4.2.1 Major Differences From STRAIGHT Instruction Set

In STRAIGHT instruction, each instruction is 32 bits by squeezing out opcode field for a wider source operand field width. In the new instruction set, compactness machine code length is sacrificed for more agile development of the architecture modelling.

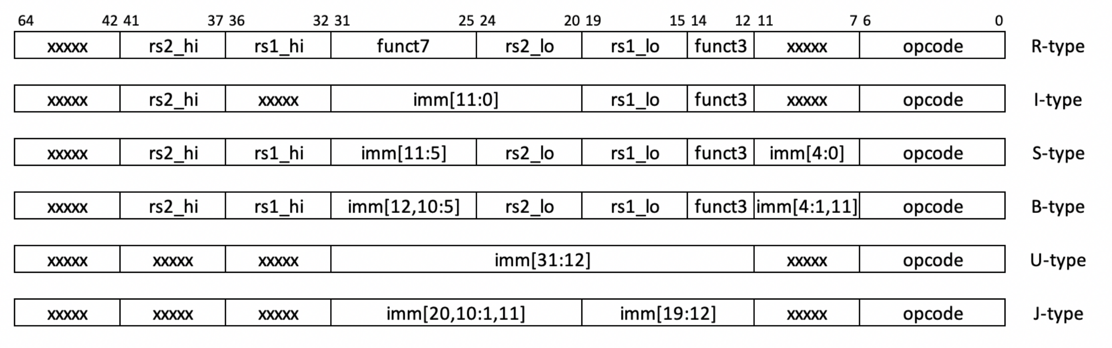
4.2.2 Modification on RISCV ISA

As shown in Figure 4.1 and 4.2 below, here is a side by side comparison between RISC-V Instruction Set and new ISA designed for STRAIGHT architecture.



31 25 24 20 19 15 14 12 11 7 6 0
funct7 rs2 rs1 funct3 rd opcode R-type
imm[11:0] rs1 funct3 rd opcode I-type
imm[11:5] rs2 rs1 funct3 imm[4:0] opcode S-type
imm[12,10:5] rs2 rs1 funct3 imm[4:1,11] opcode B-type
imm[31:12] rd opcode U-type
imm[20,10:1,11] imm[19:12] rd opcode J-type

Figure 4.1: RISC-V Instruction Set Design



64 42 41 37 36 32 31 25 24 20 19 15 14 12 11 7 6 0
xxxxx rs2_hi rs1_hi funct7 rs2_lo rs1_lo funct3 xxxxx opcode R-type
xxxxx rs2_hi xxxxx imm[11:0] rs1_lo funct3 xxxxx opcode I-type
xxxxx rs2_hi rs1_hi imm[11:5] rs2_lo rs1_lo funct3 imm[4:0] opcode S-type
xxxxx rs2_hi rs1_hi imm[12,10:5] rs2_lo rs1_lo funct3 imm[4:1,11] opcode B-type
xxxxx xxxxx xxxxx imm[31:12] xxxxx opcode U-type
xxxxx xxxxx xxxxx imm[20,10:1,11] imm[19:12] xxxxx opcode J-type

Figure 4.2: New STRAIGHT Instruction Set Design

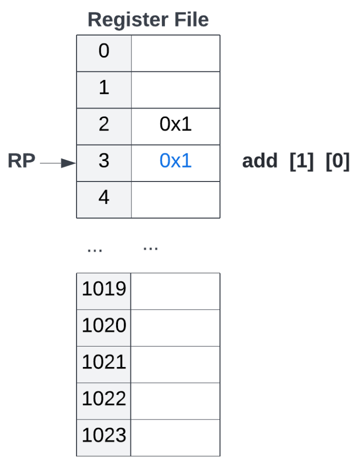
4.2.3 Special Registers

Since the new STRAIGHT architecture made all registers general purpose, a few more special registers and instructions need to be added to the conventional RISC-V instruction set.

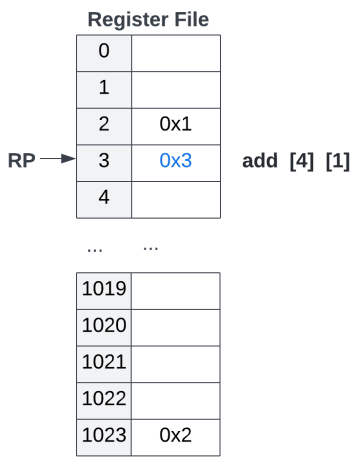
Register Pointer(RP)

Register pointer is the most important special register for the STRAIGHT architecture. It replaces the destination operand and increments every time an instruction is executed to indicate at which position the result of the instruction is written to in the physical register file. In addition, source operand field encodes the distance of the source registers from the register pointer. The source register index can be translated from RP minus distance from source operand. Register pointer should be a rigidly incrementing register regardless of the type of instruction being executed.
The maximum value of register pointer should be the size of register file, $2^{\text{no-of-bitfield}}$. Therefore, once the register pointer value reaches its maximum value 1024 since the width of the bitfield is 10, register pointer has a wrap-around effect and goes back to zero. The maximum distance of source operand that can be reached by this indexing notation is $size\,of\,register\,file - 1 = 1023$.
Since all registers in the register file serve as general purpose, zero register is defined when the source operand field is 0, meaning at a zero distance away from register pointer when the source operand is indexed at its RP.

Example As shown in the Figure 4.3, two register pointer examples are illustrated with the aid of register file visualisation. In Figure 4.3a, the register pointer is at 3 and instruction is *add [1] [0]*. It is an R-type instruction with 2 source operands. Index 1 indicates that the first source operand is 1 register away from register pointer, which translates to $RP - 1 = 2$ and takes the value of 1 stored in physical register #2. Index 0 indicates that the second source operand is zero register. Therefore, the result of this instruction is 1 and written to physical register #3. The Figure 4.3b illustrated how translation of physical register of source operands is calculated under register pointer underflow. Index 4 indicates that the first source operand is 4 registers away from the current register pointer which is pointing at 3 in the register file. The underflow in source operand index makes use of the wraparound behaviour in register pointer. Therefore, the second source operand is indexed at the last physical register #1023 in the register file.



(a) RP example with no underflow



(b) RP example with underflow

Figure 4.3: Example of Register Pointer

Stack Pointer(SP)

Another special register to add is Stack Pointer(SP). SP is an overwritable register to facilitate complicated algorithms such as function calls and memory operations. Live variables will be stored in the stack frame which is indexed by SP. A few SP-related instructions will be added to facilitate the use of the stack frame, which will be discussed in the Section below.

4.2.4 New Instructions

RMOV

Register move instruction copies the source register value into its destination register. Not only can it arrange the order of the produced values, but also can be added at the end of merging basic blocks to achieve the purpose of distance fixing. Furthermore, *RMOV* can also be used for relaying values when the maximum distance between consumer and producer exceeds the maximum distance allowed defined by max_rp.

SPADD

SPADD instruction is used to move the stack frame pointer by a certain offset. *SPADD* is a UJ-type instruction which takes an intermediate field as the offset to increment or decrement the SP pointer. *SPADD* also writes the final SP value into its RP register. *SPADD* instruction is called at the entrance and exit of the function call so to allocate and free stack frame for the function call.

SPLD and SPST

SPLD and *SPST* instructions are used to access the stack frame. *SPLD* is used to write local variables and function arguments to stack frame and *SPST* is used to used to read from stack frame local variables. The two instructions can also be replaced by *LD* and *ST* instructions given SP value is stored in the register file nearby.

4.3 Compilation

The compilation process is similar to what has been done by STRAIGHT architecture with LLVM frontend emitting Intermediate Representation(IR) and further developed backend to emit assembly code with distance fixing for source operands. of the design concepts are described in Chapter 3. Some differences in implementation details due to change in machine code length will be discussed in Chapter 5.

4.4 Assembly

Since the new STRAIGHT instruction encoding is extended on top of RISC-V encoding. All the opcode and funct code remain the same. The destination operand field is assembled as all zeros. The source operand index is assembled into 10-bit unsiged integer to fulfill the *max_rp* value of 1024. The 10 bit operand field is then split into upper bits and lower bits, where lower bits written to the [19:15] bits of the instruction and upper bits written to [36:32] bits for source operand 1. Similarly, they are written to [24:20] bits and [41:37] bits for source operand 2. The implementation details of the assembler will be talked in Chapter 5.

4.5 Architecture Modelling

4.5.1 Instruction Set Semantics Definition – Proof of Concept

Instruction set semantics can first be defined and type checked in SAIL as an early stage proof of concept. An executable emulator will be generated in OCaml by SAIL backend, which can be used to run handcrafted binaries to verify the instruction set definition.

4.5.2 Single Cycle Processor – Accuracy Modelling

Single cycle processor is similar to an instruction set emulator. The first step is to make sure the design concept is correct with instruction decoding and behaviour emulation correctly modelled in the architecture modeller. The new ISA has 6 types of instructions, each differing slightly in function and operand encoding.

R-type Instruction R-type instructions consist of two source operands, each taking 10 bit for register number decoding. The funct3 and funct7 codes determine which operations to execute on two source operands and write to the destination operand, indexed by RP.

I-type Instruction I-type instructions consist of only one 10-bit source operand and an immediate value field. The funct3 determines what operations to execute on the source operand and immediate value and write to the destination operand.

B-type Instruction B-type instructions consist of two 10-bit source operands and an immediate value field. It is a branch instruction where the decision on whether the branch is taken or not depends on the funct3 and two source operands. If the branch is taken, the next program counter value will be overwritten to program counter offset by the immediate value. It is normally used in if-else statement and loop exit conditions.

S-type Instruction S-type instructions consist of two 10-bit source operands and an immediate value field. S-type instructions are only store-related instructions, including *sw*(store word), *sb*(store bytes), etc. Funct3 determines the length of data stored. Store instruction stores source operand #2 into the address obtained by address stored in source operand #1 offset by the immediate value.

U-type Instruction U-type instructions consist of only 20-bit immediate field. The offset uses the immediate field as the upper 20 bits and loads the rest 12 bits with zeros. The offset is either loaded into the destination operand or added to the current PC value, depending on the opcode.

J-type Instruction J-type instructions consist of only 20-bit immediate field and only *jump-and-link* instruction follows this encoding. The immediate value encodes the signed offset in multiples of 2 bytes. For example, jumping PC forward by 4 bytes is translated to *jal 2*. The jump stores the address following the program counter of JAL instruction as destination operand, which means the destination register would store $PC + instruction\_width$.

Environment Call and Breakpoints *ecall* and *ebreak* belong to I-type instructions. However, since these two instructions are system calls and require simulation of operating system behaviour. In RISC-V, argument for system call is pre-defined stored in register #17. In the new instruction set, arguments of the system calls are passed from the previous destination register, $RP - 1$.

4.5.3 Out-of-Order Processor – Performance Modelling

For performance modelling of the new architecture, out-of-order processor pipeline would be modified to accommodate for changes in register renaming and instruction retirement/recovery. Figure 4.4 illustrates the conventional RISC out-of-order processor modelled in gem5. It is loosely based on the modern processor Alpha 21264[25]. Figure 4.5 shows the pipeline of the modelled new architecture in gem5 where lines in orange indicate the differences from conventional RISC processor.

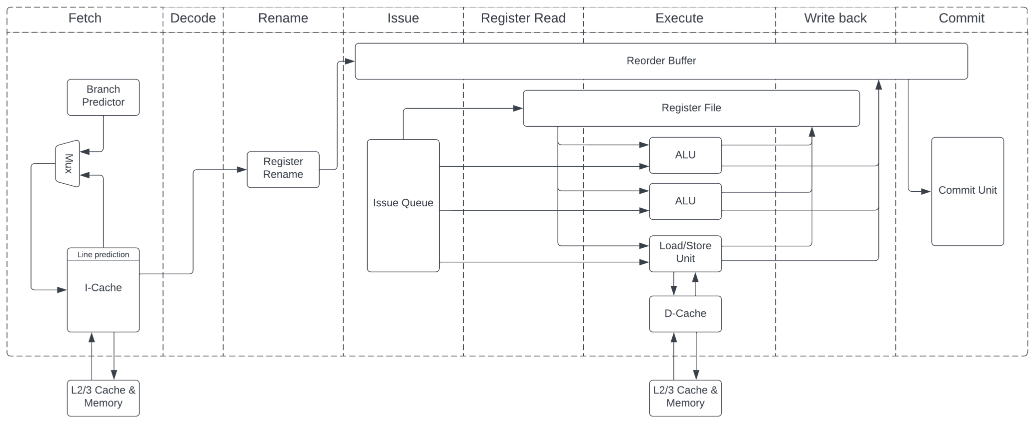


Figure 4.4: Out-of-Order Processor in gem5 loosely based on Alpha 21264

Register Renaming

The new architecture pipeline, shown in Figure 4.5 gets rid of the register renaming stage. The grayed out area means the latency is not counted even thought rename stage is not removed in gem5 but given a fixed one-to-one mapping. Instead, special register processing is required at decode stage of the pipeline. Register pointer is processed at the decode stage so the physical register number of source operands and destination operand can be translated/calculated at the decode stage of the pipeline. The pipeline would no longer need register renaming stage. Therefore after decoding the instruction and translating the register number, the instruction is pipelined straight to issue queue waiting to be scheduled for execution.

Instruction Commit/Recovery

Another difference in the new architecture is instruction retirement and recovery in case of branch misprediction. Since instructions are dispatched in ROB in order of ascending RP number, ROB

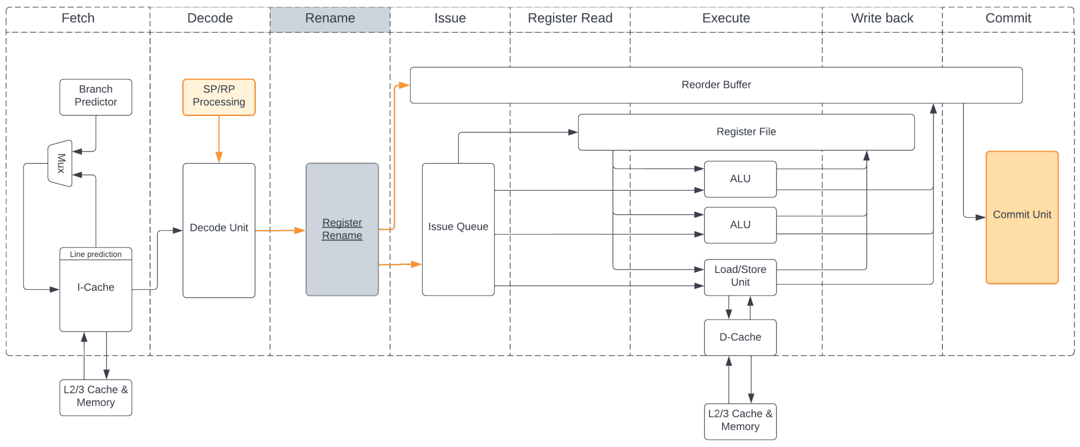


Figure 4.5: Out-of-Order Processor Modified for New Architecture

does not need to maintain RP for each instruction. Only RP at the head of reorder buffer needs to be tracked and relevant RP can be calculated from offset of instruction positions in the reorder buffer. One extra thing that needs to be kept in the ROB is SP value. A SP table is maintained storing all SP values encountered within the latest 1024 instructions executed. Each instruction in ROB will have an extra field of SP pointer, pointing at the correct location storing SP value in the SP table.
When the processor encounters a branch misprediction, the head pointer of the ROB, which is pointing at the oldest entry of the ROB, is accessed to obtain the SP and PC value. The RP the head pointer is associated is also used for RP restoration. Tail pointer is then moved to the head pointer so that when the processor restarts fetching, issuing and dispatching the correct successor of the branch into the pipeline. Instructions will be correctly written to the right entry of the ROB. Since there is no register renaming, all the architectural registers are mapped to a physical register at decode time. Side effects in register file from branch misprediction do not need to be flushed and they will be overwritten by the correct set of instructions eventually. This is one of the major performance benefits from mapping out control flow statically at compilation stage.

4.6 Conclusion

We discussed most of the design concepts for the new STRAIGHT architecture from compilation to architecture modelling. The next chapter will present implementation details of ISA emulator, assembler for new STRAIGHT instruction set and modelling in gem5, a well-established modular platform for computer-system architecture research.

