

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of control engineering

Branch Prediction with Visualization for RISC-V Educational Simulator

Jiří Štefan

Supervisor: Ing. Pavel Píša, Ph.D.
Field of study: Cybernetics and robotics
May 2024

Acknowledgements

I would like to thank both Ing. Pavel Píša, Ph.D. and Ing. Jakub Dupák, for their help and patience, especially during the final weeks of the semester. I would also like to thank all my colleagues, for their extensive moral support, and constant motivation. Lastly, I would like to thank my parents for their kindness and understanding.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 24. May 2024

Abstract

The aim of this thesis is to implement branch prediction and its visualization to the QtRvSim simulator software. Branch prediction is an important part of processor design, as it helps reduce stalling of the processor pipeline, by predicting results of jump and branch instructions, before they are executed. The branch predictor component and its visualization are implemented and tested against other similar implementations.

Keywords: QtRvSim, RISC-V, branch prediction, C++, Qt

Supervisor: Ing. Pavel Píša, Ph.D.

Abstrakt

Cílem této práce je implementovat prediktor skoků a jeho vizualizaci do simulátoru procesoru QtRvSim. Predikce skoků má důležitou roli v návrhu procesoru, protože pomáhá redukovat pozastavení pipeline procesoru předpovídáním výsledku skokových instrukcí, předtím než jsou vykonány. Komponenta predikce skoků a její vizualizace jsou implementovány a porovnány s podobnými implementacemi.

Klíčová slova: QtRvSim, RISC-V, predikce skoků, C++, Qt

Překlad názvu: Návrh a vizualizace prediktoru skoků pro výukový RISC-V simulátor

Contents

Project Specification	1
1 Introduction	3
2 RISC-V Architecture	5
2.1 Instruction set and registers	5
2.1.1 Jump and link instruction	6
2.1.2 Jump and link register instruction	7
2.1.3 Branch instructions	7
2.2 Pipelined processor	8
2.2.1 Branch and jump instructions in pipelined processor	9
3 Branch prediction	11
3.1 Branch Target Buffer	11
3.2 Predictors	12
3.2.1 Static predictors	12
3.2.2 Smith predictors	13
3.2.3 Branch History Register	15
4 Existing implementations	17
4.1 MARS and RARS	17
4.2 QtMips-Di	18
5 Implementation	21
5.1 Code structure	21
5.2 Existing implementation	22
5.3 Internal implementation	22
5.3.1 Interface to the implemented class	22
5.3.2 Branch Predictor components	23
5.4 User Interface implementation	26
5.4.1 Configuration dialog	26
5.4.2 Dock widgets	27
5.4.3 Branch Predictor and BHT Widget	28
6 Testing	31
6.1 For loop	31
6.2 Nested for loops	31
6.3 If conditions	32
6.4 Results	33
7 User manual	35
8 Conclusion	39
8.1 Future work	39
Bibliography	41

Figures

2.1 RISC-V base instruction formats, taken from [1].	6
2.2 Structure of the JAL instruction, taken from [1].	7
2.3 Structure of the JALR instruction, taken from [1].	7
2.4 Structure of branch instructions, taken from [1].	8
2.5 Diagram of pipelined instruction execution, taken from [2].	9
3.1 States and state transitions of Smith 1-bit predictor.	13
3.2 States and state transitions of Smith 2-bit predictor.	14
3.3 States and state transitions of Smith 2-bit hysteresis predictor.	14
4.1 Branch predictor UI widget of the MARS simulator	17
4.2	18
4.3 QtMips-Di BTB (left) and BHT (right) implementation.	19
5.1 Main window of the QtRvSim processor simulator [3].	21
5.2 Simulator configuration dialog	26
5.3 Menu with available dock widgets.	27
5.4 Widget showing the contents of the Branch Target Buffer.	28
5.5 Widget showing the contents of the Branch History Table, as well as other important predictor information.	28
7.1 Configuration dialog with highlighted branch predictor settings.	35
7.2 Menu where the branch predictor widgets can be enabled.	36
7.3 The branch predictor widgets highlighting rows used for prediction during the fetch stage.	37
7.4 The branch predictor widgets highlighting rows that were updated during the memory stage.	38

