Lucas Sonnabend

Adding a conditional move instruction to the RISC-V instruction set

Part II Project Dissertation

Churchill College

March 7, 2016

Proforma

Name:	**Lucas Sonnabend**
College:	**Churchill College**
Project Title:	**Adding a conditional move instruction to the RISC-V instruction set**
Examination:	**Computer Science Part II Project Dissertation, May 2015**
Word Count:	**11867**[1]
Project Originator:	Lucas Sonnabend
Supervisor:	Colin Rothwell & David Chisnall

Original aims of the project

The aim of the project is to evaluate the advantages and disadvantages of a conditional move instruction for a Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA), namely RISC-V.

The focus lies on how the conditional move instruction can reduce the number of conditional branches in the target code and how this translates into an overall speed-up of the implemented processor. I will also consider the complexity the conditional move instruction adds to the processor design.

[1] This word count was computed by `texcount -total diss.tex`

Work completed

I have extended a processor designed in bluespec to execute conditional move and debug instructions. Furthermore, I implemented several branch predictors for the processor. The RISC-V LLVM back-end was extended to generate conditional moves. In the process I fixed some bugs of the compiler. I compiled and ran tests from several benchmarks on my modified processor and I synthesized the processor to estimate its size and complexity.

Special difficulties

I started with the assumption that the RISC-V LLVM compiler back-end was fully functional. This turned out to be optimistic and I fixed some bugs in the current implementation. In the end I could not fully correct the compiler, which prevented me from compiling more elaborate benchmarks.

Declaration

I, Lucas Sonnabend of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I thank my supervisors and my director of studies for their extensive support: Colin for the help regarding any hardware related questions, David for his help and tips regarding the LLVM compiler and both of them for proof reading this dissertation and providing helpful feedback.

Chapter 1

Introduction

In this chapter I will introduce the problem that a conditional move instruction is trying to solve and what other solutions have been used in the past. I will then explain why a conditional move is advantageous and how I intend to demonstrate this.

1.1 Why is a conditional move useful?

We are always in pursuit of faster computers. As we have reached physical limits we struggle to increase the clock frequency any more. Neither can we decrease the transistor size indefinitely. To further increase the computing power, hardware architects moved to more complex designs. Pipelined processors are a well established choice.

But as the pipeline contains multiple instructions simultaneously at any time, new problems occur, such as *control hazards*. When the next instruction depends on the outcome of a previous instruction, it cannot easily be fetched before the result of the previous instruction is computed. A conditional branch is a typical example. The simplest solution is to insert *no-ops* into the pipeline or stall the pipeline until the result is known and then fetch the next instruction. This lowers the executed instructions per cycle and hence the utilization of the processor.

To overcome this particular hazard, a variety of techniques have been applied. Some designs such as MIPS processors evaluate the branch conditions at an earlier pipeline stage to reduce the number of inserted *no-ops*. This allows only simple conditions for a branch, for example comparison with zero. More complex arithmetic would increase the critical path of that pipeline stage and reduce the clock frequency of the whole pipeline. Another design choice MIPS took was executing the first instruction after a conditional branch, the *branch* *delay slot*, independent of the outcome of the branch. This is considered a questionable design choice as it exposes the processor design. A compiler can fill the *branch delay slot* with a useful instruction only up to 70% of the time in the best case[**mcfarling1986reducing**].

Branch prediction offers a different solution to increase the instructions per cycle. When a conditional branch is fetched, the processor uses additional hardware to predict the outcome of the branch and the branch target address. This can be done using the local and global history of the branches[1]. These predictors have an accuracy of 80% to 99% percent depending on their complexity[**chang1997improving**]. On the down side they add to the chip size and power consumption. Most chips nowadays use branch prediction.

The number of conditional branches in your code can also be reduced using predicated instructions (ARM) or conditional moves (ARM, MIPS, AArch64, x86). These instruction do not alter the control flow, so the succeeding instruction can be fetched immediately. Those instructions will write back their result only if a specified condition is met. If a branch can be predicted correctly using branch prediction, a conditional move instruction does not offer any further advantages. If, on the other hand, the branch is difficult to predict, the conditional move will avoid the performance penalty of a mis-predicted branch. This can save between five and 15 cycles. Often those branches depend on input data and are called *data dependent branches*. This is why contemporary RISC ISAs use conditional move instructions both for integer and floating point moves.

1.2 My approach

I will add an integer conditional move instruction to the RISC-V instruction set, which is similar to the MIPS ISAs. According to Hennessy & Patterson [**HenPat2012**] about 1% of the instructions of selected integer benchmarking programs from SPECint2000 are conditional moves. For selected floating point programs from SPECfp2000 the percentage is less than 0.1%. Therefore I consider any impact of a floating point conditional move negligible for the overall performance and I will not implement those.

The conditional move shall be designed such that adding it will require a minimal amount of changes to the existing implementation of the processor. The critical path should only increase minimally if at all.

[1] A global history stores the branch outcomes of the last executions for a particular branch, the global history stores the last outcomes of any branch.

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1-P) + P \times \frac{1}{\text{Speedup}_{\text{cmov}}}}$$

Figure 1.1: Amdahl's law; P = fraction for which conditional move instructions are used

The performance changes of conditional moves shall be compared with other possible solutions that tackle control flow hazards. I will limit this to branch prediction as it is a very general and widely applied technique.

By Amdahl's law (see figure **??**) the overall speedup that can be achieved depends on the speedup of the conditional move and the percentage P the conditional move is used. If we assume that about every 100^{th} instruction is a conditional move[2] we get an an upper limit of the overall speedup of 1%. This is only achievable with a large speedup provided by the conditional move itself. In other words every time a conditional move is used it has to replace a branch that would be mis-predicted, causing an expensive pipeline flush. If this is not the case the resulting overall speedup will be considerably less than 1%. Instead we can also use the percentage of cycles spend executing those branch instructions which we intend to replace with a conditional move. Then the estimated speedup might also be higher: Consider an ideal six stage pipeline where each instruction needs on average one cycle to execute. Assuming a branch mis-prediction causes the worst possible mis-prediction penalty of six cycles. With the same dynamic frequency, about 6% of all cycles will be spent on branches replaceable by conditional moves. This gives a much higher upper bound on performance improvements of about 6%. As this case is highly idealized, such a percentage is of course unrealistic.

If furthermore a conditional move instruction prevents an instruction cache miss, its potential speedup for the program might also improve. Because I only expect small if-statements and hence very short branches to be converted into conditional moves, I rarely expect this to happen[3]. Therefore this case will be ignored for the performance prediction and the evaluation.

Taking into account both arguments I will expect a performance increase of about the same percentage as the dynamic frequency of the conditional move instruction for a given program.

[2] This value is likely to vary depending on the application and even between functions and loops within the same program.

[3] As caches exploit spatial locality of references I expect the branch target of a short branch to be in the cache.

Such a small speedup is only beneficial if it does not come with a significant cost. In my case this would be an increased area for the chip and a reduced frequency at which the processor can be run. But as I expect the changes of the processor to be small, I only expect a negligible increase in are and decrease in frequency.

1.3 Evaluating advantages and disadvantages

I will look at the assembly code generated by the extended LLVM compiler to estimate how often conditional branch instructions are used. This will be compared with the MIPS assembly code generated by the same program.

I will use different C benchmarks to test the performance of the initial and the modified version of the processor. I will count the number of branches and conditional moves executed dynamically, as well as branch prediction misses to evaluate the prediction accuracy. The performance will be measured in clock cycles needed to execute and Instructions Per Cycle (IPC). To determine the costs of a conditional move instruction I will attempt to synthesize the processor design on an FPGA and compare the resources needed by each implementation.

The results shall answer the question whether it is advisable to add a conditional move to the RISC-V instruction set and similar RISC ISAs.

Chapter 2

Preparation

In this chapter I investigate how a conditional move has been implemented in other RISC instruction sets. This leads to a description of the RISC-V instruction set and why it does not implement a conditional move. Then I will have a closer look at the processor I plan to extend and the compiler I will modify. The benchmark suites used for evaluation are also introduced. It closes with a description of my software engineering approach to the project and the tools I could use.

2.1 Conditional move in MIPS

A conditional move was introduced in the ALPHA APX ISA in 1992[**sites1993alpha**] and in the MIPS IV instruction set in 1995[**MIPSIV**]. The MIPS IV ISA defined two instructions for integer registers[1]:

```
MOVZ rd, rs, rt
MOVN rd, rs, rt
```

The first one moves the content of register `rs` into register `rd` if the content of register `rt` is zero. The second one executes the same move if the register `rt` is not zero. This is often combined with the `SLT rd, rs, rt` which sets register `rd` to 1 if the content of `rs` is less than the content of `rt` and sets it to 0 otherwise. In the latest version of MIPS, MIPS64r6, this has been replaced by select instructions:

```
SELEQZ rd, rs, rt
SELNEZ rd, rs, rt
```

[1] MIPS IV also contains floating point conditional moves.

If the value in `rt` is zero – or non-zero for the second instruction – the value of `rs` is written into `rd`, otherwise `rd` is set to 0. This allows for simpler implementations compared to the conditional move, as I will explain:

Assume the architecture is designed to always write to `rd`. This is needed for register renaming[2]. It that case the conditional move needs three read ports into the register file to get the values of `rs`, `rt` and `rd`. The select instruction only requires two as it writes back zero instead of the value of `rd`. This makes the implementation simpler and may reduce the chance of a data dependency.

2.2 The RISC-V instruction set

The RISC-V instruction set is a RISC ISA that originated from a research project at the Computer Science Division of the EECS Department at the University of California, Berkeley. It aims to provide an open source ISA for both academia and industry which does not limit processor design to a particular style or implementation and which is highly extensible.

2.2.1 Instruction format

The instruction set is split into several sets according to functionality and width of the integer registers. It provides instructions for 32-bit and 64-bit integer registers (and therefore 32-bit and 64-bit address spaces) and a 128-bit version is being planned. I will only work with the version called RV64G. Its registers are 64-bit wide and it contains all ‘standard extensions’, supporting integer multiplication and division, floating point operations of single and double precision and instructions to perform atomic memory reads and writes.

According to the RISC-V instruction set manual[**RISCV˙MAN**], RV64G uses only 32-bit long, fixed size instructions. Other instruction lengths are supported as well, so that a few bits of each instruction have to be used to determine the instruction length, as described in figure **??**.