Chapter 5

Implementation – ISA Definition and Compilation Support

This chapter will discuss the implementation details of defining new instruction set in SAIL, an ISA emulator. We will also discuss how to implement the static linker and assembler based on the design from Chapter 4.

5.1 Instruction Set Modelling in SAIL

As an early stage exploration, ISA is defined in SAIL. We also define one instruction of different types – R-, I-, B-, and U-type, to investigate the feasibility of the architecture.

Special Registers

Special registers need to be defined in SAIL first to differentiate from general-purpose registers. We define program counter(PC), next program counter(nextPC) and register pointer(RP).

PC PC tracks the current address of the instruction being executed. After each instruction gets fetched, the program counter is assigned by nextPC value.

NextPC NextPC tracks the address of the instruction to be executed next. After each instruction gets fetched, next program counter value increments by 8 since the instruction is 8 bytes long. In case of a jump or branch instruction, next program counter will be overwritten by the address that the current instruction wants to jump to.

RP RP tracks the index of the destination operand in the register file. It points at the place where the result of the current instruction should write to. After each instruction gets executed, it gets incremented by 1 pointing at the next position of the register file.

```
(* special regs declarations *)
register PC : xlenbits
register nextPC : xlenbits
register RP: regbits
```

Listing 5: Definition of Special Registers in SAIL

Source Operand Translation

Before reading the value from the register file, the source operand index needs to be translated with respect with RP. The translation algorithm was defined in SAIL, as shown in Listing 6. When the source operand is 0, it means it is reading from a zero register, a special feature in the new architecture. Otherwise, the processor needs to check whether there is an underflow in the offset.

Wraparound behaviour is exhibited in case of register underflow. *Max_rp* is a constant storing the size of the register file so the index of the register can be correctly translated under register underflow.

```
function check_underflow(regpointer, operand) = {
  let max_rp = 1024 in
  let result = if unsigned(regpointer) < unsigned(operand) then (regpointer + max_rp - operand) else (regpointer - operand) in
  result
}

function translate_reg(regpointer, operand) = {
  let result = if unsigned(operand) == 0 then zero_reg else X(check_underflow(regpointer, operand)) in
  result
}
```

Listing 6: Definition of translate_reg Function in SAIL

Instruction Semantics Definition

Instructions are first decoded into their respective types according to their opcode. The source operand fields are then extracted and concatenated from lower 5 bits and upper 5 bits because of the special source operand encoding layout in the new instruction set. As illustrated in Listing 7, R-type instructions has 2 source operands. They are being decoded and concatenated in the decode function to give source operand indices.

R-type In R-type instructions, each individual instruction is further assigned an op name, *op* according to the funct3 field. For example, all zeros, *000*, represent add instruction. At execution stage, 2 source operands values can be obtained by reading from the register file using the relative distance from RP. Afterwards, R-type instructions execute on the two source operands and the result is written to the location in the register file indexed by register pointer.

```
union clause ast = RTYPE : (regbits, regbits, rop)
function clause decode zeros12 : bits(22) @ rs2_upper: halfregbits @ rs1_upper: halfregbits @ 0b0000000 @ rs2_lower : halfregbits @ rs1_lower : halfregbits @ 0b000 @ zeros5 : bits(5) @ 0b0110011
    = Some(RTYPE(rs2_upper @ rs2_lower, rs1_upper @ rs1_lower, STRAIGHT_ADD))
function clause execute (RTYPE(rs2, rs1, op)) = {
    let rs1_val = translate_reg(RP, rs1);
    let rs2_val = translate_reg(RP, rs2);
    let result : xlenbits = match op {
        STRAIGHT_ADD  => rs1_val + rs2_val,
        STRAIGHT_SUB  => rs1_val - rs2_val
    };
    X(RP) = result;
}
```

Listing 7: Definition of R-type Instructions in SAIL

I-type I-type instructions are similar to R-type. There is a 12 bit immediate field, which is sign extended at execution stage. Only 1 source operand is available in the instruction. After reading from the source register and decoding the immediate value, they are executed according to their op names. The result is written to the register file indexed by RP.

```
union clause ast = ITYPE : (bits(12), regbits, iop)

function clause decode zeros17 : bits(27) @ rs1_upper: halfregbits @  imm :
↪ bits(12) @ rs1_lower : halfregbits @ 0b000 @ zeros5 : bits(5) @ 0b0010011
  = Some(ITYPE(imm, rs1_upper @ rs1_lower, STRAIGHT_ADDI))

function clause execute (ITYPE (imm, rs1, STRAIGHT_ADDI)) = {
  let rs1_val = translate_reg(RP, rs1) in
  let imm_ext : xlenbits = EXTS(imm) in
  let result = rs1_val + imm_ext in
  X(RP) = result
}
```

Listing 8: Definition of I-type Instructions in SAIL

B-type B-type instructions are branch instructions. It takes 2 source operands and a 12-bit immediate field. The condition on whether to branch on not depends on the 2 source operands and the op name of the instruction. The immediate field will be rearranged according to the instruction set design mentioned in the Figure 4.2, left shifted and signed extended as an offset for the branch instruction. If the branch is taken, the next pc value will be overwritten to $PC + offset$. If not taken, PC and nextPC will be updated as usual.

```
union clause ast = BTYPE : (bits(13), regbits, regbits, bop)

mapping encdec_bop : bop <-> bits(3) = {
  STRAIGHT_BEQ  <-> 0b000,
  STRAIGHT_BNE  <-> 0b001
}

function clause decode zeros12 : bits(22) @ rs2_upper: halfregbits @
↪ rs1_upper: halfregbits @ imm7_6 : bits(1) @ imm7_5_0 : bits(6) @
↪ rs2_lower : halfregbits @ rs1_lower : halfregbits @ funct: bits(3) @
↪ imm5_4_1 : bits(4) @ imm5_0 : bits(1) @ 0b1100011
  = Some(BTYPE(imm7_6 @ imm5_0 @ imm7_5_0 @ imm5_4_1 @ 0b0, rs2_upper @
  ↪ rs2_lower, rs1_upper @ rs1_lower, encdec_bop(funct)))


function clause execute (BTYPE(imm, rs2, rs1, op)) = {
  let rs1_val = translate_reg(RP, rs1);
  let rs2_val = translate_reg(RP, rs2);
  let taken : bool = match op {
    STRAIGHT_BEQ  => rs1_val == rs2_val,
    STRAIGHT_BNE  => rs1_val != rs2_val
  };
  let t : xlenbits = PC + EXTS(imm);
  if taken then {
    (* Extensions get the first checks on the prospective target address. *)
  set_next_pc(t);
  }
}
```

Listing 9: Definition of B-type Instructions in SAIL

J-type J-type instructions are direct jump instructions. It takes a 2 20-bit immediate field, left shifted by 1 bit and signed extended as the jump offset. The next pc value is overwritten by pc with the jump offset.

```
union clause ast = STRAIGHT_JAL : (bits(22))

function clause decode zeros32 : bits(32) @ imm_19 : bits(1) @ imm_18_13 :
↪  bits(6) @ imm_12_9 : bits(4) @ imm_8 : bits(1) @ imm_7_0 : bits(8) @
↪  zeros5 : bits(5) @ 0b1101111
  = Some(STRAIGHT_JAL(imm_19 @ imm_7_0 @ imm_8 @ imm_18_13 @ imm_12_9 @
  ↪  0b00))

function clause execute (STRAIGHT_JAL(imm)) = {
  let t : xlenbits = PC + EXTS(imm);
  (* Extensions get the first checks on the prospective target address. *)
  X(RP) = get_next_pc();
  set_next_pc(t);
}
```

Listing 10: Definition of J-type Instructions in SAIL

5.2 Compiler and Linker

Clang compiler is used to compile C source code to LLVM IR in the front end pipeline targeting RISCV64 architecture with the compiler flag *–target=riscv64-pc_ linux-gnu*. The LLVM IR is then passed to the LLVM compiler backend to generate intermediate assembly code, which is then passed to a linker to remove global objects, section labels, link C standard library and generate STRAIGHT assembly code.

5.3 New Instruction Set Assembler

An instruction set assembler is implemented and modified from an exisitng RISC-V assembler[26] in Python to extend the source operand field, remove the destination operand field and account for the differences between RISC-V and the new instruction set. After the generating binary code, an executable file is generated from statically inserting section labels and setting start address.

5.3.1 Source Operand

All alias for registers are removed. Instead of mapping registers from its alias to actual register index, a register converter is added to parse the register in assembly and translate to bitcode format. Since source operands have been extended from 5 bits to 10 bits. The translated bitcode is broken into upper bits and lower bits, with the lower bits in the original source operand place and the upper bits in the extended instruction field, as shown in the Figure 4.2.

5.3.2 Destination Operand

Since all destination operands are removed in the new instruction set, destination operands are encoded to all zeros in R-, I-, U-, J-type instructions.

5.3.3 New Instructions

RMOV

RMOV instruction is implemented as an alias instruction of *addi [source] 0*, since there is no side effect of moving the destination register to the new register pointer.

NOP

NOP instruction is implemented as an alias instruction of *addi [0] 0*. The same encoding is also used in RISC-V conventions. The instruction does not change any user-visible state, except for advancing the program counter. It allows microarchitectural optimizations as well as for more readable disassembly output[27]. An extra advantage in the new instruction set is the instruction can be used to pad the distance for fixing distances during control flow merging blocks.

5.4 Conclusion

In this chapter, the discussion is mainly focused on compiling source code to STRAIGHT assembly code and encoding them in the new instruction set and generating executable file for architecture modelling. The next chapter will talk about the implementation details of architecture modelling in gem5 and how we modify the existing microarchitecture simulator to model the new architecture.

Chapter 6

Implementation – Architecture Modelling in gem5

After implementing the instruction set emulator and providng linking and assembly support, the front end pipeline is complete. The next step is to model the new architecture in gem5 to evaluate the performance.

6.1 Architecture Definition

Since gem5 has an isolation between different microarchitecture components, such as processor, memory protocol and ISA, the first step of implementation is to define the special instruction set in gem5. We use the domain specific ISA description language in gem5 to define the new ISA. Since it already has a full set of definition of RISC-V, the implementation differences will be discussed in the section below.

6.1.1 Operand Definition

In the operand definition file, operand *RP* is defined as an integer register, *IntReg* to replace operand *RD*, which is used as destination register in RISC-V ISA definition.

6.1.2 Bitfield Definition

In the bitfield definition, the bitfield length is extended from <31:0> to <64:0> to represent the increase in instruction length. Source operand 1 bitfield is modified from <19:15> to <36:32,19:15>. Source operand 2 bitfield is extended to 1o bits to <41:37, 24:20> similarly. Another change in bitfield is to remove destination operand bitfield.

6.1.3 Decoder Definition

In decoder file, all the instruction definitions follow RISC-V definitions closely except for the types that write result to destination register. In this case, they write to register pointer, which is not translated from bitfield definition.

6.1.4 RP Modelling

RP is a special register that needs an extra class container to store its state with other functions to access, advance and overwrite its value. Class *RPState* has one member variable of type *RegIndex*, a special type for logical register index. To advance RP, the rp will be incremented by 1. RP can also be manually set to a special value by a pointer to another *RPState* class.

6.2 Single Cycle Modelling – Accuracy Modelling

In the simple single cycle CPU model, one instruction is fetched, decoded and executed in one clock tick. The aim of the simple cycle CPU modelling is to model the new microarchitecture and verify the correctness what has been implemented in architecture definition.

6.2.1 Simple CPU Model

As shown in the Figure 6.1, the diagram is an incomplete UML diagram, illustrating how atomic single cycle cpu is implemented and modified to work with the new architecture. *BaseSimpleCPU* class is an abstract class storing all threads information. It also stores the information of the current active thread and static instruction being executed in the simulator. It sets up the basic function such as *fetch()*, *preExecute()* and *postExecute()*. *AtomicSimpleCPU* class inherits abstract *BaseSimpleCPU* class and only has a member variable of type *EventFunctionWrapper* called *tickEvent*, which is used to wrap the *tick()* function into an event to be executed in cycles.
Each *SimpleExecContext* class stores the information of one simple thread. *BaseSimpleCPU* stores a vector of *SimpleExecContext* class to represent all threads under simulation. Each *SimpleExecContext* class has a composition class of *SimpleThread* which stores pointer to PCState and RPState class. It also maintains the information of TLB, ISA and decoder. The member function *readIntReg()* and *setIntReg()* will be used for reading from and writing to registers when executing instructions.

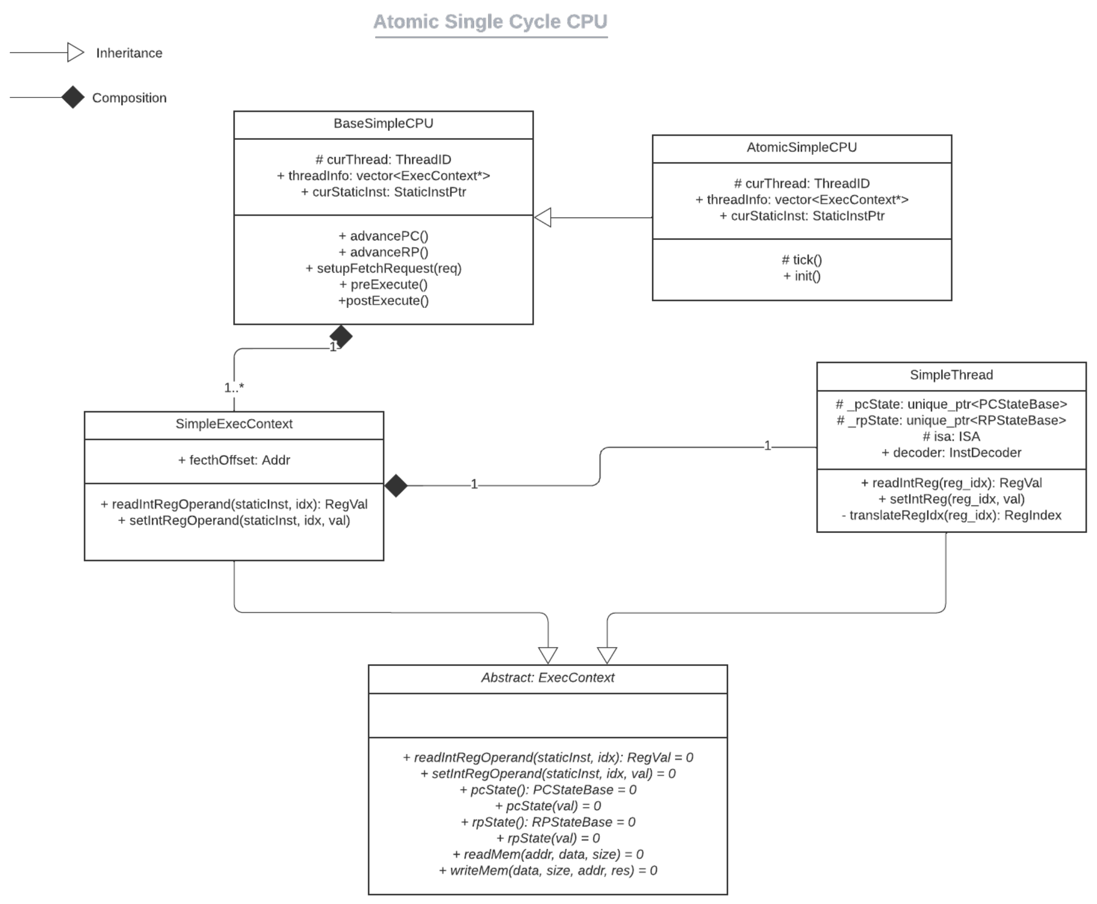

Atomic Single Cycle CPU
Inheritance
Composition
BaseSimpleCPU
# curThread: ThreadID
+ threadInfo: vector<ExecContext*>
+ curStaticInst: StaticInstPtr
+ advancePC()
+ advanceRP()
+ setupFetchRequest(req)
+ preExecute()
+postExecute()
AtomicSimpleCPU
# curThread: ThreadID
+ threadInfo: vector<ExecContext*>
+ curStaticInst: StaticInstPtr
# tick()
+ init()
1
1..*
SimpleExecContext
+ fecthOffset: Addr
+ readIntRegOperand(staticInst, idx): RegVal
+ setIntRegOperand(staticInst, idx, val)
1
1
SimpleThread
# _pcState: unique_ptr<PCStateBase>
# _rpState: unique_ptr<RPStateBase>
# isa: ISA
+ decoder: InstDecoder
+ readIntReg(reg_idx): RegVal
+ setIntReg(reg_idx, val)
- translateRegIdx(reg_idx): RegIndex
Abstract: ExecContext
+ readIntRegOperand(staticInst, idx): RegVal = 0
+ setIntRegOperand(staticInst, idx, val) = 0
+ pcState(): PCStateBase = 0
+ pcState(val) = 0
+ rpState(): RPStateBase = 0
+ rpState(val) = 0
+ readMem(addr, data, size) = 0
+ writeMem(data, size, addr, res) = 0

Figure 6.1: An Illustrative UML Diagram of Atomic Single Cycle CPU

6.2.2 Register Translation

Since there is no cycle difference between decode and execute, register translation is done when the register is accessed. Therefore, when the *readIntReg()* function is executed at execution stage to read from register number, an extra *translateRegIdx()* function is executed to translate the register number from the relative offset to register pointer to the actual physical register number. The same translation applies to writing to integer register.

```
RegVal readIntRegFlat(RegIndex idx) const override
{
    return intRegs[translateRegIdx(idx)];
}
void setIntRegFlat(RegIndex idx, RegVal val) override
{
    intRegs[translateRegIdx(idx)] = val;
}
RegIndex translateRegIdx(RegIndex reg_idx) const
{
    return (_rpState->rp() - reg_idx) % StraightISA::MaxRP;
}
```

Listing 11: Register Translation in simple CPU in gem5

6.2.3 RP Increment

Since it is a single cycle simulator, there are no side effects on branch misprediction. Register pointer is rigidly incremented at the end of each tick cycle to simulate the RP counter. It wraps around when the RP counter reaches its *MaxRP*.

```
class RPStateBase : public Serializable
{
    public:
    void advance() {
        this->_rp = (this->_rp + 1) % StraightISA::MaxRP;
    }
}
```

Listing 12: advance() function for Register Pointer in simple CPU in gem5

6.2.4 Zero Register

Unlike the conventional RISC-V, there is no special *$ZERO* register in the register file. To use zero register as the source operand, the register where RP points at refers to zero at each instruction. Every register resets itself to zero when it is read from for source operand. If the source operand is referring the to register where RP points at, the source operand is essentially accessing a zero register. This means, the maximum relative distance the source operand can refer to is $2^{\text{no-of-bits}}-1$.