Tables

3.1 Earliest state where information about jump or branch instruction is available.	12
6.1 Summary of the 1-bit and 2-bit predictor accuracy in the performed tests.	33

I. Personal and study details

Student's name: **Štefan Jiří**

Personal ID number: **474468**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Master's thesis details

Master's thesis title in English:

Branch Prediction with Visualization for RISC-V Educational Simulator

Master's thesis title in Czech:

Návrh a vizualizace prediktoru skok pro výukový RISC-V simulátor

Guidelines:

The QtRvSim is the world wide used educational RISC-V simulator.

- 1) Familiarize with already achieved functionality and implementation of the simulator
- 2) Implement branch target and branch history table and branch predictor logic
- 3) Design illustrative graphic widgets to visualize single-bit and two-bit Smith predictor states, branch history, and target table content.
- 4) Document implemented code and prepare instructions for future B35APO Computer Architectures course students

Bibliography / sources:

- [1] Czech technical University in Prague Computer Architectures guidepost page <https://comparch.edu.cvut.cz/>
- [2] Dupák, J.; Píša, P.; Štepanovský, M.; Koří, K. QtRvSim – RISC-V Simulator for Computer Architectures Classes In: embedded world Conference 2022. Haar: WEKA FACHMEDIEN GmbH, 2022. p. 775-778. ISBN 978-3-645-50194-1.
- [3] PATTERSON, David A. a John L. HENNESSY. Computer organization and design RISC-V edition: the hardware/software interface. Second Edition. Cambridge: Elsevier, [2021]. ISBN 978-0-12-820331-6.

Name and workplace of master's thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **04.09.2023**

Deadline for master's thesis submission: **09.01.2024**

Assignment valid until: **16.02.2025**

Ing. Pavel Píša, Ph.D.
Supervisor's signature

prof. Ing. Michael Šebek, DrSc.
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature



Chapter 1

Introduction

QtRvSim is a graphical RISC-V processor simulator [3] [4], based on QtMips simulator, an older implementation with the MIPS architecture [5]. It is used for courses, education, and practical examples of how a processor works and how it executes instructions. The current implementation lacks branch prediction functionality, which this thesis aims to implement.

A simple implementation of a processor only executes one instruction at a time and only starts executing the next one, after finishing all execution steps of the first instruction. This approach was improved by splitting the execution of instructions into several stages, called a pipeline [2]. To ensure instructions progress through each pipeline stage without stalling, several mechanisms are implemented. One of these is branch prediction, which allows the processor to continue executing instructions, even when it encounters a jump or a branch in the program.

This thesis describes important parts of theory behind RISC-V architecture and pipelined processors in Chapter 2 and branch prediction, in Chapter 3. Then other processor simulators and their predictor implementations are reviewed in Chapter 4. The implementation of the predictor into the QtRvSim simulator is documented in Chapter 5. Finally, the implementation is tested and the results are compared to those of other simulators, in Chapter 6. The user manual describing the intended usage of the newly added features is in Chapter 7.

Chapter 2

RISC-V Architecture

The RISC-V is an open-standard instruction set architecture (ISA) [1]. The RISC-V ISA defines the types, encoding, and function of instructions. Based on this specification, anyone can implement a processor compatible with this architecture. This text focuses on the RV32I 32-bit implementation.

2.1 Instruction set and registers

The processor works by executing instructions read from memory. These instructions are 32-bit long chunks of data, and they carry information about a single operation the processor should perform. Since these instructions can encode a wide array of functionality, like arithmetic operations, program control, and memory access operations, there are several ways to write an instruction, depending on what it does, called instruction types.

The ISA specifies, that in the processor, there are 32 data registers, that can be used to store and access 32-bit chunks of data. The basic use of these registers is to store data loaded from memory, store results of logical and arithmetical operations, and source data to be written into memory.