Figure 2.1: RISC-V instruction lengths

The programming model for RV64G is a register file with 32 64-bit registers. Register `x0` contains constant 0 and registers `x1-r31` are general purpose registers. The only other user visible register is the program counter `pc`.

[2] The processor dynamically maps the registers of an instruction to the physical registers to avoid false data dependencies.

There are different instruction formats defined for the core set as described in Figure **??**. The design is chosen so that the identifier of the accessed registers (rd: destination register, rs1: first source register, rs2: second source register), the *opcode* and the function code and the immediate are always at the same place. This allows fast instruction decoding typical for RISC ISAs.

Figure 2.2: RISC-V base instruction formats

2.2.2 Why does it not contain a conditional move?

Conditional move instructions have not been included in the RISC-V instruction set as they are considered difficult to use in combination with exceptions. Consider the following example:

```
if(c != 0) {
  a = b / c;
}
```

Naively, this could generate the following assembly code:

```
div t1, b, c
cmovnz a, t1, c
```

Then an divide-by-zero exception may be caused although the division should not be executed according to the C code.

Another argument against it was the added complexity to a processor. Assuming that unpredictable branches are rare given today's techniques of branch prediction, it was assumed that conditional moves add little to no performance improvements[**RISCV˙MAN**]. This would not justify the complexity to support them and hence they were not included in the instruction set. Furthermore the authors argued that it would be to much overhead for simple microcontrollers to implement a conditional move.

2.3 The FLUTE RISC-V implementation

FLUTE is an implementation of the RISC-V user level instruction set in Bluespec SystemVerilog, currently under development by Bluespec Inc.. It is designed as a 6-stage pipeline. The code for the processor is a snapshot and still under development with the aim to eventually provide a small but extensible implementation of the RISC-V ISA. Currently the codebase contains about 21,000 lines of source code written in bluespec and C.

2.3.1 Processor pipeline

Figure 2.3: The FLUTE pipeline

Each pipeline stage is written as an independent module in Bluespec. The stages are:

- Instruction Fetch: A memory request is send to the instruction cache to fetch the current instruction and the address of the next instruction address is determined using the current Program Counter (PC) and the branch predictor.
- Instruction Decode: The memory response is received and the instruction is decoded into control tokens. They specify the pipeline behaviour such as if and how many source and destination registers are used.
- Register Reservation: This is the only stage that can access the register file. If the registers used by the instruction are free, i.e. no other instruction will write to it, their content is read and the destination register is reserved to write to it. Otherwise the instruction stalls until the register is freed from a previous reservation.
- Execution: Simple integer Arithmetic Logic Unit (ALU) operations are executed in this stage. More sophisticated operations such as floating point operations are dispatched to separate units. Memory accesses are senr to the speculative memory buffer that sits in front of the data cache.
- Receive Response: If the execution stage called one of the separate executions units this stage blocks until a response is received. Otherwise the data from stage 4 is passed through.
- Commit: The commit stage checks whether any exceptions have been raised during the execution of the instruction. In the case of a speculatively executed instruction it is now known whether the speculation was correct or not and the branch predictor is updated. If necessary, the register file is updated and the destination register freed by signalling stage 3. Similarly, the speculative memory buffer is signalled in case of a retired store instruction.

The processor is still under development and its design is straight forward. For now it can fetch and issue[3] only up to a single instruction per cycle. The

[3] To issue means to start execution of an instruction, i.e. the transition from stage 3 to stage 4.

actual number is lower due to data hazard stalls and nullifying mis-predicted instructions. A feed forward path to avoid certain data hazards is also missing. On the other hand the processor already supports multiple speculative instructions in flight. They will be nullified in the commit stage if the speculation turns out to be incorrect. Overall, the pipeline design is quite extensible, so that it could be expanded to fetch and decode as well as issue multiple instructions per cycle.

Besides the FLUTE processor, a verifier called CISSR written in C was provided. On simulation both FLUTE and CISSR run in parallel and all committed instructions are sent to the verifier. If the processor and the verifier produce different processor states, execution will stop with an exception.

2.3.2 Branch prediction

Initially FLUTE used a basic branch predictor based on a small direct mapped Branch Target Buffer (BTB) with only eight entries. Each entry consisted of the current PC, the next PC and a valid bit. Every PC is fed into the branch predictor and its least significant bits are used to index the BTB. If there is no valid entry in the BTB then the current PC is incremented by four. Otherwise the PC is compared with the entry in the BTB and if they match the next PC of the entry is used as the predicted instruction address. All branch instructions that don't have a valid entry in the BTB are implicitly predicted non-taken. If a predicted non-taken branch has been taken, it is inserted into the BTB and set valid. If a predicted taken branch has not been taken, the corresponding entry is set invalid. The branch-penalty for mis-predicted branches is six cycles.

2.4 The RISC-V LLVM

LLVM is a compiler collection. It started as a research project in 2000 and by now it is widely used in industry[4].

The compilation is split into 3 phases: a front-end compiles the source code into an intermediate representation. The optimisation stage tries to optimise the code using several passes with different optimisation algorithms. The back-end converts the intermediate representation into machine code. I will use the Clang front-end to compile C code. The optimisation passes I use are part of the LLVM core. The back-end is the only architecture specific part and hence the only part

[4] Apple for example uses it as their standard C/C++ compiler in their development tools[**AppleXCode**].

that needs changing as the instruction set is modified. A RISC-V version of the LLVM is already freely available[**riscvWebsite**].

The input to the back-end is the program described in LLVM's own intermediate language. Each function in the intermediate representation is converted into a directed acyclic graph (DAG) called *SelectionDAG*. These graphs show the data flow within a *basic block*[5]. Nodes define operators, leaves are operands and the root node of the DAG represent the whole *basic block* or function. The LLVM back-end performs several passes of optimisations on this graph. At this stage the nodes still represent instructions that resemble instructions of the intermediate language. Now the nodes are replaced by nodes that represent actual instructions of the target machine. This can mean that one node is replaced by another node with the same functionality, but also that several nodes are inserted for one or one node replaces a pattern of several nodes. This is called the selection pass. In the last pass the selection graph is transformed into assembly code.

Figure 2.4: Extract of an LLVM *SelectionDAG* including a select node

LLVM supports conditional moves natively in its intermediate language. This is represented in the *SelectionDAG* using a node called select. Figure **??** shows and explains an example of this node: If argument 0, the condition is set, then it will evaluate to argument 1, otherwise to argument 0; condition is set if argument 0 of the setcc node is less then argument 1 of setcc. If the target machine does not support conditional moves, this is replaced by a branch instruction later on, a process called instruction lowering. Otherwise conditional moves are inserted for the select statement during the selection pass.

As those tasks are quite similar the LLVM uses a target-independent or retargetable code generator for this processes. The author of a specific LLVM back-end only has to provide a description of the target language and the computational model, such as number and type of registers. This is mostly done using target description files (.td files), but for some target dependent functionality C++ code has to be provided. Figure **??** gives a simplified example of for the x86 assembly as a target. During compilation LLVM's TableGen tool will build the back-end for the target code using both the target description files and target independent functions provided by the code generator.

Therefore I will only have to modify the target description files that are provided within the RISC-V LLVM back-end and describe the RISC-V instruction set. Fortunately the RISC-V specific code of the LLVM back-end is only about

[5] A maximum sequence of consecutive instructions without any jumps or branches in between.

10,000 lines of code, which is manageable compared to the 900,000 lines of the entire RISC-V LLVM.

Figure 2.5: A simplified view of the retargetable LLVM code generator with x86 target descriptions

The RISC-V LLVM back-end only generates assembly code. This can be assembled into machine language using the GAS (GNU Assembler), which is part of binutils. A RISC-V version of the gcc including binutils is also available, including the assembler[**riscvWebsite**]. Here I will also have to extend the set of instruction definitions for the RISC-V ISA.

2.5 Benchmarks

To measure the impact of the added conditional move instruction I need performance benchmarks. These benchmarks have to be written in C to be compiled with the Clang compiler and they should represent typical applications[6]. At the same time the benchmarks have to be small enough to run to completion on the processor as it is simulated in software. This means the performance is slower by several orders of magnitude compared to an actual machine. Furthermore they must not rely heavily on standard C libraries and system calls, as no operating system is present on the FLUTE processor. Only very few of them are supported, such as `malloc` or `printf`, but no signalling or file I/O, for example. This leads to the following selection of benchmarks.

2.5.1 Dhrystone

A small synthetic benchmark developed in 1984. It tries to provide a standardised way to measure the performance of C and Ada on a certain architecture. It largely avoids system calls to not measure the performance of the operating system. It contains operations that represent the workload of an operating system in their relative quantity which are run a certain number of iterations(10,000 iterations are recommended)[**weicker1984dhrystone**]. Therefore it seems suitable to run on the simulated FLUTE processor. Furthermore they check the correct execution of its code and can be used to find bugs in the compiler or processor.

[6] Although it is impossible to find typical code in general, most benchmark try to contain applications that are commonly used.

2.5.2 WCET

The wcet benchmark collection is a set of 35 programs, initially developed to test analysis tools to determine the worst-case execution time. See appendix C for descriptions. It ranges over a set of application from insert sort, calculating the square root or performing a cyclic redundancy check. All programs provide example input data, which is picked to cause the longest execution time given a restricted input size. They follow only one execution path[7] and hence called single path programs.[**Gustafsson:WCET2010:Benchmarks**] Their advantage is that they do not rely on system calls and due to their small size they execute quickly even on the simulated processor. But because of their small size (between 27 and 4253 lines of code, median: 163) the results obtained with them are only of limited significance when you reason about the general case. They also don't check the computed results for correctness.

2.5.3 BEEBS

BEEBS, the Bristol/Embecosm Embedded Energy Benchmark Suite is an open benchmark suite targeted at evaluating the power consumption of embedded systems. Therefore its tests are small and do not require an operating system. The tests were selected from different benchmark suites to represent real live applications relevant to embedded processors[**pallister2013beebs**]. Dhrystone and some WCET test are contained in BEEBS, but I will focus on the ones exclusive to BEEBS, leaving me with an additional 47 benchmark tests.