6.3 Out-of-Order Processor Modelling – Performance Modelling

Out-of-Order(O3) processor in gem5 simulates processes in a similar manner as a real processor. Register pointer advances and translation cannot be dealt the same way as single cycle processor. In addition, register renaming stage should be removed from the pipeline. At commit stage, instruction retirement and recovery mechanism will be altered in the next architecture. The section will discuss some of the implementation details to accommodate the changes in the new architecture.

6.3.1 RP and Register Translation

RP should be incremented at fetch stage of the pipeline and the translation of registers from its relative distance should be done at fetch stage as well.
Therefore, source operands register number is translated at fetch stage by subtracting the relative distance from the RP of the instruction, as shown in Figure 6.2.

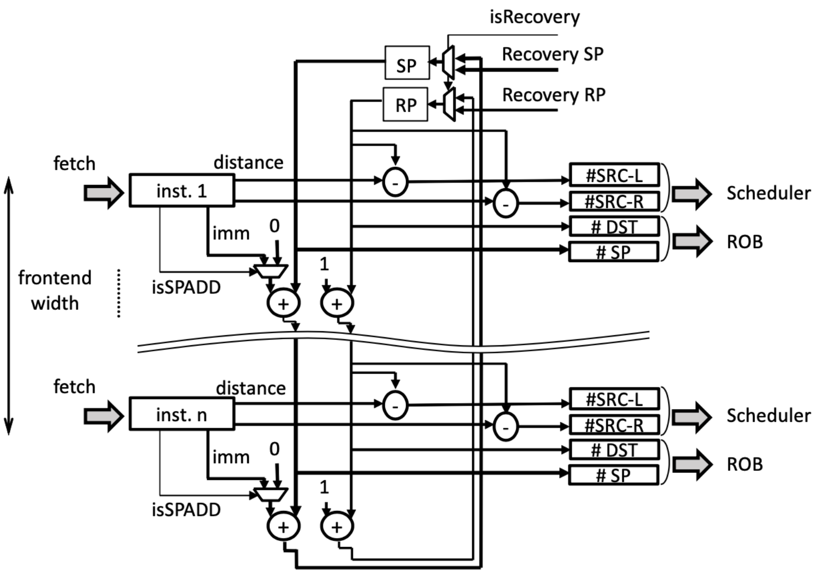


Figure 6.2: The Mechanism of Register Pointer Increment and Operand Translation at Fetch, Taken from [9]

6.3.2 Removing Register Renaming

The first major difference between the new architecture and a conventional RISC architecture is the register renaming. Since STRAIGHT computes the data dependencies at compile time, we don't need register renaming to track data dependencies. The logical register has a fixed one-to-one mapping to the physical registers. Instead of removing the tick of the *rename()* function, the one-to-one mapping between logical and physical registers is pre-defined when CPU initialises the setup. To better model the performance and generate more accurate statistics, statistics tracker for rename, such as, rename *lookups* and *renamedInsts* is discounted. In the reorder buffer, the dependencies of source operands are maintained using the physical registers.

6.3.3 Recovery Mechanism Upon Branch Prediction

When a branch misprediction detected, we need to sqaush the instructions as well as the rp value increments so the instructions relative distance remains correct for register access, since all source operands are tracked through the relative distance from RP value. This is one of the most tricky modifications required to simulate the out-of-order processor behaviour in gem5 for the new STRAIGHT architecture.
RPState and *PCState* classess are stored as a member variable in the *DynamicInst* class, when the instruction is squashed, they will use a *wire()* to relay the PC and RP information between each stage depending on which at which stage the branch misprediction was detected, as shown in Figure 6.3

Squashing at Execution

For instructions squashing at IEW execution, for example, a conditional branch instruction, the current instruction *PC* and *RP* are written to *IEWStruct* which will be wired to commit stage. *IEWStruct* also contains the following information:

1. a boolean value, *squash* to broadcast to commit stage whether squashing is needed;
2. a boolean value, *branchTaken*, to indicate whether the branch should be taken or not;
3. an instruction pointer, *mispredictInst* to broadcast the mispredicted instruction;

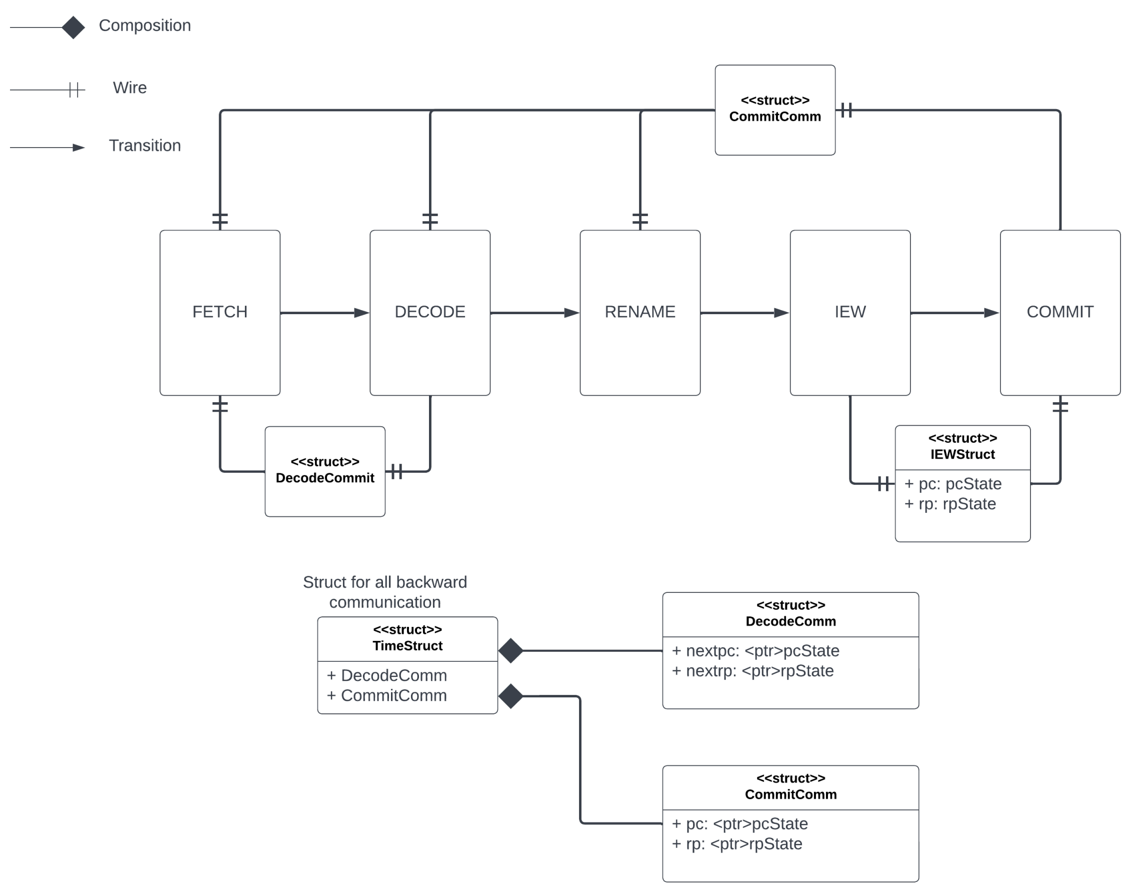


Figure 6.3: The Mechanism of Instructions Recovery in gem5

Commit stage will squash the instructions when misprediction is detected and redirect *PC* and *RP* value to previous stages through *CommitComm*, such as, fetch. Entries in reorder buffer will be squashed as well.
In fetch stage, *CommitComm* will relay the information such as *pc* and *rp* from commit to fetch. Fetch checks the squash signal from *CommitComm* and update its pc and rp value before fetching the next instruction. It also clears out its *fetchQueue* to remove any remaining entries from mispredicted branch.
All the previous stages will also be broadcast with the squash information so they can clear out their buffer/queue entries coming from the mispredicted branch as well.

Squashing at Decoding

If PC-relative control branch is predicted incorrectly, misprediction can be detected at decode stage instead of needing to wait for a few cycles to commit stage. *DecodeComm* will send back the following information to fetch stage:

1. a boolean, *squash*
2. a boolean, *branchMispredict*
3. an instruction pointer, *mispredictInst*
4. pc and rp value

PC adn *RP* will be relayed to fetch just as how squashing is done discussed in the previous section.

6.4 Conclusion

This chapter focuses on the implementation details of the new STRAIGHT architecture in gem5, including defining the new instruction set and modifying both single-cycle and out-of-order processor to model the behaviour and performance of the architecture. In the next chapter, we will use the new architecture modeller to run testbenches to evaluate the performance of this architecture.

Chapter 7

Evaluation

In this chapter, we will use the new architecture modeller to run testbenches to verify the architecture implementation and evaluate the performance of the new STRAIGHT architecture.

7.1 Evaluation Hypothesis

As seen in Figure 7.1, the performance improvement comes from a lower branch misprediction penalty by removing register renaming and shortening the pipeline as the register rename stage takes 2 cycles. With a shortened pipeline, the branch misprediction can be detected and squashed 2 cycles earlier than the conventional RISC architecture and therefore leads to a smaller misprediction penalty and faster execution time.
We are going to test the hypothesis that the performance will improve from removing register renaming and shortening the out-of-order pipeline stage by running various experiments, under the assumption that the STRAIGHT compiler will be as efficient as conventional RISC architecture and the number of instructions executed for the same program will be the same. We are also going to measure the percentage of performance improvement at different branch misprediction rates to evaluate the efficiency of this architecture.

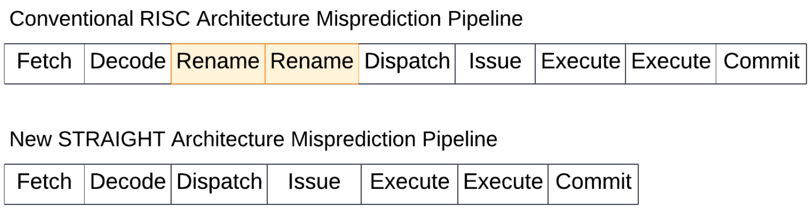

Conventional RISC Architecture Misprediction Pipeline
Fetch
Decode
Rename
Rename
Dispatch
Issue
Execute
Execute
Commit
New STRAIGHT Architecture Misprediction Pipeline
Fetch
Decode
Dispatch
Issue
Execute
Execute
Commit

Figure 7.1: Misprediction Pipeline of Conventional RISC and STRAIGHT Architecture Modelled in gem5

7.1.1 Theoretical Performance Improvement

Before running experiments, we would like to deduce the theoretical performance enhancement from shorter pipeline stage.
First, we define Average Branch Execution Time(ABET) as follows:

$$ABET = a + MP \times MR \tag{7.1}$$

where a is the average execution time per instruction before branch instruction(CPI before Branch), MP is branch misprediction penalty and MR is branch misprediction rate.
The avarage cycles per instruction(CPI) is

$$\begin{aligned} CPI &= b \times ABET + (1 - b) \times a \\ &= a + (MP - MR) \times b \end{aligned} \tag{7.2}$$

where b is percentage of branches in the program.
Hereby we deduce the performance improvement from reducing the number of pipeline stages from MP_1 to MP_2 is

$$\begin{aligned} Reduction\ in\ Execution\ Time &= (a + (MP_1 - MR) \times b) - (a + (MP_2 - MR) \times b) \\ &= (MP_1 - MP_2) \times MR \times b \end{aligned} \tag{7.3}$$

Therefore the percentage improvement from reducing misprediction penalty is

$$\begin{aligned} \%\ Improvement &= \frac{Reduction\ in\ Execution\ Time}{Execution\ Time\ Before\ Removing\ Rename} \\ &= \frac{(MP_1 - MP_2) \times MR \times b}{a + MP_1 \times MR \times b} \end{aligned} \tag{7.4}$$

Branch misprediction penalty is reduced from 9 to 7 cycles in gem5 implementation. ABET, as Equation 7.1 states, is shown in Figure 7.2, where we assume the average CPI before branch is 1.

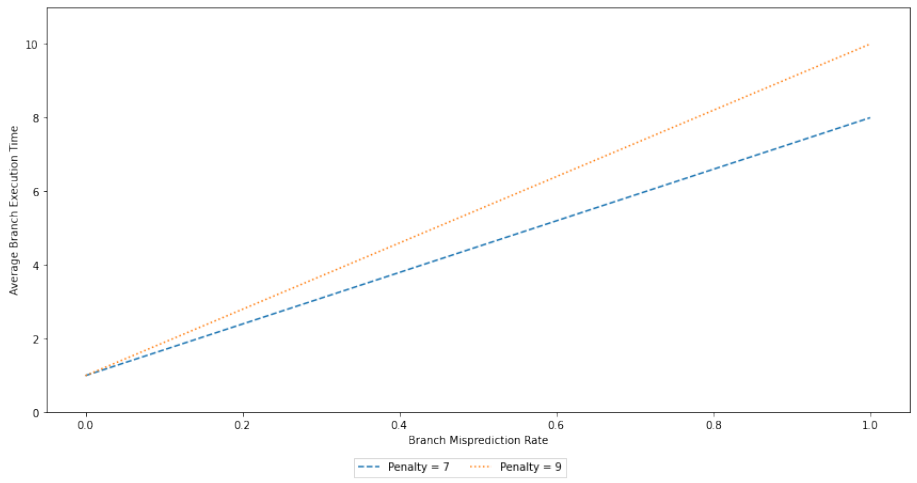


Figure 7.2: Average Branch Execution Time(ABET) for Misprediction Penalty of 7 and 9 Cycles

Equation 7.4 shows that percentage improvement in performance is a function of a, CPI before branch instruction and $MR \times b$, number of mispredicted branches per 100 instructions.
Figure 7.3 shows that under different CPI before branch, the percentage of performance improvement from reducing misprediction penalty from 9 to 7 varies drastically. It can be observed, performance increases at a fast speed as number of mispredicted branches per 100 instructions increase, before slowing down and stagnating at a certain level. At the worst extreme case where branch misprediction rate is 100% and every single instruction is a branch instruction in the program, the performance improvement is higher when CPI before branch instruction is higher. At CPI before branch of 2, the maximum performance improvement reaches up to 21%.
While it is obvious that we will likely not get a program branch misprediction rate of 100% where every single instruction is a branch instruction in real life. Some simple testbenches are simulated to study the effects on real life programs.