There is also one more register, called the Program Counter, or PC, which contains the current address in program memory, from where the next instruction will be read. Most of the time, after reading the instruction, the PC is simply incremented by four, to move the counter by four bytes, and arrive at the address of the next instruction in program memory. Some instructions, namely jumps and branches, can modify this counter by overwriting it with a specific address, causing the execution to jump to another part of the program.

Each instruction is split into parts, depending on its type. The basic parts are:

- **opcode** which has seven bits, and contains information about the instruction type, or in other words, how the remaining 25 bits of the instruction should be interpreted.
- **rd** is the 5-bit index of the destination register, for operations that need to be able to store their result.

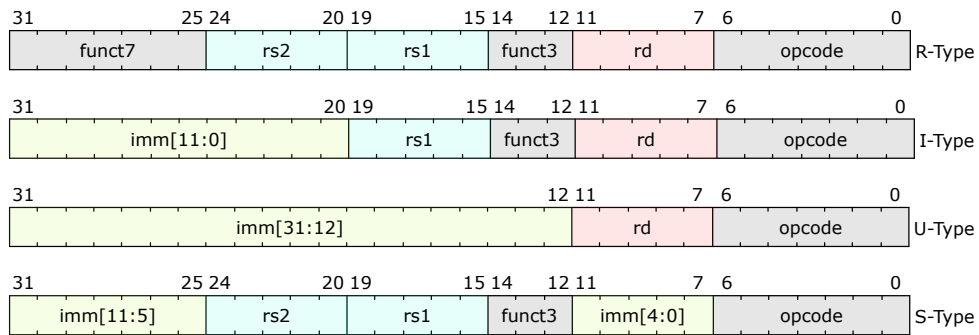


Figure 2.1: RISC-V base instruction formats, taken from [1].

- **rs1** and **rs2** are 5-bit indexes of source registers, which provide 32-bit chunks of data used in the execution of the instructions.
- **imm** is the immediate value used to encode constants directly in the instruction, which are also used during instruction execution, much like the source registers.
- **funct3** and **funct7** are 3-bit and 7-bit chunks that can be used to further specify the type of instruction, in addition to the opcode.

The immediate value encoding might be scattered over multiple parts of the 32-bit instructions but is always resolved into a single integer value.

There are four base instruction types and two derived types. The structure of the four base instruction types can be seen in Figure 2.1. Of these types, three are used to encode jump and branch instructions:

- U-type variant is used to encode the unconditional JAL instruction.
- I-type is used to encode the unconditional JALR instruction.
- S-type variant is used to encode the conditional branch instructions.

It is important to note, that technically, the JAL instruction is J-type encoded, and branch instructions are B-type encoded, but these variants are based on the above-mentioned encodings and do not change the structure of the instruction, besides the ordering of the bits in the immediate part.

■ 2.1.1 Jump and link instruction

The jump and link instruction, or JAL, is an unconditional jump instruction, with J-type encoding, which is a variant of the U-type encoding. The instruction is composed of two parts, not counting the instruction code, and its structure can be seen in Figure 2.2. The first part is the destination register *rd*, the other part is the immediate offset value. When the instruction is executed, the address of the next instruction after the jump instruction is saved into the *rd* register, to serve as a possible return address. The use of the return address provided by the instruction is optional. To get the target

address where the program execution should jump, the address of the jump instruction is summed with the offset. This means the instruction can only jump to an address that is relative to its location in the memory, in the range of $\pm 1\text{MiB}$.

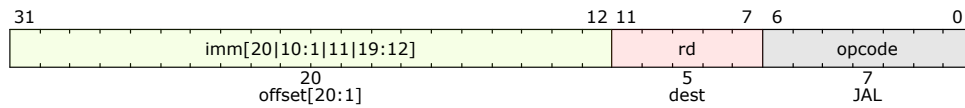


Figure 2.2: Structure of the JAL instruction, taken from [1].

2.1.2 Jump and link register instruction

The jump and link register, or JALR, is an unconditional jump instruction, with I-type encoding. The instruction is composed of three parts, not counting the instruction code and the function selector, which is always zero, and its structure can be seen in Figure 2.2. As with the JAL instruction, the first part of the instruction is the *rd* register, used for storing the next instruction address before the jump. The use of the return address provided by the instruction is also optional. The second part of the instruction is the index of the *rs1* register, which provides the 32-bit base of the destination address. The third part is the offset, which can be used to change the address stored in the *rs1* register without having to change the value in the register itself. This means that the instruction allows a jump to an absolute address, which can be offset by $\pm 2\text{KiB}$.