2.5.4 SPECint2006

These are standard benchmarks and their results have the most significance. Unfortunately they are meant to be run on top of an operating system and rely on system calls the execution environment of the processor does not provide.

I will attempt to modify the bzip2 benchmark. I hope that the compression code will lend itself to some optimisation with conditional move instructions and it only requires file I/O system calls, which can be circumvented by loaded the test data directly into memory.

[7] The input data is static so that the entire execution path is fixed. Running it several times will have exactly the same behaviour in terms of branches.

2.5.5 Other tests

The FLUTE processor suite provides simple C programs to test the functionality and correctness of it. One of them is called `filter`, which applies a sharpening 3 × 3 filter to a dataset, which is a randomly generated 32 × 32 pixel image. The programs are comparable to the WCET benchmarks in size. They have the similar shortcomings as the WCET benchmarks but they test the computed results for correctness.

2.6 Software engineering approach

As my project splits nicely into independent tasks I will tackle each of it separately. The ISA extended with the conditional move will serve as an interface between the LLVM back-end, the GNU assembler and the processor. Each of the tasks will be tested separately but building on the result of the previous tasks. Hence my approach will be almost waterfall like: each sub-tasks has to adhere its specifications in order to work with the others. Due to the modular approach the implementation of each task might be subject to change. The tasks are divided as follows:

- Extend the processor with a conditional move. The modified pipeline stages will be tested separately for functionality.
- Change the GNU assembler to translate an assembly conditional move instruction into machine code. The correctness of the code is tested on the processor.
- Modify the LLVM RISC-V back-end to produce conditional move instructions. The correctness is checked by inspecting intermediate representations of the code and using c-programs assembled with the modified assembler and run on the processor.
- Write, collect and modify benchmarks in C that will compile with the changed LLVM compiler and run on the processor with the available system calls. If possible the correctness of their results is checked within the benchmark.
- Implement more complex branch predictors that adhere to the interface already defined within the processor.

I will use the bluespec workbench to edit and compile the processor. To analyse the complexity of the processor I will use the Quartus v14.0 workbench provided by the computer laboratory. The RISC-V LLVM is built using the provided makefiles and clang++, and its debug version is build with cmake and ninja. I use gdb to debug LLVM. To evaluate the collected data statistically, I used R. Git is employed to back-up my work on the managed cluster service of the university and for revision control.

Chapter 3

Implementation

In this chapter I elaborate how I defined two conditional move instructions. I extended the processor and its verifier to execute these instructions and two debug instructions. I implemented two more powerful branch predictors as alternatives to conditional moves. To compile code for the processor I modified the LLVM back-end to replace certain code patterns in the intermediate language with conditional moves. I changed GCC assembler to translate the new instructions. During the compilation of benchmarks I encountered bugs in the LLVM backend and I was able to fix some of them. Fortunately the majority of benchmarks compiled successfully and could be used to collect data.

3.1 Extending the instruction set

There are several alternatives regarding the exact implementation of a conditional move. One solution would involve condition registers as found in ARM processors. The RISC-V has no condition registers but uses arithmetic comparisons for conditional branches[**RISCV˙MAN**]. I therefore consider it unreasonable to add processor state for one instruction that will make up an estimated 1% of the assembly code. Condition registers also complicate out-of-order execution as they introduce new dependencies. Although the processor for my project executes in order, I try to keep the approach of RISC-V as a universally usable instruction set.

The alternatives are conditions based on the comparison between two registers or comparing one register with zero, the approach chosen in MIPS IV. The move instruction itself requires identifiers for two registers, the source and the destination. To make this conditional on the comparison of two arbitrary registers implies using three source registers in a single instruction. This is technically possible and certain floating point instructions in the RISC-V ISA use three source registers, such as the fused multiply-add instruction which multiplies the first two sources and adds the third. But it also introduces one more data dependency which could cause a pipeline stall and requires an additional read port for the register file. This is less of an issue for complex floating point instructions which take more cycles to execute than an integer move.

I decided to adopt the R-type instruction format with only 2 source and one destination register, see figure **??**. This allows to compare the second source register with zero, the same approach as the MIPS IV instruction set[**MIPSIV**].

Figure 3.1: RISC-V R-type instruction format

With these considerations in mind I design the conditional move instructions as:

```
CMOVEZ rd, rs1, rs2      move the value from rs1 into rd if rs2 == 0
CMOVNZ rd, rs1, rs2 move the value from rs1 into rd if rs2 != 0
```

Given the restrictions on the *opcodes* and as I don't want the conditional move to conflict with instructions from the base ISA or standard extensions, I chose the *opcode* to be 0001011, one of the unused custom opcodes as seen in **??**. The value of $func7$ is defined to be 0000000. Defined values for $func3$ are 000 to encode CMOVEZ and 001 to encode CMOVNZ. All other values are illegal instructions and will cause a trap.

Figure 3.2: RISC-V allocation of *opcodes* for 32-bit instructions; the last two bits of the *opcode*, inst[1:0] are always 11 to specify the instruction length

3.2 Adding a conditional move to the processor

3.2.1 Extending the processor

To add the conditional move to the processor, I had to add its *opcode* and function code to the set of supported instructions. The next step was changing the instruction decode phase: as it uses the *opcode* to determine whether source and destination registers are needed, I had to extend the corresponding expressions. It was tested using a test harness that feeds in instruction information into stage 2 and receives the information that would be passed on to stage 3. Tests showed that if a conditional move *opcode* was fed in, the output indicated that two source and the destination register shall be used.

Stage 3, the register reservation stage, did not need changes. It uses the information from stage 2 to read the source register values and reserve the destination register for a later read.

Stage 4 was altered in two ways. First it needed to evaluate the condition. Secondly it needs to pass on the evaluated information, which is whether or not to actually execute the conditional move. The execution stage is implemented using a rule for each *opcode*. These pieces of code are executed once per clock cycle and only if specified conditions are satisfied. In this case it means the processor is in normal execution mode and the opcode received from stage 3 is that of a conditional move.

Listing 3.1: bluespec rule for executing the conditional move; extract from CPU_Stage4.bsv

```
rule rl_exec_op_CMOV(normal_exec_conds && (opcode == op_CMOV));
        report_exec_instr($format(”OP_CMOV x%0d x%0d x%0d”, rd, rs1, rs2))

        GPR_Data vo = v1;
        let m_trap = m_trap_none;
        let exec_move = False;

        if        (f3 == f3_CMOVEZ)    exec_move = (iv2 == 0);
        else if (f3 == f3_CMOVNZ)    exec_move = (iv2 != 0);
        else
            m_trap = m_trap_illegal_instr;

        finish_extended (m_trap, ?, vo, ?, pc+4, exec_move);

        if (isValid (m_trap))
            $display(”CPU_Stage4: opcode OP_CMOV: funct3 %3b not implemente
endrule
```

The code example shows in listing 3.1 the rule for the conditional move. It assigns the first operand as the value to be written back (line 4) and calculates whether the conditional move will be executed (lines 6-9). If the function value *f3* is anything different from the ones specified, it will trap[1]. The information about the trap, the value to be written back, the expected next instruction are passed

[1] Remember that all traps and exceptions are passed on to stage 6, where they are dealt with.

on to the remaining pipeline stages using the *finish_extended* function. I had to replace the initial *finish* function to pass on the flag *exec_move* that indicates whether the value will actually be written back to the register.

To pass on this flag I extended the data-types used to pass information between pipeline stages 4 and 5 and 5 and 6. Stage 5 needed no additional changes. I added an additional check to stage 6: If the execution flag from stage 4 is set, it will send a signal to stage 3 to write back the new value off the destination register and free its reservation. If the execution flag is not set, only the reservation is freed.

3.2.2 Extending the verifier

To add the conditional move to the verifier, I had to specify its format and add helper functions to execute the conditional move on the verifier state. Furthermore I needed a printing function that outputs the instruction in assembly like form for debugging purposes. Finally those functions were added to the main verifying function to be called in the case of a conditional move.

3.3 Adding different branch predictors

The authors of the RISC-V instruction set argued against conditional instructions because they reasoned that any improvements gained by them can be also be achieved with clever branch prediction algorithms.

The initial version of the bluespec processor uses only a small directly mapped BTB of size eight. I will subsequently refer to it as branch predictor with BTB and tablesize 8 (BTB8). A first and intuitive improvement of this branch predictor is to increase the table size, which I will refer to as branch predictor with BTB and tablesize 512 (BTB512). The size is chosen, as according to Perleberg and Smith[**perleberg1993branch**], larger tables do not result in large improvements of the prediction accuracy.

A further simple extension of the branch predictor is using saturating counters instead of a single valid bit. If the counter is above a certain threshold, usually half its capacity, the branch is predicted taken. Whenever the branch was actually taken the counter increases. It decreases if the branch was not taken. Unfortunately the interface of the branch predictor within the FLUTE processor only gives feedback on a mis-predicted branch, which made updating the saturating counter on a successful prediction difficult. Changing the interface and adding a feedback paths in the processor would be possible, but I want to compare the exact same processor with different branch predictors, including the initial one. Therefore I kept the interface and used only the feedback it provided.

I solved this problem by assuming the prediction is correct and updating the counter accordingly. On a mis-predict the counter is incremented/decremented accordingly by twice the usual amount. This is slightly biased because predicted branch instructions that are not committed (for example when an earlier mis-predictions caused a pipeline flush) are assumed to be correctly predicted even if they are not. This happens because there will not be any feedback. It also requires a three bit saturating counter because the two bit counter effectively collapses into a single prediction bit. Consider the following example: the two bit counter is at 0 and you predict the branch non-taken. Because it is already at the lowest value it can not be decreased further. But when it was actually taken it is corrected by two steps. Its current value is 2 and the next branch will be predicted taken. Therefore more than two bits are needed to implement a saturating counter that tolerates more than one mis-prediction before the prediction is changed.

My design, a branch predictor with 3-bit saturating counters (SAT) and BTB, can be seen in figure **??**. It is similar to the initial BTB but the bit indicating whether an entry is valid has been replaced by the saturating counter. The table size is 1024 entries.

Figure 3.3: A branch predictor with branch target buffer and saturating counters

More advanced branch predictors exploit the local correlation of branches. The results of the last branches is used to predict the next branch. One resource saving approach is using a shared history table. Part of the PC is used to index a table of histories of branches. The history is then used as a pointer into a shared table of saturating counters to predict the branch. As this design is used to predict branches with changing branch behaviour I chose it as an alternative to predict *data dependent branches* and to contrast it with conditional moves.