7.2 Evaluation Methodology

To evaluate the architecture accuracy and performance, C source code is first compiled to LLVM IR using clang 12.0 with the target *–target=riscv64-pc_ linux-gnu*, an optimisation level *-O2*, and the following compiler flags *-disable-llvm-passes -S -emit-llvm*. The intermediate representation is then optimised using LLVM optimiser to promote memory to register to use *alloca* instructions. Afterwards, the STRAIGHT backend[28] is used to compile LLVM IR to unlinked STRAIGHT assembly code. The assembly code is then linked by a static linker developed by the STRAIGHT research group to remove labels.

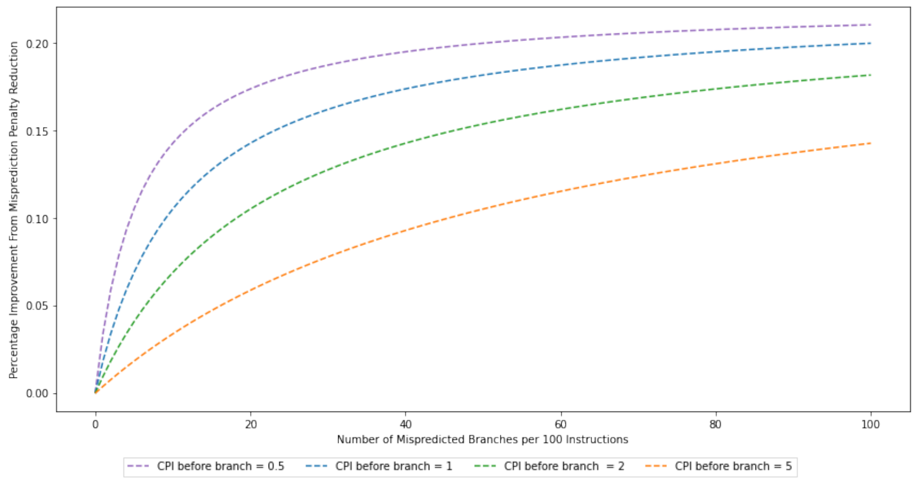


Figure 7.3: Percentage Performance Improvement from Reducing Misprediction Penalty from 9 to 7 under Different CPI before Branch

The assembly code is then passed to a custom parser to fix the differences between STRAIGHT instruction set and our new instruction set. The assembly file is then assembled by a custom assembler to generate binary code and linked with global data sections(in matrix multiply case) to generate executable ELF file which is then loaded by gem5 processor.
After verifying the correctness of the program, we will run the testbenches in out-of-order processor and generate the performance statistics for different metrics and evaluate the efficiency of the processor.

7.3 Verification of Correctness

To verify the implementation accuracy of the processor and instruction set definition, we will run three different testbenches in single-cycle processor, one arithmetic sequence, one Fibonacci number and one matrix multiplication before running the performance benchmark.
The single-cycle processor will also generate intermediate results through execution to be cross-checked with results generated by out-of-order processor.

7.3.1 Sum of Arithmetic Sequence

The mathematical arithmetic sequence is defined as:

$$a_n = a_1 + (n-1) \times d$$

where a_1 is 0 and $d = 1$. The expression to compute the sum of the arithmetic sequence is

$$\sum_{n=1}^{x} a_n$$

The arithmetic sequence contains R-type, I-type, SB-type and UJ-type instructions, which will be used mainly for testing the implementation of the instruction set definition and branch misprediction handling.

Source code

Assembly code

```
int main(){
    int n = 10;
    int a = 0;
    for (int i = 0; i < n; i++){
        a = a + i;
    }
    return a;
}
```

Listing 13: Arithmetic Sequence Function in C Source Code

```
 1      j   4
 2      nop
 3      addi    [0]     0
 4      addi    [1]     0
 5      addi    [2]     0
 6      addi    [5]     0
 7      nop
 8      slli    [4]     32
 9      srai    [1]     32
10      addi    [0]     10      /* n */
11      blt     [2]     [1]      24
12      nop
13      slli    [8]     32
14      srai    [1]     32
15      addi    [0]     93
16      ecall
17      add     [7]     [8]     /* a = a + i */
18      addi    [9]     1       /* i++ */
19      addi    [1]     0       /* i */
20      addi    [3]     0       /* a */
21      addi    [10]    0
22      j       -56
```

Listing 14: Arithmetic Sequence Function in New STRAIGHT Source Code

Explanation and Result

Listing 13 and 14 show a C and assembly implementation of computing the sum of the arithmetic sequence of a certain lenght. The highlighted lines are the their corresponding loop bodies. Line 17 computes the addition of variable a and i and line 18 increments i. Lines 19-21 follow the STRAIGHT calling convention discussed in Chapter 4 and move the producers of the arguments prior to the jump instruction. Line 11 is a conditional branch instruction to jump out of the loop body when variable i reaches 10. After running in a single-cycle processor, the program executes returns the result of the arithmetic sequence in line 14 before executing the argument passing and ecall instructions in lines 15-16.

7.3.2 Fibonacci Number

The Fibonacci numbers are defined by the recurrence relation

$$F_0 = 0, \quad F_1 = 1,$$

and

$$F_n = F_{n-1} + F_{n-2}$$

for n $>$ 1.

Source code

```c
int main(){
    int last_fib = 0;
    int length = 3;
    for (int n = 0; n <= length; n ++) {
        int prev = 0;
        int curr = 1;
        for (int i = 2; i <= n; i++) {
            int tmp = curr;
            curr = prev + curr;
            prev = tmp;
        }
        last_fib = curr;
    }
    return last_fib;
}
```

Listing 15: Fibonacci Number Function in C Source Code

Assembly code

```
1   j 4
2   nop
3   addi [0] 0
4   addi [1] 0
5   addi [2] 0
6   addi [5] 0
7   nop
8   slli [3] 32
9   srai [1] 32
10  addi [0] 4
11  blt [2] [1] 28
12  nop
13  slli [9] 32
14  srai [1] 32
15  addi [1] 0
16  addi [0] 93
17  ecall
18  addi [0] 2
19  addi [0] 0
20  addi [0] 1
21  addi [3] 0
22  addi [3] 0
23  addi [3] 0
24  addi [12] 0
25  addi [14] 0
26  nop
27  slli [6] 32
28  srai [1] 32
29  slli [4] 32
30  srai [1] 32
31  bge [1] [3] 24
32  addi [7] 1
33  addi [10] 0
34  addi [2] 0
35  addi [11] 0
36  j -112
37  add [10] [9]
38  addi [12] 1
39  addi [1] 0
40  addi [12] 0
41  addi [4] 0
42  addi [13] 0
43  addi [13] 0
44  j -68
```

Listing 16: Fibonacci Number Function in New STRAIGHT Source Code

Explanation and Result

Listing 15 and 16 show a C and assembly implementation of computing the Fibonacci number from F_0 up to F_n. A nested for-loop is used to compute all previous Fibonacci number instead of memoisation in an array since we could obtain loop of different number of iterations and we will likely get more branch mispredictions to study the effect of the pipeline depth on branch misprediction penalty and total performance.
The highlighted lines are the their corresponding conditional branch statements for the nested for-loops. Lines 27-31 and 37-44 are the inner loop body, which is used to compute Fibonacci number of a given length, n. Instruction on line 31, *bge [1] [3] 24* is the conditional branch on the inner loop. When the iteration i reaches the n^{th} position in the Fibonacci number, the branch is

not taken on the line and jumps to line 8, the beginning of the outer loop. *blt [2] [1] 28* is the conditional jump statement for the outer loop, when it reaches the number of Fibonacci number we want compute, the branch is not taken and the program exits using *ecall.*

7.3.3 Matrix Multiplication

Matrix multiplication contains a lot of load and store data from the memory section which is used to investigate the cache efficiency of the new architecture. It also provides some premeliminary work on store-load mis-speculation study. We can also verify the correctness of load and store instructions implemented in gem5.

Source code

```c
int main()
{
    int a = 12;
    int b = 3;
    static int A[a][b] = {
        /* ... */
    };
    static int B[a][b] = {
        /* ... */
    };
    static int C[6][6] = {
        /* ... */
    };

    for (int i = 0; i < a; i++) {
        for (int j = 0; j < b; j++){
            for (int k = 0; k < b; k++){
                C[j][k] += A[i][j] * B[i][k];
            }
        }
    }
    return C[0][0];
}
```

Listing 17: Matrix Multiplication Function in C Source Code

Assembly code

```
j 4
nop
addi    [0]    0
addi    [1]    0
addi    [4]    0
nop
slli    [4]    32
srai    [1]    32
addi    [0]    12
blt     [2]    [1]    32
nop
lui     16
addi    [1]    288
lw      [1]    16
addi    [1]    0
addi    [0]    93
ecall
slli    [7]    32
srai    [1]    32
addi    [0]    12
mul     [2]    [1]
lui     16
addi    [1]    0
add     [1]    [3]
lui     16
addi    [1]    144
add     [1]    [6]
lw      [1]    0
lw      [5]    0
mul     [1]    [2]
lui     16
addi    [1]    288
lw      [1]    0
add     [1]    [4]
sw      [3]    [1]    0

/*      ...    */

lw      [51]   8
lw      [55]   8
mul     [1]    [2]
lw      [49]   32
add     [1]    [2]
sw      [51]   [1]    32
addi    [73]   1
addi    [1]    0
addi    [74]   0
j       -320
```

Listing 18: Matrix Multiplication Function in New STRAIGHT Source Code

Explanation and Result

We assign the memory of all three matrices to global using the *static* keyword. In the matrix multiplication loop, we flatten both inner loops since the objective is to look at data dependence speculation. The matrices are stored in a global memory pointer starting 0x10000. In the assembly code, we use *ld* and *st* to load/store the matrix data from/to heap memory. *addi* is used to increment/decrement the offset from the memory address pointer so we fetch the data at the correct memory address. After we finish computing the multiplication, a *st* instrucion is executed to store the data back to the correct memory position before going to the next loop iteration. The highlighted lines are the conditional statement on the outermost loop. At the end of the program, we obtain the result of a 3x12 and 12x3 matrix multiplication.

7.4 Performance Evaluation

7.4.1 Configurations

The simulator models all full pipeline stages. The Table 7.1 below branch prediction, memory dependency prediction, a load-store queue (LSQ) for memory disambiguation, cache hit/miss prediction, the scheduler replay, a stream prefetcher for data caches, and the mechanisms for the misprediction recovery.

Fetch/Decode/Rename Width	2
Dispatch/Issue/Writeback Width	2
Commit Width	8
Instruction Queue Size	64
ROB Size	192
BranchPred	Tournament Predictor
LD/ST Queue	32
Integer Register File	1024
L1I Cache	4 kB, assoc 2, 64 B block 2 cycle hit latency
L1D Cache	2 kB, assoc 2, 64 B block 2 cycle hit latency
L2 Cache	64 kB, assoc 8, 64 B block 8 cycle hit latency

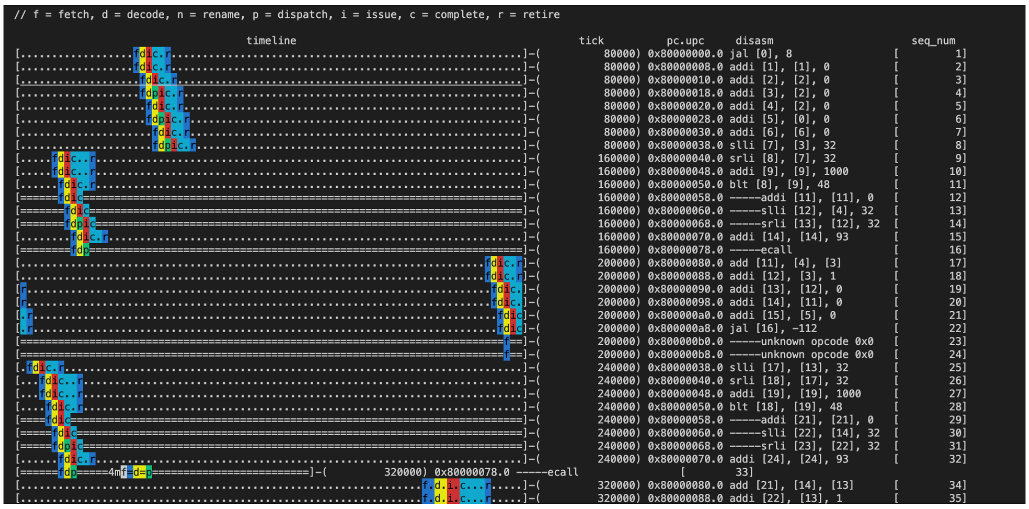
Table 7.1: Configurations of Simulator

7.4.2 Result Analysis – Sum of Arithmetic Sequence

We compute the sum of arithmetic sequence for length of 10, 50, 100, 500 and 1000 respectively to evaluate the branch prediction efficiency and performance changes as the number of iterations grow larger.