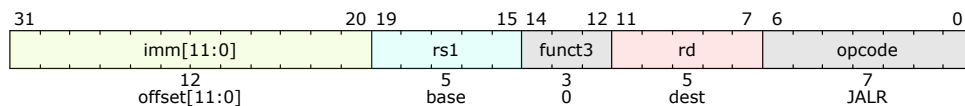


Figure 2.3: Structure of the JALR instruction, taken from [1].

2.1.3 Branch instructions

All branch instructions are implemented using the same instruction opcode and have the B-type encoding, which is a variant of the S-type encoding. The instruction is composed of four parts, not counting the instruction code, and its structure can be seen in Figure 2.4. The first part is the function selector, which denotes the type of comparison to be done. Then there are two registers *rs1* and *rs2* which contain indexes of registers with values to be compared. Then there is the immediate value which, similarly to the JAL instruction, contains the offset relative to the instruction address, where the instruction should jump. This means the instruction can only jump to an address that is relative to its location in the memory, in the range of $\pm 4\text{KiB}$. There are six types of jump instructions:

- BEQ - Branch if the values in the two registers are equal

- BNE - Branch if the values in the two registers are not equal
- BLT - Branch if value if *rs1* is less than value in *rs2*
- BLTU - Unsigned variant of BLT
- BGE - Branch if value if *rs1* is greater than or equal to value in *rs2*
- BGEU - Unsigned variant of BGE

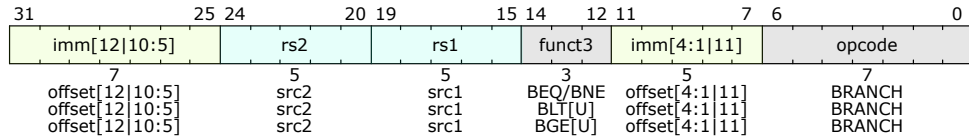


Figure 2.4: Structure of branch instructions, taken from [1].

2.2 Pipelined processor

To speed up the execution of sequences of instructions, and increase the processor clock frequency, pipelined execution is used [2]. The execution of instruction is separated into stages, where each instruction moves from one stage to the next in each clock cycle. There can be as many or as few of these stages as desired, but in general, the simplest pipeline implementation that allows for significant speed improvement, and the one that is generally used in courses and textbooks, is one with five stages, which are described below. Figure 2.5 shows how instructions pass through the pipeline during several clock cycles.

Instruction fetch: In this stage, the only information available is the value of the program counter register (PC) which contains the address in memory where the next instruction to be read and executed is stored. This stage accesses the program memory, reads the instruction, and stores it in a buffer to be used in the next stage, during the next clock cycle.

Instruction decode: In the next clock cycle, the instruction is broken down into parts, where each part denotes some information about the action to be executed. The main part of the instruction is the opcode, which contains the instruction type. If the instruction provides a register whose value should be read, it is read in this stage and stored in a buffer to be used in the next stage. Lastly, any immediate value stored in the instruction is also decoded and passed into the next stage, similarly to the register values.

Execute: In this stage, there is the Arithmetic Logical Unit (ALU) which handles operations between register(s) and immediate values, like addition, subtraction, or logical operations. The source for these operations is two register values, the immediate value from the decode stage, and the address of the instruction read from the program counter during instruction fetch.

Memory Access: In this stage, memory read and write access is done. The input is two register values, one for the memory address, and in case

of write access, one containing the value to be written to the memory. The output of this stage is the value read from memory or a value that bypassed this stage.

Write back: In this stage, any required values are written back to the appropriate register, the input is either the value read from memory or the value passed into the ALU.