Figure 3.4: Schematic design of the local history predictor with branch targets, both the history pattern table and the shared counter table have 1024 entries

In my design, called local history branch predictor (LHBP), I combined this local history predictor with the BTB to predict both whether the branch is taken and the branch target, see figure **??**.

To access the old counter an additional history bit is stored, see figure **??**

Because the branch is predicted in stage one and mis-prediction feedback is send from stage six, five other instructions are in flight in between. Hence he

Figure 3.5: A single history entry and how it is used to access the saturating counters

correction scheme fails to work correctly if the same branch has to be predicted within five consecutive instructions. I deemed this a rare enough event to neglect this. Two different branches within five consecutive instructions are more common though, which makes the branch predictor interface unsuitable for global history branch prediction.

3.4 Extending LLVM and GNU Assembler

3.4.1 LLVM

The LLVM back-end runs several optimisations on the intermediate code produced by clang before assembly code is generated. In the selection pass *SelectionDAG* nodes are replaced by the operations supported by the target instruction set. It is also able to combine several nodes and replace them with one instruction. This is how I will implement the conditional move. I defined a *SelectionDAG* node that represents the conditional move instructions using LLVMs table description language (see listing 3.2). First I define an abstract conditional move node as the class `InstCMOV`. The arguments of this node are the same as the select node. It has the additional restriction that the second source argument, which is the one picked when the condition is not set, is the same register as the destination. This corresponds to the behaviour of the conditional move: the value of the destination is kept when the condition is false. The type of the condition, source and destination registers are parameterized. This way I can re-use the definition as the condition, source and destination register types can either be of type `GR32` or `GR64`, representing 4byte integers and 8byte longs respectively. These types are internal to the LLVM back-end, and will later produce the same assembly instructions as all RISC-V registers are 64 bit wide. It is important though to generate conditional moves when either the condition variables, the source and destination or both are of type `long`.

The actual conditional move instructions are then defined as instances of this class.

Listing 3.2: Definition of the conditional move SelectionDAG node; extracted from RISCVInstrInfo.td

```
class InstCMOV<string mnemonic, bits<3> funct3, RegisterOperand ConditionRO,
          RegisterOperand DataRO> : InstRISCV<4, (outs DataRO:$dst),
          (ins ConditionRO:$cond, DataRO:$src1, DataRO: $src2),
          mnemonic#”\t$dst,␣$src1,␣$cond”, []>
{
     let Constraints = ”$src2␣=␣$dst”;
     ....
}
def CMOV_EZ_32_32 : InstCMOV<”cmovez”,0b000, GR32, GR32>;
def CMOV_EZ_64_32 : InstCMOV<”cmovez”,0b000, GR64, GR32>;
...

def CMOV_NZ_32_32 : InstCMOV<”cmovnz”,0b001, GR32, GR32>;
def CMOV_NZ_64_32 : InstCMOV<”cmovnz”,0b001, GR64, GR32>;
...
```

I then define patterns to replace certain *SelectionDAG* nodes with conditional move instructions. An example is shown in listing 3.3: the *select* node is replaced by a `cmovnz` node. This is part of a multiclass pattern definition that allows me to parameterize the type of the registers and the instructions. Later the multiclass patter is used to define patterns for different combinations of the types `GR32` and `GR64` as the condition register and source and destination register respectively.

Listing 3.3: example *SelectionDAG* pattern; extracted from RISCVInstrInfo.td

```
  def : Pat<(select (i32 (setlt CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F
                     (MOVNZInst (SLTOp CondRO:$rhs, CondRO:$lhs), RO:$T, RO:$
```

Figure 3.6: An extract of a *SelectionDAG* with a select node before the selection pass

Figure 3.7: The same extract of a *SelectionDAG* as in figure **??**, but after the selection pass

The replacement of a select node with a conditional move can be seen on the extracts from the *SelectionDAG* before and after the selection pass. Figure **??** and **??** show an applied example of the pattern from listing 3.3.

The same patterns have been defined for select nodes with set-less-than, set-less-equal, set-greater-than, set-equal and set-not-equal.

It has been tested with C-code such as `(a >= 0) ? 2 : 4` for all conditions, See appendix B.1 for the test results. This shows that ternary if statements will result in select nodes in the *SelectionDAG* and that any combination of select node with equals, non-equals, less-than, less-equals, greater-than or greater equals comparison will result in a conditional move instruction in the assembly code.

There is one special case which initially caused the compiler to crash. The C code example

```
(a < 0) ? 2 : 0
```

produced a conditional move node in the *SelectionDAG* where the second argument was the constant zero and therefore a physical register. This caused an exception at a later pass[2] of the LLVM which expected the argument to be a virtual register[3]. The problem is that the second source operand of the conditional move node is also the destination of the result. You cannot use the zero register in that case because it is not updatable. The solution was to add further special patterns that will match if the second operand is zero. They will swap the operands and use the complementary conditional move instruction compared to the normal case. The listing 3.4 shows an example, which is the special case of the listing 3.3 shown earlier. Those patterns had to be defined for all comparisons because some comparisons required the `cmovnz` instruction and some `cmovez`. See appendix A for a full list of all pattern definitions.

Listing 3.4: *SelectionDAG* pattern consisting of a select where the second source operand is zero and a set-less-than node; it replacement is a `cmovez` node and a set-less-than node with swapped operands; extracted from RISCVInstrInfo.td

```
def : Pat<(select (i32 (setlt GR32:$lhs, GR32:$rhs)), GR32:$T, 0),
                  (CMOV_EZ (SLT GR32:$rhs, GR32:$lhs), zero, GR32:$T)>;
```

The next question is whether simple if statements will also generate conditional moves. A test with the example code

```
int b = 4;
if (a < 2) b = 2 ;
```

[2] The pass in question is called twoAddressInstructionPass. It converts three address code into two address code and applies constraints to the used registers as part of the register allocation algorithm.

[3] Physical registers are registers supported by the machine, virtual registers exist only at compile time and are mapped to physical register in the register allocation phase.

reveals that, when LLVM optimisations are enabled using the `-O2` compiler option, the branch is optimized to a select statement which is replaced by a conditional move. This code snippet produces the *SelectionDAG*s in figure **??** and figure **??**, which have been used as an example for a conditional move earlier on. A single test is sufficient as the condition is not affected by this optimisation.

The LLVM contains an optimisation pass called *SimplifyCFG*,which reduces branches to select nodes in certain cases. Other tested cases are:

```
if (a < 2) a = b + 2;

if(a < 2) a = b + 2;
else a = b - 2;

if (a < 2) {
    a = 2;
    b = 2;
}
```

For all of the above examples, the branches are simplified using select nodes, which in the end produce conditional move instructions. The assembly code can be found in appendix B.2.

3.4.2 Contributing to the RISC-V LLVM

Figure 3.8: Simplified view of the instruction lowering process for a select instructions, vr1 - vr4 are arbitrary identifiers for virtual registers

While I was testing the correctness of my compiler changes I noticed that the following code example picked the incorrect value at execution:

```
int result = (a >= 0) ? 0 : 5;
```

Most interestingly this happened not only with the modified LLVM but with the initial version provided by LLVM. It turned out to be a bug in the code that lowers a select instruction to a branch. Figure **??** shows how it works in general: The jump to the empty basic block will be executed iff the condition is not met. The Phi instruction is later replaced with code that moves the value of `vr3` into `vr4` if the branch was taken. If it was not taken `vr2` is copied into it. Unfortunately the Phi instruction only works with virtual registers, and if one of the operands of the select instruction is zero, the physical register `r0` is used. To solve this, the zero value needs to be copied into a physical register which is then used as the argument of the Phi node. The bug was that this copy instruction was inserted in the wrong basic block if the second select operand was zero. To fix the problem the copy operation had to be inserted in its correct place, which is in the first basic block right before the branch.

I discovered another bug in the LLVM were certain programs would not compile when optimisations were enabled. The cause was a function to insert branch statements. It was missing code for certain conditional jumps and causing the compiler to crash if such a conditional jump was supposed to be inserted. I added the code for the missing cases, thus fixing the problem.

3.4.3 The GNU Assembler

The GNU assembler scans the program line by line and matches the commands with a list of riscv *opcode* `structs`. To make the assembler understand the conditional move instructions list of known instructions has to be extended. It can be found in the source file `riscv-opc.c`.

In this file I had to define the conditional move instructions as instances of `struct` called `riscv_opcode`. This `struct` is used to define all assembly instructions, and can be found in the file `riscv.h`. The constants and bit-masks used for the definition of my added instructions can be found in the header file `riscv-opc.h`. This is all the GAS needs to know to translate and disassemble the additional instructions.

The functionality was tested by writing an assembly program that performs conditional moves. It was assembled and disassembled and then run on the the bluespec processor. The final register state of the processor was inspected.

3.5 Adding debug instructions

To better debug programs running on the processor I added the following debug instructions:

```
printall
printone     sr
```

The first instruction will print the entire processor state to standard output, the second one only the value stored the register specified by `st`. I declared and implemented the instruction in a similar way as the conditional move. I used the other custom *opcode* (see figure **??**) and the same instruction format as the conditional move (see figure **??**). The function code *func7* is zero for both instructions, and *func3* is defined to `000` and 001 for the two instructions.

I chose to not actually implement the output functionality in the processor but only in the verifier as it already contained methods to easily access the content of a single register or to print the entire processor state. Therefore the only change need in the processor was to include the instructions in the definitions of the ISA and a rule in the execution stage, stage 4. The only purpose of that rule is to prevent an illegal instruction exception for the *opcode* and check for illegal values of *func3* and func7.

The verifier is extended in same way as it was for the conditional move, with helper functions to print the assembly code of the instructions and to simulated its execution on the processor state of the verifier. This execute method will actually print the processor state or the register value.

The gnu assembler was also extended accordingly to translate the instructions into machine code. I did not modify the LLVM or Clang compiler as the instructions can be used as in-line assembly code.

3.6 Testing the processor

3.6.1 Execution environment

I kept two versions of the RISC-V LLVM: the *initial version*, which only contains the changes I made to fix bugs, and the *cmov version* which is capable of generating conditional move instructions. They are used to create versions of a test with and without conditional moves.