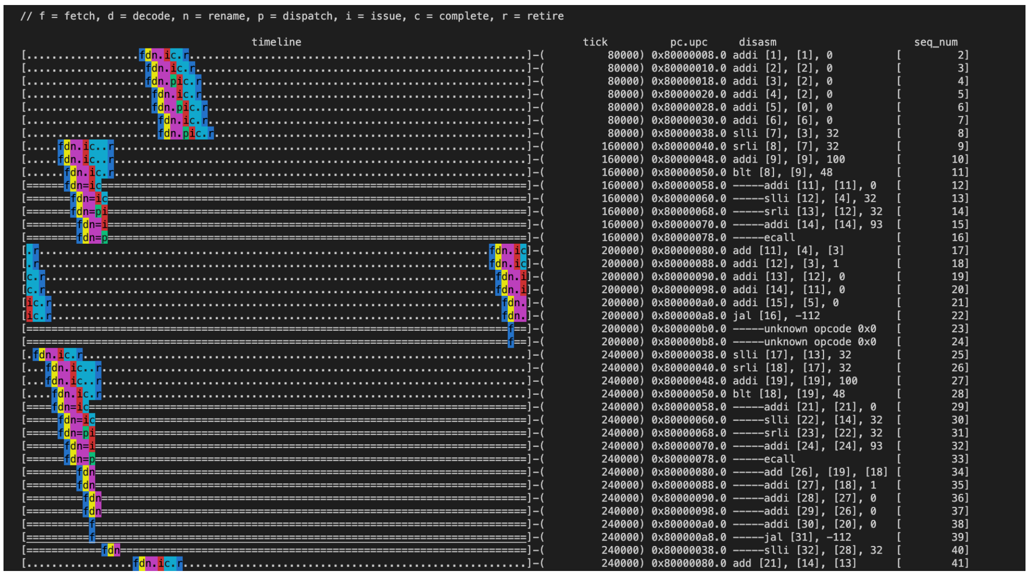
Visualisation

Figure 7.4 illustrates the out-of-order pipeline in a timeline. Two instructions are fetched in each cycle and branch misprediction happens in the *blt* instruction. Therefore, subsequent instructions are squashed. As most iterations are executed in the processor, the branch predictor becomes more effective and most branches are predicted correctly towards the end of the pipeline and there are more in-flight instructions at each cycle and the performance improves as the number of iterations increases, shown in Figure 7.5 It can be observed from the Figure 7.4 that the branch misprediction penalty becomes smaller when *blt* instruction is mispredicted. Figure 7.4a is the pipeline view withou rename while Figure 7.4b with rename. It can be observed that the pipeline depth redices from 9 to 7 cycles nad the pink rename stage in removed in the pieline. It costs the RISC architecture with renaming, in Figure 7.4a, 12 instructions squashing compared to only 5 in the STRAIGHT architecture, in Figure 7.4b.



```
// f = fetch, d = decode, n = rename, p = dispatch, i = issue, c = complete, r = retire

                                   timeline                                              tick         pc.upc     disasm                      seq_num
[..................fdic.r........................................................]-(         80000) 0x80000000.0 jal [0], 8                 [         1]
[..................fdic.r........................................................]-(         80000) 0x80000008.0 addi [1], [1], 0           [         2]
[...................fdic.r.......................................................]-(         80000) 0x80000010.0 addi [2], [2], 0           [         3]
[...................fdpic.r......................................................]-(         80000) 0x80000018.0 addi [3], [2], 0           [         4]
[....................fdic.r......................................................]-(         80000) 0x80000020.0 addi [4], [2], 0           [         5]
[....................fdpic.r.....................................................]-(         80000) 0x80000028.0 addi [5], [0], 0           [         6]
[.....................fdic.r.....................................................]-(         80000) 0x80000030.0 addi [6], [6], 0           [         7]
[.....................fdpic.r....................................................]-(         80000) 0x80000038.0 slli [7], [3], 32          [         8]
[.....fdic..r....................................................................]-(        160000) 0x80000040.0 srli [8], [7], 32          [         9]
[.....fdic..r....................................................................]-(        160000) 0x80000048.0 addi [9], [9], 1000       [        10]
[......fdic.r....................................................................]-(        160000) 0x80000050.0 blt [8], [9], 48          [        11]
[======fdic======================================================================]-(        160000) 0x80000058.0 -----addi [11], [11], 0   [        12]
[=======fdic=====================================================================]-(        160000) 0x80000060.0 -----slli [12], [4], 32   [        13]
[=======fdpic====================================================================]-(        160000) 0x80000068.0 -----srli [13], [12], 32  [        14]
[........fdic.r..................................................................]-(        160000) 0x80000070.0 addi [14], [14], 93       [        15]
[========fdp=====================================================================]-(        160000) 0x80000078.0 -----ecall                [        16]
[..........................................................................fdic.r]-(        200000) 0x80000080.0 add [11], [4], [3]        [        17]
[..........................................................................fdic.r]-(        200000) 0x80000088.0 addi [12], [3], 1         [        18]
[r..........................................................................fdic.]-(        200000) 0x80000090.0 addi [13], [12], 0        [        19]
[r..........................................................................fdic.]-(        200000) 0x80000098.0 addi [14], [11], 0        [        20]
[.r..........................................................................fdic]-(        200000) 0x800000a0.0 addi [15], [5], 0         [        21]
[.r..........................................................................fdic]-(        200000) 0x800000a8.0 jal [16], -112            [        22]
[=============================================================================f==]-(        200000) 0x800000b0.0 -----unknown opcode 0x0   [        23]
[=============================================================================f==]-(        200000) 0x800000b8.0 -----unknown opcode 0x0   [        24]
[.fdic.r.........................................................................]-(        240000) 0x80000038.0 slli [17], [13], 32       [        25]
[...fdic..r......................................................................]-(        240000) 0x80000040.0 srli [18], [17], 32       [        26]
[...fdic..r......................................................................]-(        240000) 0x80000048.0 addi [19], [19], 1000     [        27]
[....fdic.r......................................................................]-(        240000) 0x80000050.0 blt [18], [19], 48        [        28]
[====fdic========================================================================]-(        240000) 0x80000058.0 -----addi [21], [21], 0   [        29]
[=====fdic=======================================================================]-(        240000) 0x80000060.0 -----slli [22], [14], 32  [        30]
[=====fdpic======================================================================]-(        240000) 0x80000068.0 -----srli [23], [22], 32  [        31]
[......fdic.r....................................................................]-(        240000) 0x80000070.0 addi [24], [24], 93       [        32]
[======fdp=====4mf=d=p=========================]-(         320000) 0x80000078.0 -----ecall                [        33]
[................................................................f.d.i.c...r.....]-(        320000) 0x80000080.0 add [21], [14], [13]      [        34]
[................................................................f.d.i.c...r.....]-(        320000) 0x80000088.0 addi [22], [13], 1        [        35]
```

(a) Beginning of the pipeline after removing register renaming



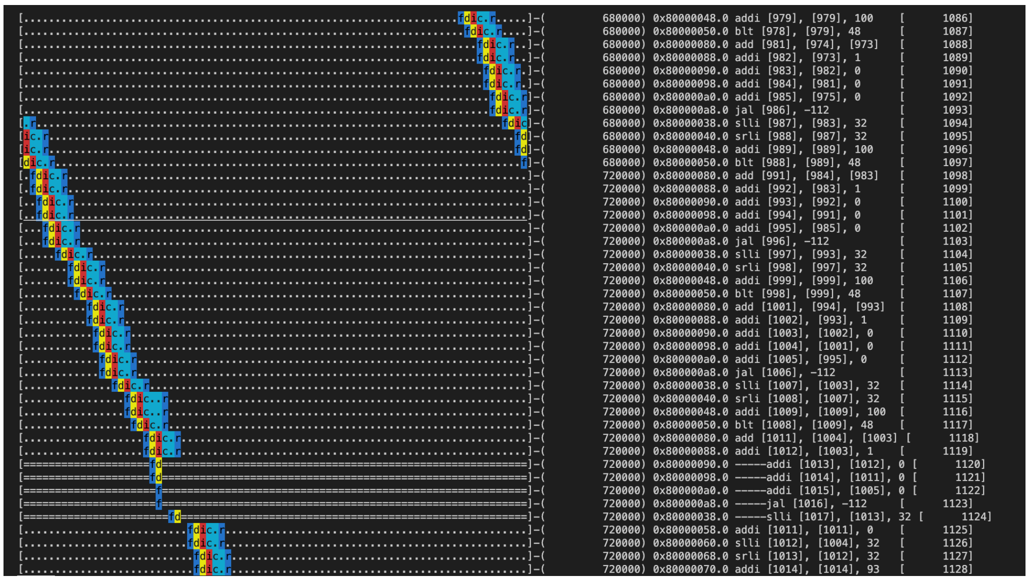
```
// f = fetch, d = decode, n = rename, p = dispatch, i = issue, c = complete, r = retire

                                   timeline                                              tick         pc.upc     disasm                      seq_num
[..................fdn.ic.r......................................................]-(         80000) 0x80000008.0 addi [1], [1], 0           [         2]
[...................fdn.ic.r.....................................................]-(         80000) 0x80000010.0 addi [2], [2], 0           [         3]
[...................fdn.pic.r....................................................]-(         80000) 0x80000018.0 addi [3], [2], 0           [         4]
[....................fdn.ic.r....................................................]-(         80000) 0x80000020.0 addi [4], [2], 0           [         5]
[....................fdn.pic.r...................................................]-(         80000) 0x80000028.0 addi [5], [0], 0           [         6]
[.....................fdn.ic.r...................................................]-(         80000) 0x80000030.0 addi [6], [6], 0           [         7]
[.....................fdn.pic.r..................................................]-(         80000) 0x80000038.0 slli [7], [3], 32          [         8]
[.....fdn.ic..r..................................................................]-(        160000) 0x80000040.0 srli [8], [7], 32          [         9]
[.....fdn.ic..r..................................................................]-(        160000) 0x80000048.0 addi [9], [9], 100        [        10]
[......fdn.ic.r..................................................................]-(        160000) 0x80000050.0 blt [8], [9], 48          [        11]
[======fdn=ic====================================================================]-(        160000) 0x80000058.0 -----addi [11], [11], 0   [        12]
[=======fdn=ic===================================================================]-(        160000) 0x80000060.0 -----slli [12], [4], 32   [        13]
[=======fdn=pi===================================================================]-(        160000) 0x80000068.0 -----srli [13], [12], 32  [        14]
[========fdn=i===================================================================]-(        160000) 0x80000070.0 -----addi [14], [14], 93  [        15]
[========fdn=p===================================================================]-(        160000) 0x80000078.0 -----ecall                [        16]
[.r........................................................................fdn.ic]-(        200000) 0x80000080.0 add [11], [4], [3]        [        17]
[.r........................................................................fdn.ic]-(        200000) 0x80000088.0 addi [12], [3], 1         [        18]
[c.r........................................................................fdn.i]-(        200000) 0x80000090.0 addi [13], [12], 0        [        19]
[c.r........................................................................fdn.i]-(        200000) 0x80000098.0 addi [14], [11], 0        [        20]
[ic.r........................................................................fdn.]-(        200000) 0x800000a0.0 addi [15], [5], 0         [        21]
[ic.r........................................................................fdn.]-(        200000) 0x800000a8.0 jal [16], -112            [        22]
[=============================================================================f==]-(        200000) 0x800000b0.0 -----unknown opcode 0x0   [        23]
[=============================================================================f==]-(        200000) 0x800000b8.0 -----unknown opcode 0x0   [        24]
[.fdn.ic.r.......................................................................]-(        240000) 0x80000038.0 slli [17], [13], 32       [        25]
[...fdn.ic..r....................................................................]-(        240000) 0x80000040.0 srli [18], [17], 32       [        26]
[...fdn.ic..r....................................................................]-(        240000) 0x80000048.0 addi [19], [19], 100      [        27]
[....fdn.ic.r....................................................................]-(        240000) 0x80000050.0 blt [18], [19], 48        [        28]
[====fdn=ic======================================================================]-(        240000) 0x80000058.0 -----addi [21], [21], 0   [        29]
[=====fdn=ic=====================================================================]-(        240000) 0x80000060.0 -----slli [22], [14], 32  [        30]
[=====fdn=pi=====================================================================]-(        240000) 0x80000068.0 -----srli [23], [22], 32  [        31]
[======fdn=i=====================================================================]-(        240000) 0x80000070.0 -----addi [24], [24], 93  [        32]
[======fdn=p=====================================================================]-(        240000) 0x80000078.0 -----ecall                [        33]
[========fdn=====================================================================]-(        240000) 0x80000080.0 -----add [26], [19], [18] [        34]
[========fdn=====================================================================]-(        240000) 0x80000088.0 -----addi [27], [18], 1   [        35]
[=========fdn====================================================================]-(        240000) 0x80000090.0 -----addi [28], [27], 0   [        36]
[=========fdn====================================================================]-(        240000) 0x80000098.0 -----addi [29], [26], 0   [        37]
[==========f=====================================================================]-(        240000) 0x800000a0.0 -----addi [30], [20], 0   [        38]
[==========f=====================================================================]-(        240000) 0x800000a8.0 -----jal [31], -112       [        39]
[============fdn=================================================================]-(        240000) 0x80000038.0 -----slli [32], [28], 32  [        40]
[.................fdn.ic.r.......................................................]-(        240000) 0x80000080.0 add [21], [14], [13]      [        41]
```

(b) Beginning of the pipeline before removing register renaming

Figure 7.4: Pipeline View of Out-of-Order Processor for Sequence length of 100

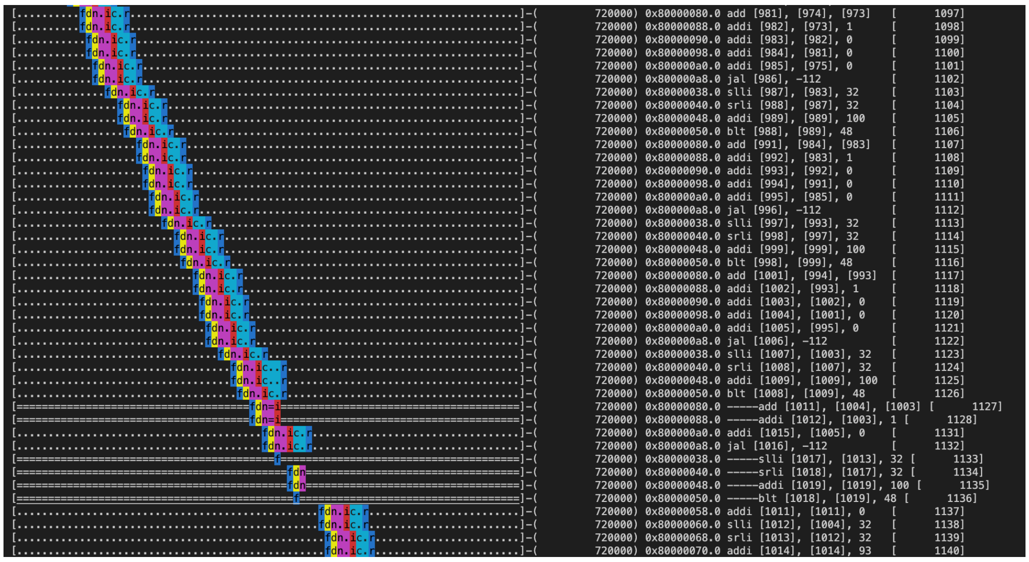
Performance – IPC and Branch Misprediction

As shown in Figure 7.6, the branch misprediction rate decreases as the number of values in the summation increases since the branch predictor and branch target buffer will update the branch prediction result and the prediction accuracy increases. Therefore misprediction rate reduces from over 50% to below 1% and instructions per cycle(IPC) increases from 0.17 to 1.05 in Figure 7.6. Therefore, the effect of branch misprediction penalty becomes less significant to the overall execution time.



```
680000) 0x80000048.0 addi [979], [979], 100    [      1086]
680000) 0x80000050.0 blt [978], [979], 48      [      1087]
680000) 0x80000080.0 add [981], [974], [973]   [      1088]
680000) 0x80000088.0 addi [982], [973], 1      [      1089]
680000) 0x80000090.0 addi [983], [982], 0      [      1090]
680000) 0x80000098.0 addi [984], [981], 0      [      1091]
680000) 0x800000a0.0 addi [985], [975], 0      [      1092]
680000) 0x800000a8.0 jal [986], -112           [      1093]
680000) 0x80000038.0 slli [987], [983], 32     [      1094]
680000) 0x80000040.0 srli [988], [987], 32     [      1095]
680000) 0x80000048.0 addi [989], [989], 100    [      1096]
680000) 0x80000050.0 blt [988], [989], 48      [      1097]
720000) 0x80000080.0 add [991], [984], [983]   [      1098]
720000) 0x80000088.0 addi [992], [983], 1      [      1099]
720000) 0x80000090.0 addi [993], [992], 0      [      1100]
720000) 0x80000098.0 addi [994], [991], 0      [      1101]
720000) 0x800000a0.0 addi [995], [985], 0      [      1102]
720000) 0x800000a8.0 jal [996], -112           [      1103]
720000) 0x80000038.0 slli [997], [993], 32     [      1104]
720000) 0x80000040.0 srli [998], [997], 32     [      1105]
720000) 0x80000048.0 addi [999], [999], 100    [      1106]
720000) 0x80000050.0 blt [998], [999], 48      [      1107]
720000) 0x80000080.0 add [1001], [994], [993]  [      1108]
720000) 0x80000088.0 addi [1002], [993], 1     [      1109]
720000) 0x80000090.0 addi [1003], [1002], 0    [      1110]
720000) 0x80000098.0 addi [1004], [1001], 0    [      1111]
720000) 0x800000a0.0 addi [1005], [995], 0     [      1112]
720000) 0x800000a8.0 jal [1006], -112          [      1113]
720000) 0x80000038.0 slli [1007], [1003], 32   [      1114]
720000) 0x80000040.0 srli [1008], [1007], 32   [      1115]
720000) 0x80000048.0 addi [1009], [1009], 100  [      1116]
720000) 0x80000050.0 blt [1008], [1009], 48    [      1117]
720000) 0x80000080.0 add [1011], [1004], [1003] [      1118]
720000) 0x80000088.0 addi [1012], [1003], 1    [      1119]
720000) 0x80000090.0 -----addi [1013], [1012], 0 [      1120]
720000) 0x80000098.0 -----addi [1014], [1011], 0 [      1121]
720000) 0x800000a0.0 -----addi [1015], [1005], 0 [      1122]
720000) 0x800000a8.0 -----jal [1016], -112     [      1123]
720000) 0x80000038.0 -----slli [1017], [1013], 32 [      1124]
720000) 0x80000058.0 addi [1011], [1011], 0    [      1125]
720000) 0x80000060.0 slli [1012], [1004], 32   [      1126]
720000) 0x80000068.0 srli [1013], [1012], 32   [      1127]
720000) 0x80000070.0 addi [1014], [1014], 93   [      1128]
```