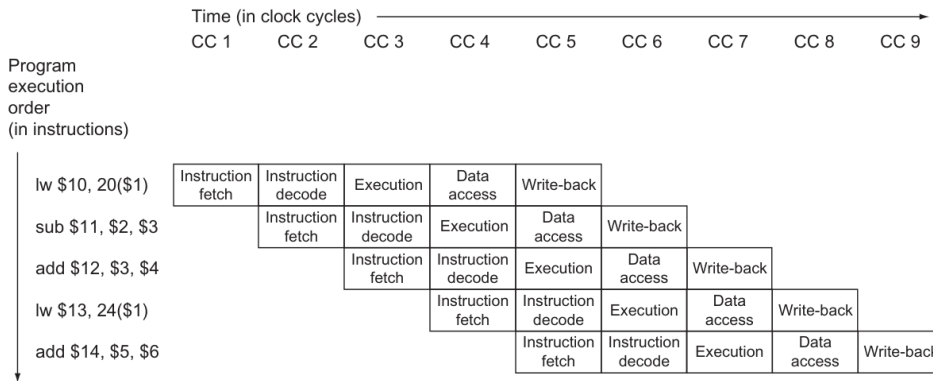


Figure 2.5: Diagram of pipelined instruction execution, taken from [2].

2.2.1 Branch and jump instructions in pipelined processor

Branch and jump instructions introduce so-called control hazards into the processor pipeline [2]. This is caused by the fact, that when the PC is incremented, and another instruction is fetched, but the branch or jump causes a jump to another address, all instructions fetched before this point have to be removed, or flushed, from the pipeline. If this does not happen, instructions that should not have been executed will overwrite values in the memory, or in processor registers, causing unexpected behavior. This is why the processor needs to keep track of the jump and branch instructions, their targets, and the currently fetched instructions.

Chapter 3

Branch prediction

In a pipelined processor, it is important to know which instruction to read from program memory in the fetch stage. In a program with no branches or jumps, the next instruction address is obtained by simply incrementing the program counter by four, to receive the address of the next four-byte instruction. In the event of a jump, however, the next instruction is unlikely to be immediately after the jump instruction.

The jump instruction itself is first fetched, then decoded and executed. The earliest stage where both the jump result and target are known, for all jump instructions, is the start of the memory stage. For branch instructions, both the result and the target are known after the execute stage, as both are the result of addition or comparison. The JAL and JALR instructions have their target known after the address is summed with the offset, earliest at the beginning of the memory stage. Their result, however, is known immediately after decode, at the beginning of the execute stage. This information could be used to optimize the update of prediction. For simplicity, however, it was decided to update results and targets for all instructions in the memory stage. The stage where the jump result and the target address are available can be seen in 3.1.

3.1 Branch Target Buffer

When a jump is predicted as taken, is it necessary to know the target address. However, because the target is always computed in the execute stage, it is not known during the fetch stage of the instruction following a jump or a branch. This means, that after the address is computed, it should be stored in a buffer, where it can be accessed during the fetch stage next time the jump instruction is fetched from memory. This buffer is called the Branch Target Buffer (BTB).

In the memory stage, during the predictor update, the target address is stored in the buffer, along with the address of the jump or branch instruction. If the jump is resolved as not taken, the address is not stored in the buffer, to not remove entries for jumps that are taken. The buffer is accessed using an index which is taken from the lower bits of the instruction address. The two lowest bits of the address are discarded, because the addresses are aligned in

not jumping back only once, when the loop is finished executing.

3.2.2 Smith predictors

The other kind of predictors are dynamic predictors which keep track of some internal state. Of these, the simplest example is the Smith predictor.

1-bit

In its simplest form, the Smith predictor has an internal state represented by a single bit, which tracks the result of the last jump or branch result, and decides if the next prediction will be taken or not. This is called the 1-bit Smith predictor, or a 1-bit saturating counter, and a graphical representation of its state transitions can be seen in Figure 3.1. An example where this predictor can work well is a for loop.

The initial jump back to repeat the loop will be guessed correctly or incorrectly, depending on the initial state. Then the jump result will set the predictor state to "Taken", and it will guess all other jumps correctly, except for the last one, where the loop is finished, and the program will not branch back to repeat the loop. When the for loop starts again, it typically mispredicts due to the state being changed to "Not Taken" by the last result of the previous loop. BTFNT predictor behaves better there since the program execution typically jumps to a lower address after every loop iteration.