The FLUTE processor provided a small library collection that offers implementations of basic system calls, including `printf` and `malloc`. It also contained a linking script, that would link the object files of the program with the libraries. A C program compiled and linked with these libraries will run on the synthesized processor by loading the code into memory and starting itself.

3.6.2 Compiling and running tests

My initial aim was to test the processor using the SPECInt benchmarks. Unfortunately they rely on libraries and function calls that the FLUTE processor does not provide, for example file I/O or sockets. This means I can compile them into assembly code, but I am missing libraries to link them with.

My first idea to circumvent this was to use the source code of SPEC 2000 benchmark tests modified by the PERSim project[**persim**]. They were run on an OpenRISC machine without operating system. I started with the `bzip2` code. After avoiding the unsupported system calls that were still in the code (mainly `fprintf`) I was able to compile the code. When executed though, the program did not terminate, neither on the FLUTE processor nor on my x86-64.

The second idea was to modify the code of the bzip2 SPECint2006 benchmark myself. It contains functions to compress and decompress a string in memory. Hence generating a string, encrypt it, decrypt it and check the result can be done without any unsupported system calls such as file access. It executed successfully on my x86-64 machine and on the FLUTE processor when compiled with the RISC-V gcc. However when compiled with the *initial* RISC-V LLVM or the *cmov* version and optimisations enabled, the program did not terminate. The compiler apparently changed the semantics of a program in some cases. One example is given in listing 3.5.

Listing 3.5: Example C code that will compile incorrectly if compiled with the RISC-V LLVM and optimisations enabled

```
while (True) {
        ...
        if (zPend < 2) break;
        zPend = (zPend - 2) / 2;
}
```

When the bzip2 code was run, this code looped indefinitely although the initial value of `zPend` was -1. When the same control flow was mocked up in a test, the loop was executed twice when `zPend` was -1 initially. It only happened when compiled with optimisations (for both level `-O1` and `-O2`). At first I tried to rewrite the code such that the malfunctioning optimisation would not apply, but as infinite loops had been introduced at different places with different control flow structure, this approach was not effective. As the optimisation passes in LLVM are usually shared among different versions and extensively tested, a bug in the code generation is likely.

Unfortunately I did not have the time to find the source of this bug. I only had the incorrectly generated output, which did not point to any particular part in the RISC-V LLVM backend, which comprises almost 10,000 lines of code. Therefore I accepted that more complex benchmark tests might not compile correctly. Instead I focused on the tests that executed successfully.

Some simple test programs came with the implementation, such as a “hello world”[4] program or a so called filter program, that applies a 3x3 filter to a two-

[4] This was useful to test the general functionality of the compiler and processor.

dimensional array that represents a random picture. All could be compiled and run successfully Compiling and executing Dhrystone was also no problem.

The WCET benchmarks suite turned out to be no problem either, although one out of the 35 test could not link. As it did not generate conditional moves, I ignored this case. The BEEBS benchmark suite proved to be a bit more challenging. Out of 47 tests, 8 did not compile at all. Three tests needed functions from the math library I could not provide and the five other cases caused the compiler to crash. Some benchmarks were slightly modified to compile and execute successfully, for example by adding a square-root or to-lower-case function. Even then, 7 tests that compiled could not be executed for various reasons. Some seemed to not terminate. Others were missing libraries at the linking stage and for one test execution stopped because the processor and the verifier disagreed on the result of an instruction, even on the unmodified processor. Fortunately I still had enough valid test results to be able to disregard the ones not working.

All tests that executed successfully produced the same output for both the initial and my extended processor. This indicates that my changes preserve the semantics.

3.6.3 Collecting data

To collect the data needed I used the CISSR verifier that runs parallel to the processor. It already counts the number of instructions executed in a run and I extended it to use count conditional move instructions, branches and jumps separately. The number of branches and jumps are needed to calculate the prediction accuracy of the branch predictor. I used only to count the number of misses using debug outputs. Each update of the branch predictor is counted as a miss, so that predicted branch instructions that are not committed will not be counted.

The simulation test bench of the processor also provides the number of cycles the program needed to execute. It already excludes the cycles needed to load the program into memory. These counts are used to evaluate the prediction accuracy and the overall performance in IPC as follows:

$$\text{prediction accuracy} = 1 - \frac{\text{No. of misspredictions}}{\text{No. of branches and jumps}} \tag{3.1}$$

$$\text{IPC} = \frac{\text{No. of instructions}}{\text{No. of cycles}} \tag{3.2}$$

The general speedup of a program will be measured as the following relation between the clock cycles the new and old version need to execute. If the new version only needs half the clock cycles to complete is 100% faster.

$$\text{Speedup} = \frac{\text{No. of cycles}_{old}}{\text{No. of cycles}_{new}} - 1 \tag{3.3}$$

Chapter 4

Evaluation

In this section, I evaluated the performance of my compiler and processor. To begin with, I compared the number of conditional moves my LLVM back-end produces with the MIPS back-end to find that mine performs almost as well. From 72 benchmarks, 31 generated conditional moves. Seven of them failed to terminate and 15 actually executed conditional moves at run-time. This results in an average dynamic frequency of 0.84% over all executable benchmarks. Out of those 15, seven exhibited significantly reduced numbers of mis-predictions and 9 experienced speedups of up to 19.7%. Those speedups weakly correlate with the number of times a conditional move was executed dynamically. In some cases a better branch predictors provided a larger speedup, but for four benchmarks the conditional move outdid even the local history branch predictor and provided at least twice the speedup. Synthesising my design indicated that adding a conditional move has no negative impact on its clock frequency, nor does it come with a noticeable extra cost of resources.

All measurements were, if not stated otherwise, completely deterministic so no standard deviation or error bars are given.

4.1 Comparison to MIPS

To examine the quality of my compiler changes and to see how many conditional move it can produce I compared it with the MIPS LLVM back-end.

Table **??** contrasts number of conditional moves my RISC-V clang compiler generates with the number of conditional moves the MIPS version of clang uses. For the 43 unlisted benchmarks neither of the two compilers generates any conditional moves. My compiler produces fewer conditional moves in six cases, but more conditional moves in one case.

benchmark	RISC-V cmovs	MIPS cmovs
dhrystone	1	1
filter	3	3
wcet: adpcm	54	45
wcet: compress	8	8
wcet: crc	16	16
wcet: expint*	3	3
wcet: fibcall*	1	1
wcet: janne_complex*	1	1
wcet: ndes	1	1
wcet: select*	2	2
beebs: aha-compress	4	4
beebs: aha-mont64	0	15
beebs: blowfish*	7	7
beebs: ctl-stack	10	10
beebs: dtoa**	46	51
beebs: huffbench**	7	7
beebs: levenshtein	2	2
beebs: mergesort	8	8
beebs: miniz**	41	41
beebs: nettle-cast128*	1	1
beebs: newlib-exp*	0	1
beebs: newlib-mod*	1	1
beebs: picojpeg**	24	24
beebs: qrduino	83	84
beebs: rijndael*	1	1
beebs: sglib-arraybinsearch	2	3
beebs: sglib-arrayheapsort	4	4
beebs: sglib-arrayquicksort	1	1
beebs: sglib-queue	2	2
beebs: slre**	17	18
beebs: stringsearch1**	1	1
beebs: wikisort	20	20
bzip2**	92	93
total	464	480

Table 4.1: number of occurrences of conditional move instructions in the assembly code when compiled with my modified RISC-V and the MIPS clang compiler; *: no conditional move at run-time; **: could not link or run successfully

Only looking at the accumulated number of conditional moves, my compiler generates conditional moves in 96.7% of the cases when the MIPS compiler generates one[1]. Hence my RISC-V clang compiler is almost as good as the MIPS compiler.

The difference may partly be explained with floating point conditional moves which MIPS supports but my implementation lacks.

4.2 Dynamic frequency of conditional moves

When you look at the dynamic frequencies in table **??** you notice that not all benchmarks that use conditional moves actually execute them at runtime. One explanation for this is that all these benchmarks are single path programs[2]. Only 15 out of the 29 benchmarks I could successfully compile and execute use the conditional move at runtime.

Summarizing over all executable benchmarks, the combined dynamic frequency[3] of conditional move instructions was 0.8%. This is slightly less than the book value of 1% from Hen&Pat[**HenPat2012**]. This is a very crude measurement however, as the values were quite spread out, with a maximum of 9.4%, and the executed instructions differed by up to three orders of magnitude between the benchmarks.

4.3 Performance changes

4.3.1 Changes is branch prediction accuracy

The concept behind a conditional move is to replace branches and avoid the cycle penalty of branch prediction misses. First I look at the fraction of mis-predictions with conditional move over mis-predictions without a conditional move, as depicted in figure **??**.

Figure 4.1: The percentage of branch prediction misses with cmov over branch prediction misses without cmov

[1] This ignores that at least for `wcet: adpcm` my compiler generates conditional move at places where the MIPS compiler does not.

[2] If the program requires input, only one example data set is provided in the program, causing it to go along one execution path.

[3] I took the sum of all executed conditional moves over sum of all instructions.

We see that the conditional move has significant impact for several benchmarks: For seven of them the number of mis-predictions decreases by $\sim 20\%$ or more. Seven more cases only show a minor improvement. Interestingly, the number of misses increases for `sglib-arraybinsearch`. A possible explanation is that two branch instructions that used to have different indices into the BTB now have the same. They clash and cause mis-predictions.

4.3.2 Changes in execution speed

Figure 4.2: relative change of instructions per cycle from non-cmov program to cmov program

Figure 4.3: Speedup $= \frac{\text{No. of cycles}_{old}}{\text{No. of cycles}_{new}} - 1$ per benchmark

Figures **??** and **??** show how the increased prediction accuracy translates into an overall speedup of the program, measured in IPC and speedup in terms of clock cycles, respectively.

Four of the benchmarks show significant improvements in both measurements. This is unsurprising as those tests show the highest reduction in mis-predictions. `sglib-arraybinsearch` is an interesting case because it has a high dynamic frequency of conditional moves, which translates into a comparatively high IPC improvement. Yet its speedup is only moderate. This happens because the number of instructions actually increases (see table **??**, gray highlight) with a conditional move. Figure **??** reveals the context in which the conditional moves are used. If the condition is true in most cases more instructions are executed. The combination of the increased number of instructions and only slightly fewer executed cycles result in a higher IPC.