(a) End of the pipeline after removing register renaming



```
720000) 0x80000080.0 add [981], [974], [973]   [      1097]
720000) 0x80000088.0 addi [982], [973], 1      [      1098]
720000) 0x80000090.0 addi [983], [982], 0      [      1099]
720000) 0x80000098.0 addi [984], [981], 0      [      1100]
720000) 0x800000a0.0 addi [985], [975], 0      [      1101]
720000) 0x800000a8.0 jal [986], -112           [      1102]
720000) 0x80000038.0 slli [987], [983], 32     [      1103]
720000) 0x80000040.0 srli [988], [987], 32     [      1104]
720000) 0x80000048.0 addi [989], [989], 100    [      1105]
720000) 0x80000050.0 blt [988], [989], 48      [      1106]
720000) 0x80000080.0 add [991], [984], [983]   [      1107]
720000) 0x80000088.0 addi [992], [983], 1      [      1108]
720000) 0x80000090.0 addi [993], [992], 0      [      1109]
720000) 0x80000098.0 addi [994], [991], 0      [      1110]
720000) 0x800000a0.0 addi [995], [985], 0      [      1111]
720000) 0x800000a8.0 jal [996], -112           [      1112]
720000) 0x80000038.0 slli [997], [993], 32     [      1113]
720000) 0x80000040.0 srli [998], [997], 32     [      1114]
720000) 0x80000048.0 addi [999], [999], 100    [      1115]
720000) 0x80000050.0 blt [998], [999], 48      [      1116]
720000) 0x80000080.0 add [1001], [994], [993]  [      1117]
720000) 0x80000088.0 addi [1002], [993], 1     [      1118]
720000) 0x80000090.0 addi [1003], [1002], 0    [      1119]
720000) 0x80000098.0 addi [1004], [1001], 0    [      1120]
720000) 0x800000a0.0 addi [1005], [995], 0     [      1121]
720000) 0x800000a8.0 jal [1006], -112          [      1122]
720000) 0x80000038.0 slli [1007], [1003], 32   [      1123]
720000) 0x80000040.0 srli [1008], [1007], 32   [      1124]
720000) 0x80000048.0 addi [1009], [1009], 100  [      1125]
720000) 0x80000050.0 blt [1008], [1009], 48    [      1126]
720000) 0x80000080.0 -----add [1011], [1004], [1003] [      1127]
720000) 0x80000088.0 -----addi [1012], [1003], 1 [      1128]
720000) 0x800000a0.0 addi [1015], [1005], 0    [      1131]
720000) 0x800000a8.0 jal [1016], -112          [      1132]
720000) 0x80000038.0 -----slli [1017], [1013], 32 [      1133]
720000) 0x80000040.0 -----srli [1018], [1017], 32 [      1134]
720000) 0x80000048.0 -----addi [1019], [1019], 100 [      1135]
720000) 0x80000050.0 -----blt [1018], [1019], 48 [      1136]
720000) 0x80000058.0 addi [1011], [1011], 0    [      1137]
720000) 0x80000060.0 slli [1012], [1004], 32   [      1138]
720000) 0x80000068.0 srli [1013], [1012], 32   [      1139]
720000) 0x80000070.0 addi [1014], [1014], 93   [      1140]
```

(b) End of the pipeline before removing register renaming

Figure 7.5: Pipeline View of Out-of-Order Processor for Sequence length of 100

Performance Difference Before and After Register Renaming

As a result of removing register renaming in the processor pipeline, there is a clear performance boost at different number iterations for arithmetic sequence, as seen in Figure 7.7. As a result of improving branch misprediction rate, it can also be proved in Figure 7.8, the percentage improvement in performance in terms of execution time is becoming smaller, from 3.5% to below 0.5%. It confirms our hypothesis that removing register renaming increases the program performance from earlier instruction squashing and recovery.

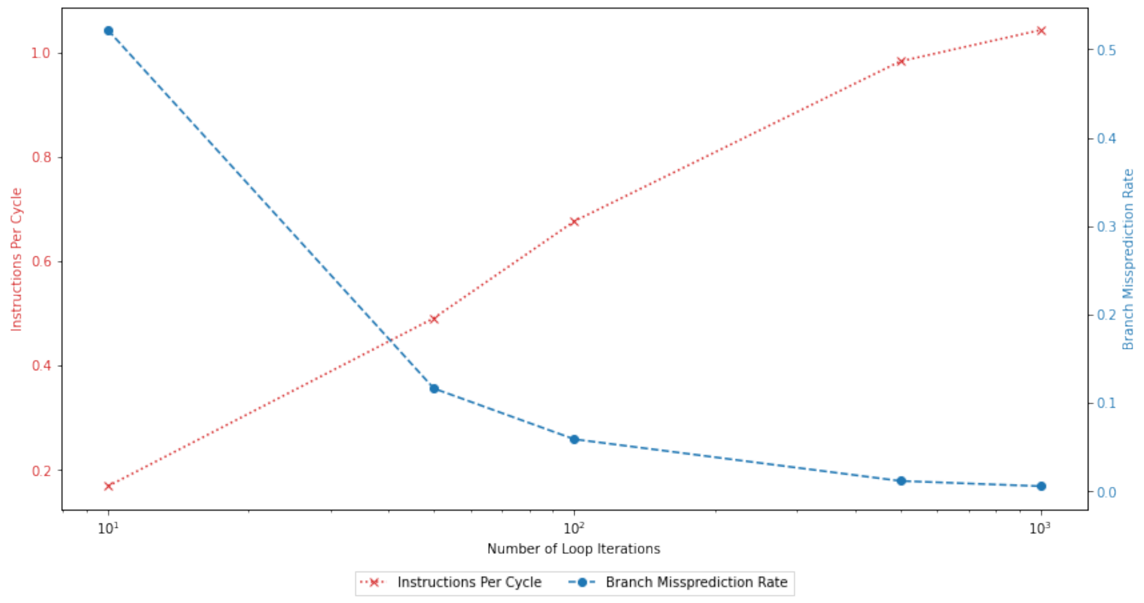


Figure 7.6: Instructions per Cycle and Branch Misprediction Rate for Different Iterations in Logrithmetic Scale in Arithmetic Sequence

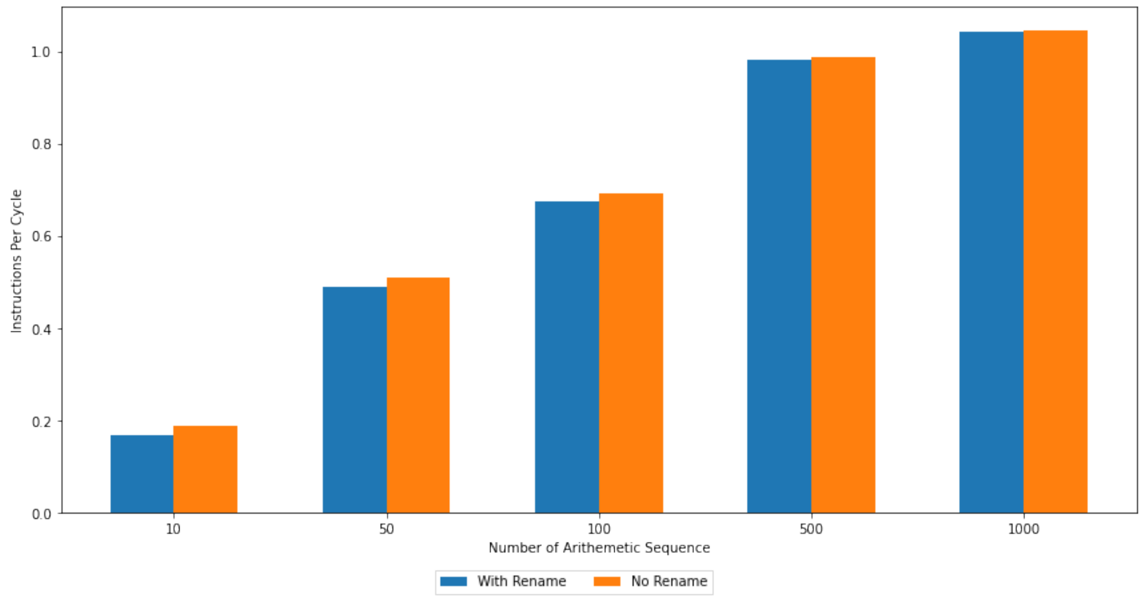


Figure 7.7: Instruction Per Cycle Before and After Removing Register Renaming for Arithmetic Sequence

7.4.3 Result Analysis – Fibonacci Number

We compute the Fibonacci Number from 1 to 50 to evaluate the effect of branch misprediction on the performance of the program.

Performance – IPC and Branch Misprediction Rate

As shown in Figure 7.9, branch misprediction rate stays high above 30% when computing less than 10 Fibonacci numbers. This is due to the nature of the program that early numbers have a smaller number of iterations, which means branch has an similar chance of 'taken' and 'not taken', mkaing the branch misprediction rate high. As the number of Fibonacci number computed increases, the branch predictor efficiency becomes higehr and IPC also increases to above 1 when computing more than 30 Fibonacci numbers.

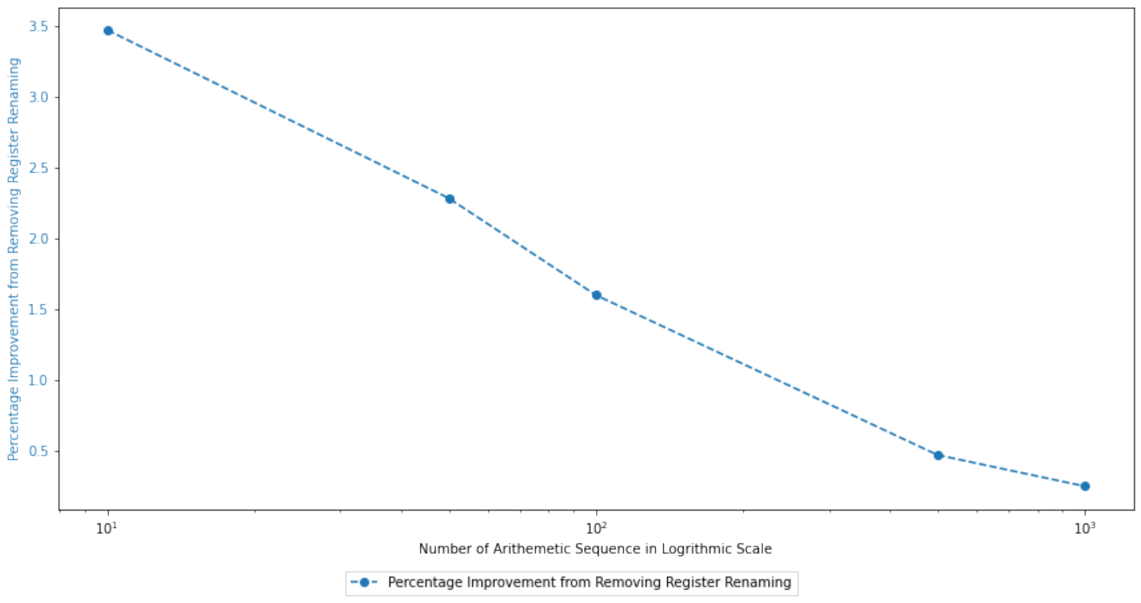


Figure 7.8: Percentage of Performance Improvement from Removing Register Renaming in Arithmetic Sequence

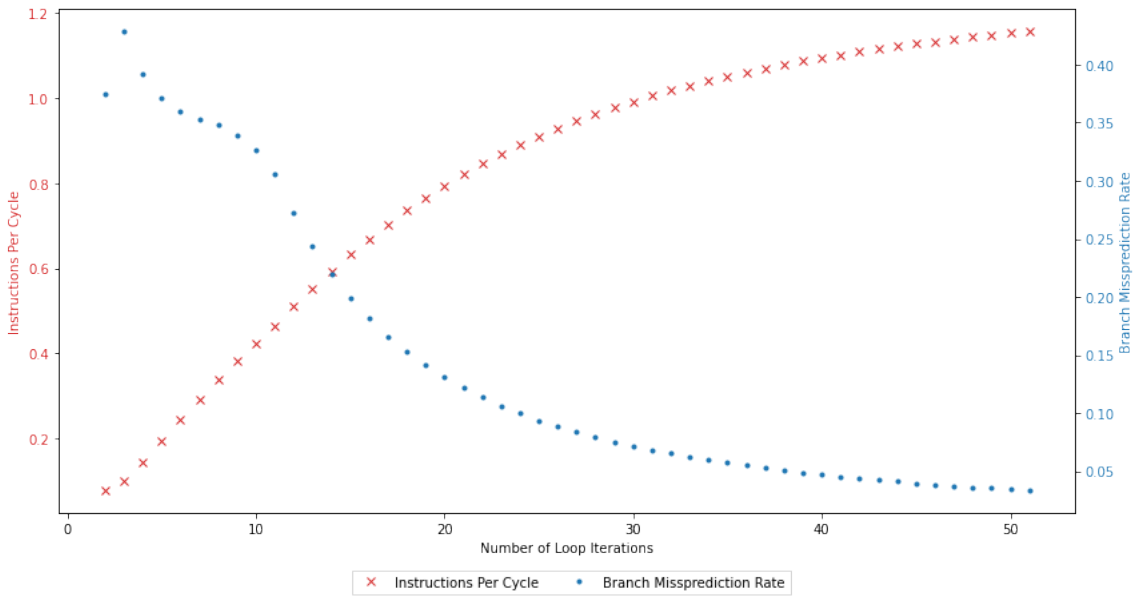


Figure 7.9: Iterations Per Cycle(IPC) and Branch Misprediction Rate for Different Iterations in Fibonacci Number

Performance Difference Before and After Register Renaming

As a result of high branch misprediction rate at early iterations, Figure 7.10 shows that the performance increase from removing register renaming peaked at computing 10 Fibonacci numbers. At 10, the number of branch mispredictions and branch misprediction rate are both very high, where we can see a clear and significant performance improvement from removing register renaming and having a shorter pipeline stage. As the number of Fibonacci number increaess, branch misprediction rate improves significantly to below 10% after around 30 Fibonacci numbers. Therefore, the percentage in performance improvement is also lower.

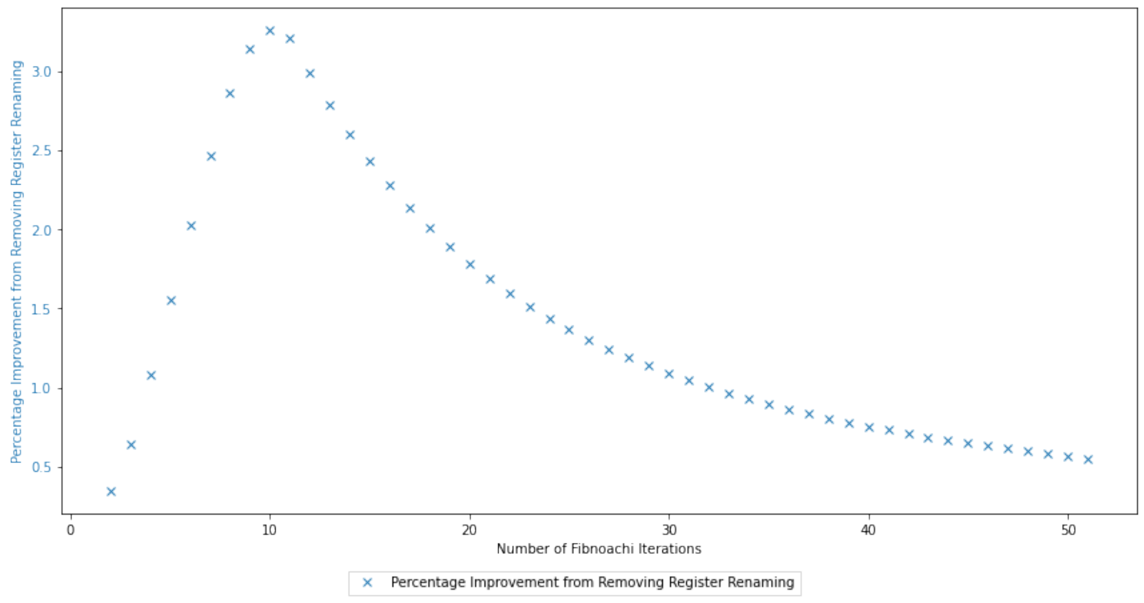


Figure 7.10: Percentage of Performance Improvement from Removing Register Renamingin Fibonacci Number

7.4.4 Result Analysis – Matrix Multiplication

As seen in Figure 7.11, level 1 data cache miss rate goes up as the size of the matrix becomes larger. Similarly the average time taken for a load instruction also increases from 5 to 8 cycles.