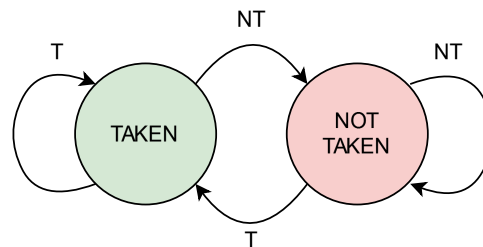


Figure 3.1: States and state transitions of Smith 1-bit predictor.

2-bit

The predictor can have more than one bit assigned to its memory, this is called the 2-bit Smith predictor. In the case of two bits, there are four states, namely "Strongly taken", "Weakly taken", "Weakly not taken" and "Strongly not taken" and the predictor switches between these states with every update. The 2-bit predictor states and their transitions can be seen in Figure 3.2.

The advantage of this predictor is that it takes two consecutive jumps with the same result to change what the predictor will predict. This compensates for the above-mentioned problem of the 1-bit predictor and has the same success rate as BTFNT.

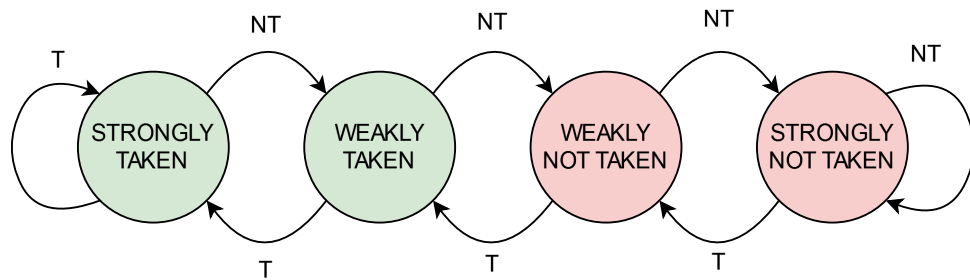


Figure 3.2: States and state transitions of Smith 2-bit predictor.

■ 2-bit with hysteresis

A special case of the Smith 2-bit predictor is one where switching from the "Weakly taken" or "Weakly not taken" always results in a change into the "Strongly taken" or "Strongly not taken" states. This behavior can be seen in Figure 3.3.

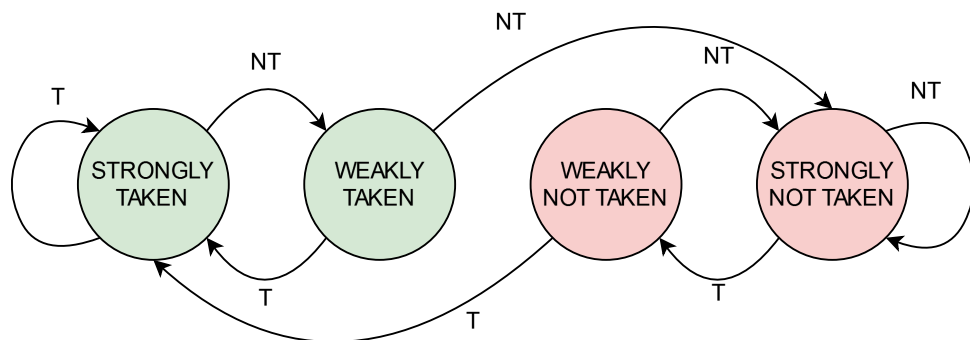


Figure 3.3: States and state transitions of Smith 2-bit hysteresis predictor.

■ Branch History Table

Using just one memory state for all predictions is problematic, as it means that any two different jump or branch instructions will interfere with each other. The result of one might change the state and cause a misprediction of the other instruction, which otherwise would not have happened. For this reason, Smith predictors use something called branch history table, or BHT. It is a one-dimensional table of Smith predictors, as they were described in previous chapters, where each row keeps track of its state. When it is time to make a prediction, the appropriate row is selected, and its state is used to choose the branch result. The primary way to select a row is by using an index, obtained by selecting bits from the instruction address. The process is similar to the indexing of the BTB.