Figure 4.4: Code example from `sglib-arraybinsearch`(variable names changed) and how it is compiled with and without a conditional move

The `ctl-stack benchmark` is a good example for another peculiarity. Its IPC decreases but you still see a speedup as fewer cycles are executed. Figure **??** shows typical occurences of conditional move instructions from it. In the first example fewer instructions will be executed if the condition is satisfied and the branch taken. In the second case fewer instructions are executed even if the

benchmark	Dynamic	Instruction count		IPC	Speedup
	Frequency	with	without	change	in %
		cmov	cmov	in %	
wcet: crc	9.4	21780	21012	12.38	8.4
beebs: levenshtein	6.77	30417	31969	1.66	6.8
beebs: sglib-heapsort	6.53	16146	17251	12.02	19.7
beebs: sglib-arraybinsearch	6.13	8903	8584	5.26	1.5
beebs: ctl-stack	5.67	7090	8090	-2.53	11.2
beebs: sglib-queue	4.23	20215	21258	6.09	11.6
beebs: qrduino	3.02	1990334	2013766	-0.06	1.1
filter	1.99	116280	126215	6.19	15.3
beebs: aha-compress	0.52	13898	13848	-0.18	-0.5
wcet: compress	0.29	17729	19103	-5.51	1.8
dhrystone	0.28	3564747	3554747	0.83	0.5
beebs: sglib-quicksort	0.16	8723	8714	0.22	0.1
wcet: adpcm	0.11	82077	82116	0.03	0.1
beebs: mergesort	0.07	515669	515916	-0.06	0.0
wcet: ndes	0.04	37350	37250	0.19	0.1

Table 4.2: Dynamic frequency of conditional moves, dynamic instruction count, IPC change and speedup per benchmark, sorted by dynamic frequency; green: significant speedups($> 5\%$) , yellow: minor speedups(between 1% and 2%)

condition is false. This reduces the number of executed instructions by 12% (see table **??**, gray highlight) for ctl-stack. The number of cycles only decreases by 10%, which results in a lower IPC. This explanation is transferable to other cases where the decrease of cycles outperforms the change in IPC.

Figure 4.5: example of semantically equal assembly code with and without conditional moves, from the `ctl-stack` benchmark

Table **??** indicates that the speedup correlates with the dynamic frequency of conditional moves. The correlation coefficient of the dynamic frequency and the speedup is $r = 0.6$, indicating a week linear correlation between the two, see figure **??**.

In summary, six out of 75 benchmarks[4] showed a major speedup and three

[4] Excluding those that produced conditional moves but could not be executed.

Figure 4.6: Speedup mapped onto dynamic frequency for each benchmark; correlation coefficient $r = 0.6$

more presented a minor improvement. The overall speedup of the program is a more reliable measurement of the performance than the IPC. The examples demonstrated how changes in the number of executed instructions may skew the IPC, making it less suitable. A weak positive correlation could be found between the dynamic frequency of conditional moves and the resulting speedup.

4.4 The effect of different branch predictors

A conditional move was not included in the standard RISC-V instruction set as the authors argued that advanced branch prediction will predict most branches, thereby making condition moves obsolete. To counter that argument I investigate how performance changes with different branch predictors.

The branchpredictors I used were BTB8: abranch predictor with BTB and tablesize 8, BTB512: a branch predictor with BTB and tablesize 512, SAT: a branch predictor with 3-bit saturating counters and LHBP: local history branch predictor. Again I will start by looking at the changes in branch prediction misses compared to the base case without a conditional move, see figure ??.

(a) Prediction misses

Figure 4.7: Branch prediction misses and speedup for BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

In the majority of cases (`dhrystone`, `ndes`, `compress`, `aha-compress`, `ctl-stack`, `mergesort`, `qrduino`, `sglib- arraybinsearch` and `sglib- quicksort`), the branch predictor reduces the number of misses more than the conditional move could. Notice that in some cases (`aha-compress`, `ctl-stack`, `qrdruino`, `sglib-arraybinsearch` and `sglib-quicksort`) the combination of the BTB512 and the conditional move can further reduces the number of misses. This indicates that the conditional move is still capable of avoiding branches the predictor fails to predict correctly. Only for the remaining four benchmarks (dhrystone, ndes, compress, mergesort) the conditional move offers no advantages over the branch predictor.

In six cases however the conditional move has a larger impact on the prediction misses. Again you can distinguish the tests in which only the conditional moves provide an advantage (`crc`, `sglib-heapsort` and `sglib-queue`), and the benchmarks where both offer advantages (`filter`, `adpcm`, `levenshtein`).

In general a large decrease in prediction misses produces a significant speedup. The only exception is `adpcm`. For this benchmark the processor only executes 0.075 instructions per cycle, indicating that some other instructions, for example loads causing a cache miss, limits the performance. The frequency of branch instructions is only 7.9% without conditional moves, so improving the speed of them does not have a large impact. For `sglib-arraybinsearch` the combination of conditional move and BTB512 performs slightly slower than the BTB512 alone. This can also be explained by the increase in executed instructions with a conditional move.

For the more advanced branch predictors I omitted the results for `compress`, `ctl-stack`, `mergesort`, `qrduino`, `sglib-arraybinsort` and `quicksort`. They behave similar to dhrystone and ndes and add no value to the discussion. The full results can be found in appendix D.

(a) Prediction misses

(b) Speedup

Figure 4.8: Branch prediction misses and speedup for SAT and BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

Figure **??** looks at the prediction misses and speedups when the branch predictor with saturating counter is used. The values with the BTB512 are kept for comparison. This shows that SAT performs worse in some cases. The most likely explanation is that the predictor will cause more misses initially before it is properly trained. Because the benchmarks are so small, some may not leave this training period and perform worse.

The benchmarks where the conditional move causes a larger speedup remain unchained in that respect. For `levenshtein` the number of misses with a conditional move and with SAT respectively are of the same level. Yet the conditional moves produce a larger speedup because fewer instructions are executed[5].

Also noticeable are the benchmarks `levenshtein`, `sglib-heapsort` and `sglib-queue`. Here the combination of conditional move and SAT reduce the misses even further, resulting in a higher speedup.

[5] Again the conditional move might result in fewer executed instructions, see figure **??**.

(a) Prediction misses

(b) Speedup

Figure 4.9: Branch prediction misses and speedup for LHBP and BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

CPU version	CPU Clock frequency
initial	117 ± 5 MHz
with cmov	121 ± 2 MHz

Table 4.3: Mean clock frequency of the CPU for the initial and the cmov design

The local history branch predictor produces similar results, see figure **??**. It excels for `dhrystone`, which runs long enough to train the saturating counter for all history patterns. In the majority of cases it performs worse than BTB512 due to insufficient training. One striking case is the `crc` benchmark. LHBP produces fewer misses than BTB512 or SAT but still more than any version with a conditional move. Still the speedup is slightly larger because the conditional move version has more instructions to execute.

In conclusion conditional moves provide a larger speedup for four of the benchmarks, even with advanced branch predictors. However, this is only of limited validity as three BEEBS programs are quite small and may fail to train the branch predictor properly.

4.5 Changes in clock frequency

To check whether my design would have negative effects on the clock frequency of the processor I synthesised it for the terasic DE4 Field Programmable Logic Array (FPGA) board using Quartus. Then I compared the results of the timing analysis with the results for the initial version. As the Quartus compilation is non-deterministic I collected results from 12 different compilation runs for each version. The Shapiro-Wilk test [**shapiro1965analysis**] showed that the results are distributed normally. Welch's t-test[6] indicates that the distributions of both samples and therefore their means are different.

[6] This test determines whether two samples originate from the same normal distribution. It differs from Student's t test because it does not assume that the variances of the two samples are equal[**welch1947generalization**].

CPU version	Combinational ALUT	Registers	ALM
initial	35134 ± 5	19354 ± 2	29500 ± 170
with cmov	35144 ± 8	19353 ± 1	29120 ± 160
Mann-Whitney test	different	same	different

Table 4.4: Mean resources needed by the fitting algorithm for the initial and the conditional move version of the processor over 12 runs each; the Mann-Whitney test indicates whether the samples were generated by the same or different distributions

The results indicate that the cmov version of the processor is slightly faster than the initial version, see table **??**. This is surprising because I only added functionality to the processor. A possible explanation is that my design happens to be easier for the fitter to place on the FPGA board. The analysis also showed that the critical path limiting the clock frequency is accessing the data-cache, which is not directly effected by the conditional move. If the fitting was to be hand optimized I would expect the same clock frequencies for both version.

In summary you can still conclude that adding a conditional move to the processor does not harm the clock frequency.

4.6 Resource usage

Synthesising the design also helped me to estimate its complexity. I specifically looked at the resources the fitting algorithm needed to place the design on the chip. The FPGA chip consists of arrays of Adaptive Logic Module (ALM). Each module contains two Adaptive Look-Up Table (ALUT) and two registers. The fitter within the Quartus compiler will program and link those modules to synthesize the described hardware.

Table **??** shows the resources needed for the CPU module only. I used the Mann-Whitney[**mann1947test**] test to determine whether the samples were generated by different probability distributions[7]. According to this test the number of combinational ALUTs are generated by different distributions, allowing us to compare the means. The cmov version shows an increase of less than 0.1% for ALUTs. The same applies for the ALMs, only that the number actually decreases by about 1%. This only indicates that the cmov design can use both the ALUT

[7] The Mann-Whitney test does not assume that the distributions are normal, which was not the case for the resources needed.

and the register within one ALM more often. The number of registers can be assumed to be the same, as they are generated by the same distribution.

From the resources allocated by the fitter you can conclude that an actual hardware implementation of the conditional move would not increase the size and material cost of a chip significantly.

4.7 Complexity of the compiler

The authors of the RISC-V instruction set also claimed that it would be difficult to handle exceptional behaviour in combination with conditional move optimisations. Remembering the example

```
if(c =! 0) {
  a = b / c;
}
```

I compiled this to find that LLVM detects this special case and does not use a conditional move[8]. Therefore incorrect behaviour is avoided without additional effort or complexity because LLVM is already aware of this. Alternatively a non-trapping divide operation would also fix the problem.