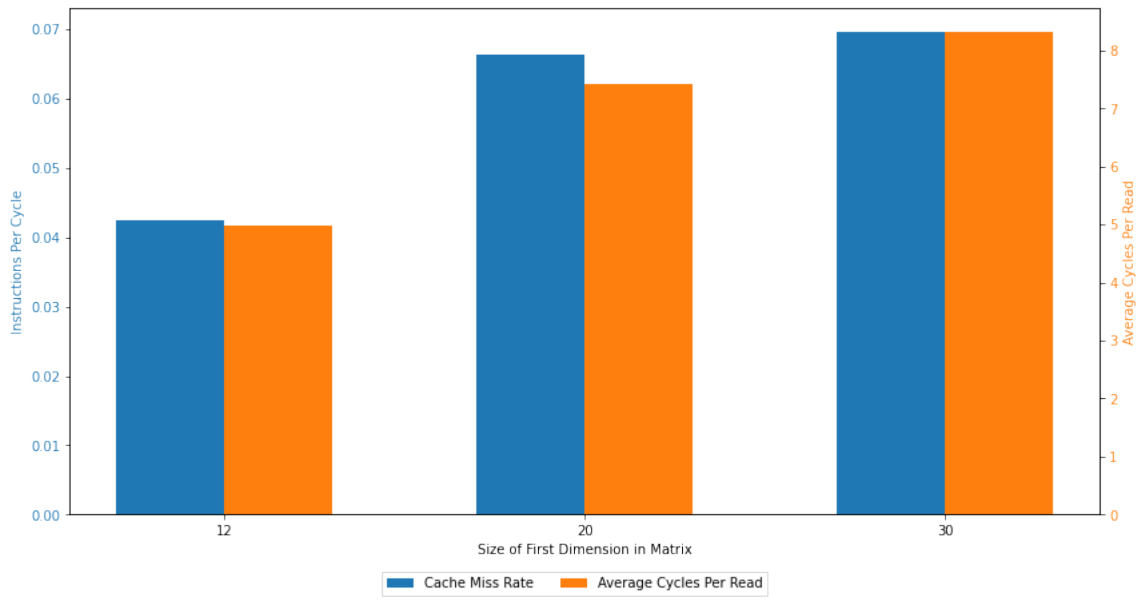


Figure 7.11: L1 Dcache Miss Rate and Average Cycles per Memory Access

This creates the opportunity for store-load forwarding and change of mis-speculation where data needs to be rolled back. The objective of this experiment is to study data dependence spectulation and effect of removing register renaming on mis-speculation, which could be extended beyond as future work.

7.5 Comparison Between Theoretical and Actual Performance Improvement

Now we combine the results of theoretical performance improvement with experimental results from computing arithmetic sequence and Fibonacci number. In Figure 7.12, the upper bound is when CPI before branch instruction is 0.5 since the simulator's fetch/decode/issue width is 2. The highest performance attainable is 2 instructions per cycle. The lower bound is when CPI before branch is 9, assuming all instructions are executed sequentially. It is hard to decide what CPI before branch is since instructions are fetched at a rate of 2 instrutions per cycle. However the pipeline is stalled due to L1 instruction cache miss and other stalls due to data dependency problems. We use the average CPI of both programs when runnning at their highest iterations, 1000 for arithmetic sequence and 50 for Fibonacci number, where the approximate stands at 1.4 CPI. That is when misprediction rate is lowest in both cases.
This result does not include any performance benchmarking from matrix multiplication since in matrix multiplication, there is load/store memory instructions. The load and store latency is much higher when L1 data cache miss or even, in rare case, L2 data cache miss occurs. It would make determination of CPI before branch more unpredictable.

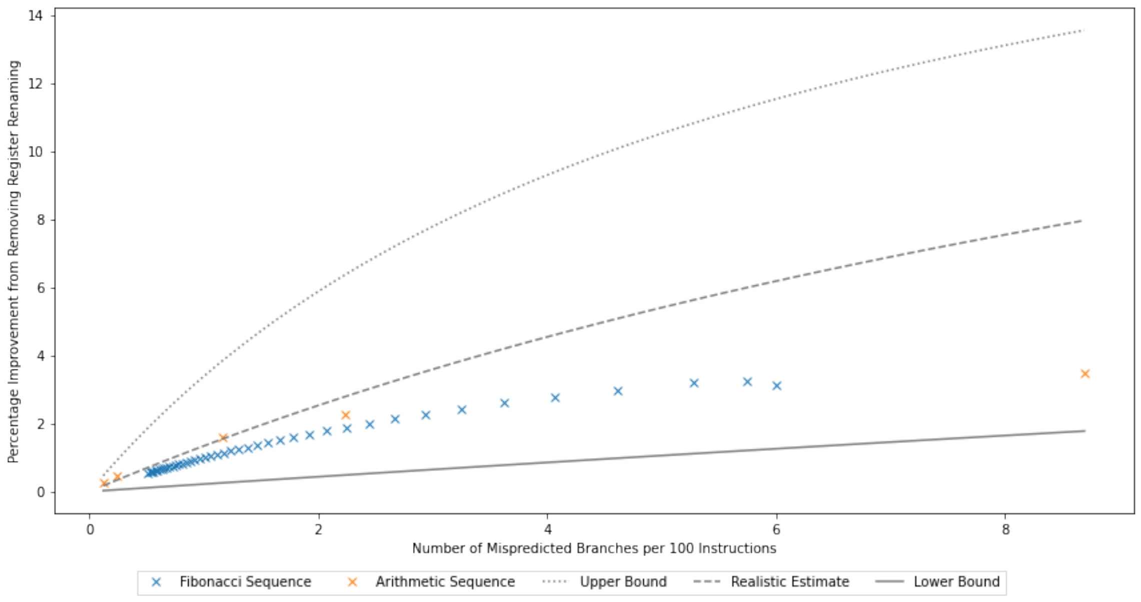


Figure 7.12: Comparison Between Theoretical and Actual Percentage Performance Improvement For Both Computing Arithmetic Sequence and Fibonacci Number

As seen in Figure 7.12, both Fibonacci Number and Arithmetic Sequence follow the trend of theoretical bound at a low number of mispredicted branches per 100 instructions. When the branch predictor becomes quite inefficient and the number of mispredicted branches per 100 instructions increases to more than 5%, meaning there are 5 mispredicted branches out of 100 instructions, percentage improvement in performance becomes stagnates and decreases afterwards. This observation can be explained by the nature of the code where most of these happen when total number of instructions is small and branch misprediction rate is high. The program itself is experiencing a lot of instruction cache misses and stalls from data dependency problems. Therefore the CPI before branch is much lower than the estimated value of 1.4. The percentage improve in performance peaks around 3% in our experimental results.

7.6 Conclusion

Throughout compiling C programs, investigating the internal structure of the gem5 out-of-order processor and runnning experiments, it has been proven that the new ISA design fulfills sufficient functionality for programs to be compiled and run on. However, it is also found that some competitive edges of this new STRAIGHT architecture cannot be exhibited and quantified in gem5

processor, a modified version of RISC processor which was made to model conventional RISC instruction set. Another drawback is compilation flow is not optimised. For example, in the assembly code shown in Figure 14 and 18, there are multiple repetitive register move and nop instructions in order to pad the instruction length to fix the distance between two merging blocks. It could have been optimised away for performance.
The next chapter will focus on analysing the actual unexihibited advantages of microarchitectural simplification in new STRAIGHT architecture from conventional RISC achitecture.

Chapter 8

Impact on Microarchitectural Simplification

Apart from the performance improvement seen when branch misprediction and instruction squashing occur, there are other benefits that come with the microarchitectural simplification and pipeline reduction, such as hardware logic simplification. This chapter will provide a more comprehensive qualitative analysis about impacts of removing register renaming on the microarchitecture.

8.1 Performance Benefits from Instruction Recovery

As discussed in Chapter 7, the new architecture experiences a lower branch misprediction penalty from a reduced pipeline stage. The processor in gem5 has a 2-cycle stage for register renaming, as seen in Figure 8.1. Without register renaming, the branch misprediction can be detected 2 cycles earlier and fewer in-flight instructions need to squashed. It improves both performance and power efficiency of the processor. The improvement is more significant when the branch misprediction rate is high in the program in cases of, for example, a less efficient branch predictor or source code with a lot of *switch* statements or branches.

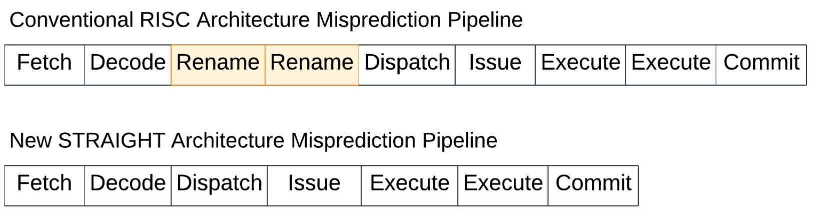

Conventional RISC Architecture Misprediction Pipeline
Fetch
Decode
Rename
Rename
Dispatch
Issue
Execute
Execute
Commit
New STRAIGHT Architecture Misprediction Pipeline
Fetch
Decode
Dispatch
Issue
Execute
Execute
Commit

Figure 8.1: Misprediction Pipeline of Conventional RISC and STRAIGHT Architecture Modelled in gem5

8.2 Hardware Logic Simplification from Register Renaming

In register renaming, there are a few components to track the renamed registers and their renaming history. To model the architecture of no register renaming stage in gem5, we adopted a fixed 1-to-1 mapping in *rename()* method to always map the logical register to the same physical register. In the new architecture design, some of the complicated hardware logic adopted in the conventional processor is not needed. We will discuss some logic used in gem5 processor, based on Alpha 21264 superscalar microprocessor. This section will focus on a few key components which can be simplified or removed.

8.2.1 Rename Map

Rename map stores the mapping between logical and physical registers, the fundamental component of register remapping. We need to look up twice for source register in rename map and record a new entry for destination register for every R-type instruction. Apart from rename map, we also need a scoreboard to keep track of which physical register is free to allocate and which one is not. At each instruction rename, the respective scoreboard sets the register as unready when destination register is renamed and ready after the instruction has finished executing. A free list is also needed to assign free physical register to.

Rename Map Update Upon New Instruction

As shown in Figure 8.2, the processor will look up in the rename map for physical register of the source operands. To assgin a new physical register to the destination operand, a lookup in the freelist is executed to determine the next free physical register the destination register can be assigned to. The scoreboard is then updated, marking the physical register as unready and the register is also removed from the free list. A new enry is entered in the rename map table as well as the rename history table(to be discussed in the next section). For the new architecture, we can remove the hardware of scoreboard, free list and rename table.

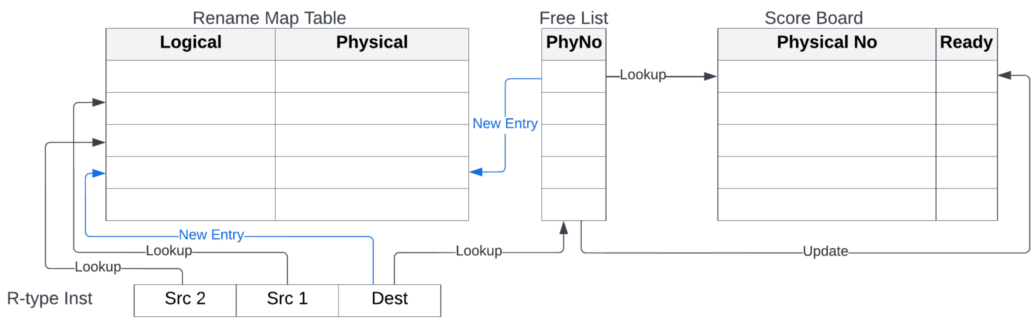


Figure 8.2: Rename Table Lookup and Update Upon New Instruction

8.2.2 Rename History Table

Not only do we need a rename map table, rename history table is also required to record rename history. It holds each logical number with the new physical register mapping as well as the previous one. It carries two responsibilities:

1. Undo register mapping when instructions are squashed by looking up the old physical register the logical register is mapped to;
2. Free up older rename of logical registers when the associated instructions are committed.

Instruction Commit

As shown in Figure 8.3, when an instruction is committed, we need to free the previous physical register mapping and delete the entry in the rename history table. The newly freed physical register will then be added to the free list to be mapped to other logical registers. Upon committing instruction in the new architecture, physical registers do not need to be freed since the mapping is fixed.

Instruction Squashing

As shown in Figure 8.4, when there is a misprediction in branch or other speculative instruction, instructions need to be squashed and the rename table needs to be mapped. The rename map will first set the renaming of the logical register to the previous renamed physical register which can

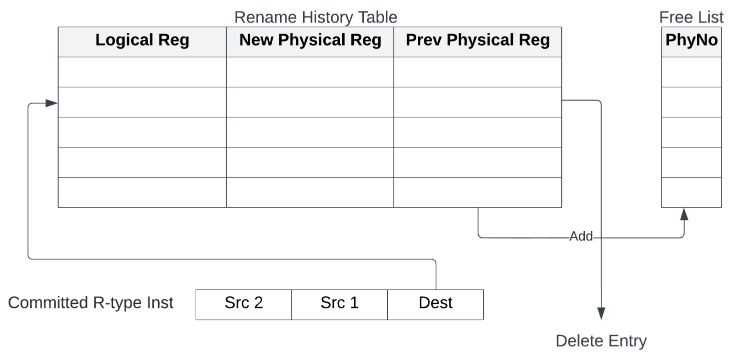


Figure 8.3: Rename History Table Update Upon committing Instruction

be found in the rename history table. It will then add the renamed physical register on the free list, a list that holds the unused physical register which rename can assign upon arrival of new instructions. Squashing instructions becomes much simpler due to the lack of register renaming for unmapping logical to physical registers.

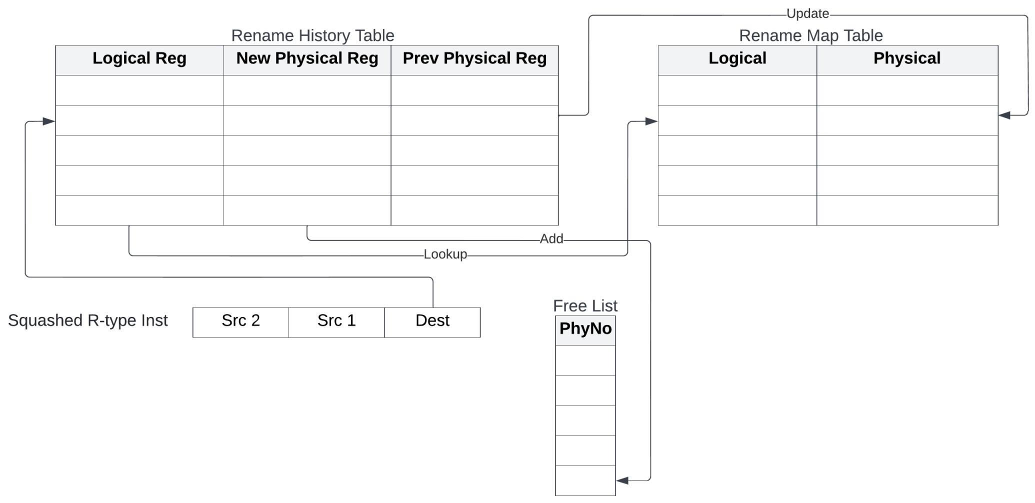


Figure 8.4: Rename History Table Update Upon Squashing Instruction

8.2.3 Dependency Graph in Instruction Queue(IQ)

Dependency Graph is a hardware component in instruction queue to keep track of the producing and consuming registers of each physical register. For each instruction, the source register will be looked up in the rename map for the physical register and the associated instruction will be added to the dependency graph as a consumer of the register. The destination register will be looked up for the physical register number and add the instruction as a producer. At write back of execution stage when an instruction finishes execution, the dependency will be removed from the graph.

Instruction Issuing

To construct the dependency graph in the conventional architecture, we need to do one extra lookup in the rename map for every entry. Three lookups are required for R-type instruction, 2 for I-type and S-type and 1 for U type, which makes rename map table more overloaded. In the new STRAIGHT achitecture, no lookup in the rename map table is required to update dependency graph.

Instruction Squashing

To squash the instructions, dependencies should be removed from dependency graph. All logical registers will be looked up in the rename map table to obtain its physical registers. It adds more workload to the rename map table. Likewise, in the new STRAIGHT architecture, the extra translation step is not required.

8.2.4 Reorder Buffer(ROB)

Reorder Buffer(ROB) tracks the state of all inflight instructions in the pipeline in their fetched order. When the head instruction of the ROB is done executing, ROB will start committing the instruction and remove its entry. When a misprediction is detected, ROB will squash all inflight instructions after the squashed instruction and start re-fetching instructions from the correct PC address.

Instruction Retirement

To retire an instruction in RISC microprocessor, we first go to reorder buffer and start committing from the head of ROB. After that, we would free up the old rename of logical registers in the rename history table as discussed in the previous sections.

In the new STRAIGHT architecture, no update in rename map and rename history is needed. To commit an instruction in ROB, as seen in Figure 8.5, we commit the instruction by removing the head of the ROB, which can be done by shifting the head pointer up by one instruction and when the instruction is confirmed committed, we increment the *headRP* counter by 1 to keep track of RP value in the register being committed.

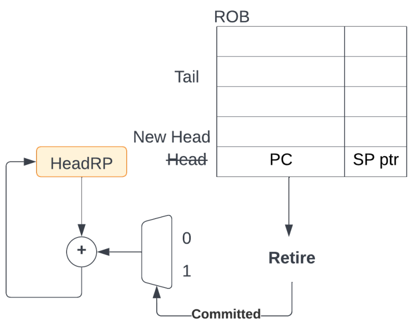


Figure 8.5: Reorder Buffer Update Upon Committing Instruction

Instruction Recovery

When there is a misprediction in branch or other speculative instruction, the recovery mechanism is very complicated in the conventional architecture. The processor will send recovery information to all previous stages to squash and clear all relevant entries in rename map, rename history and dependency graph.

In the new architecture, recovery mechanism, as seen in Figure 8.6, has become much simpler since no rename unmapping of logical registers is needed. ROB is updated by moving the tail pointer from the latest added instruction to the instruction before the squashed instruction. Register file does not need to be updated and the only thing we need to do is to roll back the RP counter, which is maintained in ROB and the registers after the RP will be overwritten by the subsequent correct instructions. RP value can be calculated from adding the relative position from the head of ROB to *headRP*.

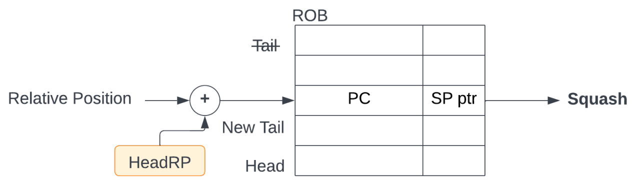


Figure 8.6: Reorder Buffer Update Upon Squashing Instruction

8.3 Power Consumption and Other Performance Improvement from Removing Register Renaming

In this section, we will mainly put together past literature and related work to shed lights on benefits on energy consumption and other potential performance improvement from this hardware simplification.

8.3.1 Power Consumption

In STRAIGHT research [9], an RTL model is developed for their architecture to measure the power reduction. It is measured that 5.7% of the total power consumption belongs to register renaming logic. As a trade off, there is a slight increase in power consumption, up to 18% for the increase in size of the register file and up to 5% for other modules. However, this increase is cancelled out by an increase in performance and shorter execution time in total energy.

8.3.2 Increasing Issue Width and Clock Rate

As discussed in the previous section, rename map is accessed up to 3 times for renaming an instruction, 2 reads and 1 write for a R-type instruction. As discussed in Section 3.1, complexity of logic gates increases exponentially with issue width, $\mathcal{O}(n^k)$ and that of circuit delay increases quadratically, $\mathcal{O}(k^2 logn)$, where k is the issue width and n is the number of instructions in the instruction set [18]. Register rename map is on the processor's critical path and therefore limits the clock frequency. Simplifying this hardware component on the critical path will lead to an increase in clock frequency, or as a tradeoff, an increase in issue width.

8.4 Conclusion

In this chapter, we mainly talk about the hardware simplifications of the new STRAIGHT architecture as opposed to the conventional RISC architecture. Performance improvement comes from shorter pipeline stage by removing register renaming. Power improvement and potential increase in issue width and clock rate comes from these hardware simplifications associated with logical to physical register mapping. These would have been better modelled out in hardware simulation where number of logic can be measured.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This project uses the STRAIGHT architecture, an architecture where data dependencies are determined statically and encoded in the instructions to remove register renaming. We modify the RISC-V instruction set for the new ISA, describe the new ISA semantics in SAIL, provide compilation, assembly and linking support. To understand the performance benefits from the microarchitectural simplification, we model the microarchitecture in out-of-order processor in gem5 and simulate the architecture with benchmark programs. It is found that the performance has an up to 3.5% improvement from smaller branch misprediction penalty, enabled by removing the register renaming stage in the out-of-order processor. Furthermore, we analyse other benefits, such as, energy consumption and potential wider instruction window size, from the hardware simplification.

9.2 Limitations

Due to the timeline of the project, design choices are made in favor of producing the minimal viable product and shortest round trip to explore the performance and microarchitectural impacts of the new architecture while omitting implementing detailed designs to fulfill the full functionality. Here lists some of the major limitations in the project.

9.2.1 Compiler Efficiency

Assumption is made prior to running the experimental results that the new STRAIGHT architecture compiler is as efficient as conventional RISC architecture. However, in reality, the existing LLVM backend compiler does generates more executed instructions for the same program. It is reported that STRAIGHT compiler generates redundant *RMOV* instructions to pad the distances between two merging blocks, such as *PHI* node. A conventional RISCV compiler generates much more efficient and compact code for the programs we run in Chapter 7.
To run the control experiment, we simply assume the compiler will be as efficient as conventional RISCV compiler so the result can focus on the effect of reducing the pipeline stage on the performance. However, since we need to pad the distances, as discussed is Section 3.4.3, which is an extra step compared to the conventional architecture, the compiler may not generate the same number of instructions.

9.2.2 Scale and Coverage of Experiments

Since some of the features are not fully implemented in gem5 and compiler backend, such as support for floating point and vector as well as special register – *Stack Pointer*, it limits the variety of programs can be simulated in the microarchitectural simulator. Therefore the coverage of the performance study is mainly focused on branch misprediction and cache efficiency. However the coverage of the microarchitectural study could extend beyond given more work is done in compiler support and microarchitectural simulator. For example, we could study the performance differences in function calls, global versus local data section and so on.

9.2.3 Power and Hardware Simulation

gem5 is a cycle-accurate simulator to simulate the out-of-order processor performance and memory efficiency at different memory hierarchies. However, one of the limitations of gem5 is it is incapable of visualising the architectural design and the hardware simplification as we discussed in Chapter 8. Therefore, a lot of microarchitectural simplification cannot be quantified and compared. More research into transistor count and power consumption is required before obtaining the full picture of the architectural benefits from the new architecture.

In addition, understanding the hardware simplification will also give a good estimate on how much the processor can increase its clock rate and widen its fetch/dispatch/issue width since the critical path determining the clock rate often lies on the register renaming path.

9.3 Future Work

Here lists some main extensions and areas of study to extend beyond the current project work.

9.3.1 gem5 Extensions

As discussed in the above subsection, there are a few limitations over the testbenches we can simulate in gem5 simulator mainly due to lack of features to be implemented in gem5. To extend on the current work, more gem5 support for the new architecture needs to be supported including providing a special register – *Stack Pointer(SP)* and simulating SP-related instructions including, *SPADD*, *SPLD* and *SPST*. By providing this support, we can use global stack pointer to allocate local memory at runtime and execute function calls.

Another important feature to extend on gem5 is floating point and vector support in the new architecture. The performance improvement would be better understood more if larger testbenches with floating point support can be run so it provides a more comprehensive overview.

9.3.2 Experiments

Our experimental trials focus on instructions with mathematical computation, mainly integer addition and subtraction, and branches. The advantage of running programs is the behaviour of each instruction, number of cycles taken per instruction, is consistent. Therefore it makes our analsys of performance benchmarking easier from branch misprediction penalty. However, it prevents us from seeing the full picture and behaviour of real programs. To extend beyond the current work, more testbenches should be simulated to study the behaviour of branch misprediction and branch rate of a program on program execution time, so to get a better estimate how much performance improvement this architectural simplification will bring to real programs.

Another experiment to be trialed beyond the current work is to study the performance benefits this architecture brings to data dependence mis-speculation, such as load-store forwarding prediction.

9.3.3 Hardware Simulation

In this project, we mainly focus on software cycle-accurate simulation of the new architecture. Power reduction and critical path latency reduction, as discussed in Section 8.3, brought by hardware simplification is not quantitatively measured. Since register renaming is reportedly one of the most power-consuming components in the microprocessor design, we believe that the architectural benefits in terms of energy consumption brought by removing register renaming is not fully explored and exhibited. In addition, by removing the component on the crital path of the pipelne, more performance benefits such as over-clocking and increase instruction window size can be made possible. The upper bound and percentage in improvement in issue width or clock rate, and their respective power consumption under high issue width/clock rate is yet to be found out. Another direction of study is to simulate the microarchitecture in RTL design and understand the microprocessor from hardware point of view.

9.3.4 Further Reducing Misprediction Penalty Through Data Dependencies

Reducing misprediction penalty is a popular area of study in microprocessor design. A large number of literatures, discussed in Chapter 3, have suggested the idea of selective instruction recovery, avoiding squashing instructions that are independent of the control flow. However, these schemes in conventional RISC architecture often experience implementation difficulties as the instruction window size becomes larger.

One of the implementation techniques is called Register Integration [22], as discussed in Section 3.2.1. It requires lookup and update in the rename table, to reuse the instruction independent of the squashed instruction.

The new architecture could use the idea of Register Integration due to its simple instruction squashing pathway and readily available data dependencies encoded in the instructions, making it much easier to trace the data independent of the squashed instruction for result reuse.

One of the biggest advantages from the new architecture compared to the conventional RISC architecture is as we trace the squashed instructions, we do not need to go to register renaming. We simply trace the data flow, mark the independent instruction from the control and data flow in the register file, and avoid re-fetch/decode/execution of instructions that already write the correct result to the respective physical register. The next step would be to integrate register integration mechanism into the architectural simulator and study how much squashed instructions are reused.

Similar discussion has been addressed in EDGE architecture [23] where the EDGE instruction set encodes the data dependence in the instructions. The compiler computes the data flow graph and statically schedules instructions to ALUs using the data flow graph to simplify register file use and rename logic. By giving compiler more control through data flow analysis, this architecture could achieve a more distributed, scalable selective recovery scheme in case of mis-speculation, which solves the performance bottleneck in scalability of selective recovery schemes in conventional architecture, further reducing the performance loss brought by the misprediction penalty.

Chapter 10

Ethical Considerations

10.1 Legal Issues

This project will be built upon a few open-source projects, including LLVM, SAIL, gem5 and STRAIGHT backend. Permissions are given to use the software and modify the files under their licenses. Permission from STRAIGHT backend is given to use their software tool in the project. Extra attention is given when using the source code to make sure no copyrights are violated.

10.2 Other Issues

This project is considered safe with ethical issues since no data usage is involved in this project. The purpose of the project is focused on low-level architecture and any prototype or source code is not easily misused by malicious users for any illegal or harmful intentions.

Bibliography

[1] Cotofana S, Vassiliadis S. On the design complexity of the issue logic of superscalar machines. In: Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204). vol. 1; 1998. p. 277–284 vol.1.

[2] Shen JP, Lipasti MH. Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill Higher Education; 2002. Available from: https://books.google.co.uk/books?id=VIWLAAAACAAJ.

[3] Petit S, Ubal R, Sahuquillo J, López P. Efficient Register Renaming and Recovery for High-Performance Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2014;22(7):1506–1514.

[4] Smith J, Pleszkun A. Implementation of Precise Interrupts in Pipelined Processors. vol. 13; 1985. p. 36–44.

[5] pes20. Sail; 2019. Http://www.cl.cam.ac.uk/ pes20/sail/. Available from: http://www.cl.cam.ac.uk/~pes20/sail/.

[6] Lowe-Power J. TraceCPU; 2021. Available from: https://www.gem5.org/documentation/general_docs/cpu_models/TraceCPU.

[7] Gandhi A, Akkary H, Srinivasan ST. Reducing branch misprediction penalty via selective branch recovery. In: 10th International Symposium on High Performance Computer Architecture (HPCA'04); 2004. p. 254–264.

[8] Burger D, Keckler SW, McKinley KS, Dahlin M, John LK, Lin C, et al. Scaling to the end of silicon with EDGE architectures. Computer. 2004;37(7):44–55.

[9] Irie H, Koizumi T, Fukuda A, Akaki S, Nakae S, Bessho Y, et al. STRAIGHT: Hazardless Processor Architecture Without Register Renaming. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2018. p. 121–133.

[10] Mitsuno S, Kadomoto J, Koizumi T, Shioya R, Irie H, Sakai S. A High-Performance Out-of-Order Soft Processor Without Register Renaming. In: 2020 30th International Conference on Field-Programmable Logic and Applications (FPL); 2020. p. 73–78.

[11] Harris DM, Harris SL. 7 - Microarchitecture: With contributions from Matthew Watkins. In: Harris DM, Harris SL, editors. Digital Design and Computer Architecture (Second Edition). second edition ed. Boston: Morgan Kaufmann; 2013. p. 370–473. Available from: https://www.sciencedirect.com/science/article/pii/B9780123944245000070.

[12] Hennessy JL, Patterson DA. Computer Architecture, Fifth Edition: A Quantitative Approach. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2011.

[13] Moshovos A. Checkpointing Alternatives for High Performance, Power-Aware Processors. In: Proceedings of the 2003 International Symposium on Low Power Electronics and Design. ISLPED '03. New York, NY, USA: Association for Computing Machinery; 2003. p. 318–321. Available from: https://doi.org/10.1145/871506.871585.

[14] Gray KE, Kernels G, Mulligan D, Pulte C, Sarkar S, Sewell P. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2015. p. 635–646.

[15] Wikipedia. LLVM; 2021. Available from: https://en.wikipedia.org/wiki/LLVM.

[16] Shigenobu K, Ootsu K, Ohkawa T, Yokota T. A Translation Method of ARM Machine Code to LLVM-IR for Binary Code Parallelization and Optimization. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR); 2017. p. 575–579.

[17] Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, et al. The Gem5 Simulator. SIGARCH Comput Archit News. 2011 aug;39(2):1–7. Available from: https://doi.org/10.1145/2024716.2024718.

[18] Cotofana S, Vassiliadis S. On the design complexity of the issue logic of superscalar machines. In: Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204). vol. 1; 1998. p. 277–284 vol.1.

[19] Safi E, Moshovos A, Veneris A. Two-Stage, Pipelined Register Renaming. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2011;19(10):1926–1931.

[20] Vajapeyam S, Mitra T. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. SIGARCH Comput Archit News. 1997 may;25(2):1–12. Available from: https://doi.org/10.1145/384286.264119.

[21] Chou Y, Fung J, Shen JP. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. In: Proceedings of the 13th International Conference on Supercomputing. ICS '99. New York, NY, USA: Association for Computing Machinery; 1999. p. 109–118. Available from: https://doi.org/10.1145/305138.305175.

[22] Roth A, Sohi GS. Register integration: a simple and efficient implementation of squash reuse. In: Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000; 2000. p. 223–234.

[23] Desikan R, Sethumadhavan S, Burger D, Keckler SW. Scalable Selective Re-Execution for EDGE Architectures. SIGARCH Comput Archit News. 2004 oct;32(5):120–132. Available from: https://doi.org/10.1145/1037947.1024408.

[24] Koizumi T, Sugita S, Shioya R, Kadomoto J, Irie H, Sakai S. Compiling and Optimizing Real-world Programs for STRAIGHT ISA. In: 2021 IEEE 39th International Conference on Computer Design (ICCD); 2021. p. 400–408.

[25] Kessler RE. The Alpha 21264 microprocessor. IEEE Micro. 1999;19(2):24–36.

[26] kcelebi. RISCV Assembler; 2021. Available from: https://github.com/kcelebi/riscv-assembler.

[27] Waterman EA, Asanovi´c K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. RISC-V Foundation; May 2017. Available from: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[28] straight dev. STRAIGHT env; Feb 2022. Available from: https://github.com/straight-dev/env.