[8] If, on the other hand, addition is used instead of the division operator, a conditional move instruction is generated because no exception can be raised.

Chapter 5

Conclusion

This chapter I comment on the complex interplay of the different components responsible for the performance of the processor. I conclude that a conditional move is worth including in the RISC-V instruction set. It is especially beneficial for simple microprocessors whose branch prediction is limited. Furthermore I reflect on my implementation of this project. I finish with further extensions and questions that emerged in the course of this project but which I was unable to follow in the time given.

5.1 Summary of results

It is advantageous to include a conditional move instruction in the RISC-V instruction set. Although it only provides a speedup in some application specific cases, those benchmarks run up to 20% faster. I found a weak correlation between the dynamic frequency of conditional moves and the resulting speedup, thus validating my initial hypothesis. The speedup was also influenced by the number total number of executed instructions. This might decrease or increase with conditional moves depending on the application, making potential improvements difficult to predict.

It is also important that, for none of the tests, the conditional move instructions had a negative effect on the number of clock cycles needed to execute. Furthermore it did not have any negative impact on the clock frequency of my processor, nor does it add significantly to the size of my design. This observation is limited to simple pipelines with in-order execution and one instruction fetch per cycle. The result need not hold for complex super-scalar processors.

This means the conditional move instructions and its speedups come at no significant cost for simple microprocessors. Within the limits of the small benchmarks I had available, I could show that these improvements could not be matched with branch prediction schemes. The added complexity for the compiler turned out to be no problem as well. Therefore I could demonstrate that the reservations against a conditional move, mentioned in the RISC-V manual were unfounded.

You have to keep in mind though that the results were obtained using mainly small benchmarks. Furthermore my extension of the FLUTE processor does not allow for register renaming, so the result are only valid for simple, non-superscalar designs. But especially for simple microprocessors, whose size and energy constrains do not allow complicated branch prediction, a conditional move is useful.

5.2 Reflection on my approach

I chose to extend the RISC-V LLVM compiler instead of the RISC-V gcc because of its cleaner and modular design, along with the support the LLVM intermediate code representation offered for conditional moves. In hindsight this limited me in the number and type of benchmarks I could run. Hence it might have been better to extend the RISC-V gcc compiler instead.

5.3 Potential further steps

Unfortunately my project suffered from bugs in the RISC-V LLVM back-end which I did not expect. Hence an obvious continuation of my project is to fix the compiler back-end and test the conditional moves with SPEC integer benchmark tests.

Another further extension on the compiler side would be to investigate the differences between the MIPS and the RISC-V back-ends. Although my compiler performed almost as well as the MIPS version, the latter still produced more conditional moves in some cases. I could also look into the compiler optimisation passes responsible for inserting conditional moves (or rather the `select` intermediate instruction that later on generates a conditional move) to see whether they could be more aggressively generating them. It would be interesting to see at what stage it becomes disadvantageous to replace a branch with conditional moves. Alternatively you could try to find a more conservative behaviour that only generates conditional moves when they will not increase the total number of executed instructions.

On the hardware side of my project it would be interesting to compare different designs of conditional moves, as for example the MIPS64 `select` instructions.

Appendix A

LLVM extension

The following listing shows the definition of the conditional move instructions within the RISC-V LLVM back-end, and the definition of the instruction patterns that shall be replaced with conditional moves.

```
//Conditional moves
class InstCMOV<string mnemonic, bits<3> funct3, RegisterOperand Condition
  : InstRISCV<4, (outs DataRO:$dst), (ins ConditionRO:$cond, DataRO:$src1
                 mnemonic#"\t$dst,␣$src1,␣$cond",
                                 []> {

  let Constraints = "$src2␣=␣$dst";

  field bits<32> Inst;
  bits<5> RD;
  bits<5> RS1;
  bits<5> RS2;
  let Inst{31-25} = 0b0000000;
  let Inst{24-20} = RS2;
  let Inst{19-15} = RS1;
  let Inst{14-12} = funct3;
  let Inst{11- 7} = RD;
  let Inst{6 - 0} = 0b0001011;
}

def CMOV_EZ_32_32 : InstCMOV<"cmovez",0b000, GR32, GR32>; // Condition:
def CMOV_EZ_64_32 : InstCMOV<"cmovez",0b000, GR64, GR32>; // Condition:
def CMOV_EZ_32_64 : InstCMOV<"cmovez",0b000, GR32, GR64>; // Condition:
```

```
def CMOV_EZ_64_64 : InstCMOV<”cmovez”,0b000, GR64, GR64>; // Condition: 64 b

def CMOV_NZ_32_32 : InstCMOV<”cmovnz”,0b001, GR32, GR32>; // Condition: 32 b
def CMOV_NZ_64_32 : InstCMOV<”cmovnz”,0b001, GR64, GR32>; // Condition: 64 b
def CMOV_NZ_32_64 : InstCMOV<”cmovnz”,0b001, GR32, GR64>; // Condition: 32 b
def CMOV_NZ_64_64 : InstCMOV<”cmovnz”,0b001, GR64, GR64>; // Condition: 64 b

// select patterns
multiclass MovzPats0<RegisterOperand CondRO, RegisterOperand RO,
                     Instruction MOVZInst, Instruction SLTOp,
                     Instruction SLTuOp, Register ZEROReg> {
  //setge
  // Fun fact: the lhs and rhs of all these conditions seem the wrong way ro
  def : Pat<(select (i32 (setge CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F),
            (MOVZInst (SLTOp CondRO:$rhs, CondRO:$lhs), RO:$T, RO:$F)>;
  def : Pat<(select (i32 (setuge CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F),
            (MOVZInst (SLTuOp CondRO:$rhs, CondRO:$lhs), RO:$T, RO:$F)>;

  //setle
  def : Pat<(select (i32 (setle CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F),
            (MOVZInst (SLTOp CondRO:$lhs, CondRO:$rhs), RO:$T, RO:$F)>;
  def : Pat<(select (i32 (setule CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F),
            (MOVZInst (SLTuOp CondRO:$lhs, CondRO:$rhs), RO:$T, RO:$F)>;

  // Special cases if the second source operand (the 'False' option) is zero
  //setgt
  def : Pat<(select (i32 (setgt CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
            (MOVZInst (SLTOp CondRO:$lhs, CondRO:$rhs), ZEROReg, RO:$T)>
  def : Pat<(select (i32 (setugt CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
            (MOVZInst (SLTuOp CondRO:$lhs, CondRO:$rhs), ZEROReg, RO:$T)
  //setlt
  def : Pat<(select (i32 (setlt CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
            (MOVZInst (SLTOp CondRO:$rhs, CondRO:$lhs), ZEROReg, RO:$T)>
  def : Pat<(select (i32 (setult CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
            (MOVZInst (SLTuOp CondRO:$rhs, CondRO:$lhs), ZEROReg, RO:$T)
}
multiclass MovzPats1<RegisterOperand CondRO, RegisterOperand RO,
                     Instruction MOVZInst, Instruction SLTOp,
                     Instruction SLTuOp, Instruction XOROp, Register ZEROReg
  //seteq
  def : Pat<(select (i32 (seteq CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F)
                                (MOVZInst (XOROp CondRO:$lhs, CondRO:$rhs
  def : Pat<(select (i32 (seteq CondRO:$lhs, 0)), RO:$T, RO:$F),
                                (MOVZInst CondRO:$lhs, RO:$T, RO:$F)>;
  //setne
  def : Pat<(select (i32 (setne CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                                (MOVZInst (XOROp CondRO:$lhs, CondRO:$rhs
  def : Pat<(select (i32 (setne CondRO:$lhs, 0)), RO:$T, 0),
                                (MOVZInst CondRO:$lhs, ZEROReg, RO:$T)>;
}

multiclass MovnzPats0<RegisterOperand CondRO, RegisterOperand RO,
                      Instruction MOVNZInst, Instruction SLTOp,
                      Instruction SLTuOp, Register ZEROReg> {
  //setlt
  def : Pat<(select (i32 (setlt CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F)
                (MOVNZInst (SLTOp CondRO:$rhs, CondRO:$lhs), RO:$T, RO:$F
  def : Pat<(select (i32 (setult CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F
                (MOVNZInst (SLTuOp CondRO:$rhs, CondRO:$lhs), RO:$T, RO:$
  //setgt
  def : Pat<(select (i32 (setgt CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F)
                (MOVNZInst (SLTOp CondRO:$lhs, CondRO:$rhs), RO:$T, RO:$F
  def : Pat<(select (i32 (setugt CondRO:$lhs, CondRO:$rhs)), RO:$T, RO:$F
                (MOVNZInst (SLTuOp CondRO:$lhs, CondRO:$rhs), RO:$T, RO:$

  // Special cases if the second source operand (the 'False' option) is
  //setge
  def : Pat<(select (i32 (setge CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                (MOVNZInst (SLTOp CondRO:$rhs, CondRO:$lhs), ZEROReg, RO
  def : Pat<(select (i32 (setuge CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                (MOVNZInst (SLTuOp CondRO:$rhs, CondRO:$lhs), ZEROReg, RO
  //setle
  def : Pat<(select (i32 (setle CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                (MOVNZInst (SLTOp CondRO:$lhs, CondRO:$rhs), ZEROReg, RO
  def : Pat<(select (i32 (setule CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                (MOVNZInst (SLTuOp CondRO:$lhs, CondRO:$rhs), ZEROReg, RO
}
```

```
multiclass MovnzPats1<RegisterOperand CondRO, RegisterOperand RO,
                      Instruction MOVNZInst, Instruction SLTOp,
                      Instruction SLTuOp, Instruction XOROp, Register ZEROReg
  //seteq
  def : Pat<(select (i32 (seteq CondRO:$lhs, CondRO:$rhs)), RO:$T, 0),
                                (MOVNZInst (XOROp CondRO:$lhs, CondRO:$rhs),
  def : Pat<(select (i32 (seteq CondRO:$cond, 0)), RO:$T, 0),
                                (MOVNZInst CondRO:$cond, ZEROReg, RO:$T)>;
  //setne
  def : Pat<(select (i32 (setne CondRO:$lhs, CondRO:$rhs)), RO:$src, RO:$dst
                                (MOVNZInst (XOROp CondRO:$lhs, CondRO:$rhs),
  def : Pat<(select (i32 (setne CondRO:$cond, 0)), RO:$src, RO:$dst),
                                (MOVNZInst CondRO:$cond, RO:$src, RO:$dst)>;
}

// Instantiation of conditional move patterns.

// condition 32 bit value, source/destination 32 bit value:
defm : MovzPats0<GR32, GR32, CMOV_EZ_32_32, SLT, SLTU, zero>;
defm : MovzPats1<GR32, GR32, CMOV_EZ_32_32, SLT, SLTU, XOR, zero>;
defm : MovnzPats0<GR32, GR32, CMOV_NZ_32_32, SLT, SLTU, zero>;
defm : MovnzPats1<GR32, GR32, CMOV_NZ_32_32, SLT, SLTU, XOR, zero>;
// condition 64 bit values, source/destination 32 bit value:
defm : MovzPats0<GR64, GR32, CMOV_EZ_32_32, SLT64, SLTU64, zero>,
                 Requires<[IsRV64]>;
defm : MovzPats1<GR64, GR32, CMOV_EZ_64_32, SLT64, SLTU64, XOR64, zero>,
                 Requires<[IsRV64]>;
defm : MovnzPats0<GR64, GR32, CMOV_NZ_32_32, SLT64, SLTU64, zero>,
                 Requires<[IsRV64]>;
defm : MovnzPats1<GR64, GR32, CMOV_NZ_64_32, SLT64, SLTU64, XOR64, zero>,
                 Requires<[IsRV64]>;
// condition 32 bit values, source/destination 64 bit value:
defm : MovzPats0<GR32, GR64, CMOV_EZ_32_64, SLT, SLTU, zero_64>,
                 Requires<[IsRV64]>;
defm : MovzPats1<GR32, GR64, CMOV_EZ_32_64, SLT, SLTU, XOR, zero_64>,
                 Requires<[IsRV64]>;
defm : MovnzPats0<GR32, GR64, CMOV_NZ_32_64, SLT, SLTU, zero_64>,
                 Requires<[IsRV64]>;
defm : MovnzPats1<GR32, GR64, CMOV_NZ_32_64, SLT, SLTU, XOR, zero_64>,
                    Requires<[IsRV64]>;
// condition 64 bit values, source/destination 64 bit value:
defm : MovzPats0<GR64, GR64, CMOV_EZ_32_64, SLT64, SLTU64, zero_64>,
                    Requires<[IsRV64]>;
defm : MovzPats1<GR64, GR64, CMOV_EZ_64_64, SLT64, SLTU64, XOR64, zero_64
                    Requires<[IsRV64]>;
defm : MovnzPats0<GR64, GR64, CMOV_NZ_32_64, SLT64, SLTU64, zero_64>,
                    Requires<[IsRV64]>;
defm : MovnzPats1<GR64, GR64, CMOV_NZ_64_64, SLT64, SLTU64, XOR64, zero_6
                    Requires<[IsRV64]>;
```

Appendix B

Functional testing results

B.1 Test LLVM with ternary if statement

The C program listed in **??** is designed to generate one conditional move per function, except for the main function. The different functions are designed to test all possible conditions. The inline attribute is used to prevent the compiler to statically compute the result. The assembly code produced shows that it produces indeed conditional move statements for every function, see listing **??**. When assembled and run on the simluated FLUTE processor, it produces the result shown in listing B.1.

Listing B.1: Result of the test with ternary if statements

```
result of f_not_equal_1(1,-100): 5, expected: 5
result of f_not_equal_1(-100,-100): 1, expected: 1
result of f_not_equal_1(100,0): 5, expected: 5
result of f_not_equal_1(0,0): 1, expected: 1

result of f_equal_1(1,-100): 1, expected: 1
result of f_equal_1(-100,-100): 5, expected: 5
result of f_equal_1(100,0): 1, expected: 1
result of f_equal_1(0,0): 5, expected: 5

result of f_ge_1(-5, 0): 4, expected: 4
result of f_ge_1(-5, -5): 0, expected: 0
result of f_ge_2(5, 10): 4, expected: 4
result of f_ge_2(5, 1): 0, expected: 0

result of f_gt_1(-5, -5): 4, expected: 4
result of f_gt_1( 5, −5): 2, expected: 2
result of f_gt_2( 5, 10): 4, expected: 4
result of f_gt_2(10,  5): 2, expected: 2

result of f_le_1(1, −5): 4, expected: 4
result of f_le_1(−5, −5): 2, expected: 2
result of f_le_2(6, 5): 4, expected: 4
result of f_le_2(5, 5): 2, expected: 2

result of f_lt_1(−5,−5): 4, expected: 4
result of f_lt_1(−5, 1): 2, expected: 2
result of f_lt_2(5,  5): 4, expected: 4
result of f_lt_2(1,  5): 2, expected: 2
```

B.2 Further LLVM tests

This section shows tests for some special cases and how LLVM can produce conditional moves for them.

B.3 Assembly tests

The assembly program to test `cmovez`: The assembly program after assembling and de-assembling:

```
00000000000106f0 <main>:
   106f0:      00a18193          addi     s1,s1,10
   106f4:      00520213          addi     s2,s2,5
   106f8:      0041828b          cmovez   s3,s1,s2
   106fc:      0071830b          cmovez   s4,s1,s5
   10700:      00008067          ret
   10704:      00000013          nop
   10708:      00000013          nop
   1070c:      00000013          nop
```

The register state of the processor after running the program:

```
GPR  0:  r0        0  ra    10038  s0     2028  s1        a
GPR  4:  s2        5  s3        0  s4        a  s5        0
GPR  8:  s6        0  s7        0  s8        0  s9        0
GPR 12: s10         0 s11        0  sp   100000  tp        0
GPR 16:  v0         0  v1    10f70  a0        0  a1   100008
GPR 20:  a2         0  a3        0  a4        0  a5        0
GPR 24:  a6         0  a7        0  t0    11518  t1    11518
GPR 28:  t2         0  t3        0  t4        0  gp    11ca0
```

The assembly program to test `cmovnz`: The assembly program after assembling and de-assembling:

```
00000000000106f0 <main>:
   106f0:        00a18193                addi    s1,s1,10
   106f4:        00520213                addi    s2,s2,5
   106f8:        0041828b                cmovez  s3,s1,s2
   106fc:        0071830b                cmovez  s4,s1,s5
   10700:        00008067                ret
   10704:        00000013                nop
   10708:        00000013                nop
   1070c:        00000013                nop
```

The register state of the processor after running the program:

```
GPR  0:  r0         0  ra    10038  s0     2028  s1        a
GPR  4:  s2         5  s3        a  s4        0  s5        0
GPR  8:  s6         0  s7        0  s8        0  s9        0
GPR 12: s10         0 s11        0  sp   100000  tp        0
GPR 16:  v0         0  v1    10f60  a0        0  a1   100008
GPR 20:  a2         0  a3        0  a4        0  a5        0
GPR 24:  a6         0  a7        0  t0    11508  t1    11508
GPR 28:  t2         0  t3        0  t4        0  gp    11c90
```

Appendix C

Benchmark descriptions

The following table contains short descriptions for each WCET benchmarks, which were taken ‘from WCET Benchmarks - past, present and future’[**Gustafsson:WCET2010:Benchmarks**].

Test	Description
adpcm	Adaptive pulse code modulation algorithm. Completely well-structured code.
bs	Binary search for the array of 15 integer elements. Completely structured.
bsort100	Bubblesort program. Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.
cnt	Counts non-negative numbers in a matrix. Nested loops, well-structured code.
compress	Data compression program. Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data.
cover	Program for testing many paths. A loop containing many switch cases.
crc	Cyclic redundancy check computation on 40 bytes of data. Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.

Test	Description
duff	Using “Duff’s device” from the Jargon file to copy 43 byte array. Unstructured loop with known bound, switch statement.
edn	Finite Impulse Response (FIR) filter calculations. A lot of vector multiplications and array handling.
expint	Series expansion for computing an exponential integral function. Inner loop that only runs once, structural WCET estimate gives heavy overestimate.
fac	Calculates the faculty function. Uses self-recursion
fdct	Fast Discrete Cosine Transform. A lot of calculations based on integer array elements.
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm. A lot of calculations based on floating point array elements.
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30). Parameter-dependent function, single-nested loop
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample. Inner loop with varying number of iterations, loop-iteration dependent decisions.
insertsort	Insertion sort on a reversed array of size 10. Input-data dependent nested loop with worst-case of $(n^2)/2$ iterations (triangular loop).
janne_complex	Nested loop program. The inner loops number of iterations depends on the outer loops current iteration number.
lcdnum	Read ten values, output half to LCD. Loop with iteration-dependent flow.
lms	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise. A lot of floating point calculations.

Test	Description
ludcmp	LU decomposition algorithm. A lot of calculations based on floating point arrays with the size of 50 elements.
matmult	Matrix multiplication of two 20x20 matrices. Multiple calls to the same function, nested function calls, triple-nested loops.
minver	Inversion of floating point matrix. Floating value calculations in 3x3 matrix. Nested loops (3 levels).
ndes	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.
ns	Search in a multi-dimensional array. Return from the middle of a loop nest, deep loop nesting (4 levels).
nsichneu	Simulate an extended Petri Net. Automatically generated code containing large amounts of if-statements (more than 250).
prime	Calculates whether numbers are prime. Uses integer division and modulo function.
qsort-exam	Non-recursive version of quick sort algorithm. The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.
qurt	Root computation of quadratic equations. The real and imaginary parts of the solution are stored in arrays.
recursion	A simple example of recursive code. Both self-recursion and mutual recursion are used.
select	A function to select the Nth largest number in a floating point array. A lot of floating value array calculations, loop nesting (3 levels).
sqrt	Square root function implemented by Taylor series. Simple numerical calculation.
st	Statistics program. This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.

Test	Description
statemate	Automatically generated code. Generated by the STAtechart Real-time-Code generator STARC.
ud	Calculation of matrixes. Loop nesting of 3 levels.

Appendix D

Evaluation results

Figure D.1: Speedup for SAT and BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

Figure D.2: Branch prediction misses for LHBP and BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

Figure D.3: Speedup for LHBP and BTB512, the base value for each benchmark is the number of misses without cmovs and BTB8

Appendix E

Project Proposal