3.2.3 Branch History Register

It is possible for a branch instruction to be dependent on the result of previous branch instructions. That is, whether a branch is taken or not, may be correlated to the result of previous branches.

```
int a;
...
int b = 0;
if (a > 0) {
    b = 1;
}
...
if (b == 0) {
    ...
}
```

In the example C code, the first and second conditions are either both executed, or neither of them are. This correlation can be accounted for by keeping track of the global history of branch results and using this history to select a predictor from the Branch History Table. There are two ways to use the branch history register to select an index.

The simpler way is called Gselect, where the bits of the register are simply appended as the more significant bits, to the bits provided by the instruction address

The other way called Gshare, has the BHR bits XORed together with the instruction address bits. This approach helps distribute the indexes more evenly in the branch history table.

Chapter 4

Existing implementations

In order to implement the desired functionality, other implementations of branch prediction in processor simulators were considered. The older version of this simulator, the QtMips simulator, does not have a functioning branch predictor implementation, so other simulators were checked.

4.1 MARS and RARS

The MIPS Assembler and Runtime Simulator [6], or MARS, is an extensive simulator created at Missouri State University. Its last release was in 2014, and it has since been replaced by the RARS [7], or RISC-V Assembler and Runtime Simulator. The functionality and user interface of the two simulators remains mostly the same, especially with regard to branch prediction. The user interface can be seen in Figure 4.1.

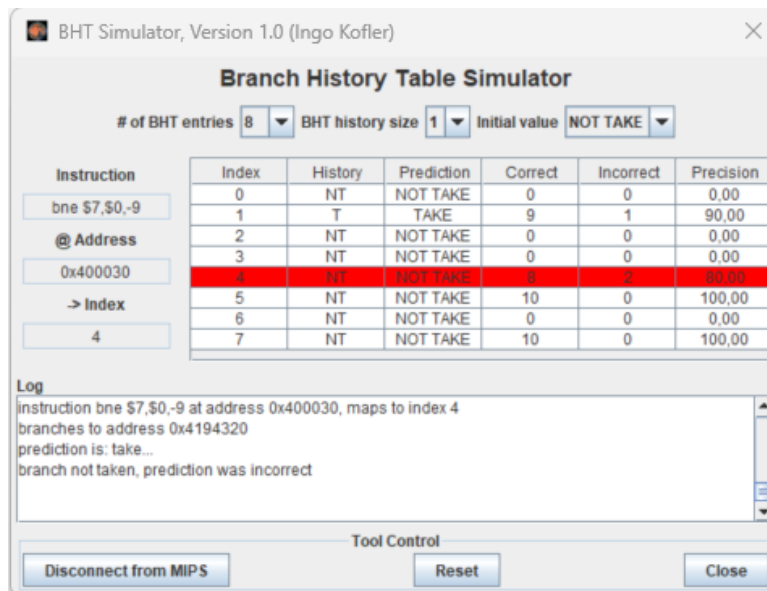


Figure 4.1: Branch predictor UI widget of the MARS simulator

The simulator seems to lack a view of the Branch Target Buffer and only provides the Branch History Table view. In there, the user can select between a 1-bit and a 2-bit local history-based predictor, its initial state, and the number of BHT entries. During program execution, the fields on the left update along with the table, and detailed description of what is happening is contained in the log view. One interesting thing to note is, that the history of the 2-bit predictor is shown as the result of two last jumps, instead of its internal state. The BHT also keeps track of prediction statistics of individual entries, but not the total statistics of the whole predictor.

4.2 QtMips-Di

A second example of branch predictor implementation was found in the older fork of the QtMips project, the QtMips-DI [8] simulator. This implementation contains a view of both the branch target buffer and the branch history table, both can be seen in Figure 4.3. It also contains a configuration screen, shown in Figure 4.2. In some respects, it is an improvement compared to the MARS and RARS simulators, but the implementation contains some strange design choices and it lacks visual clarity of the implemented widgets when compared to the MARS BHT implementation.

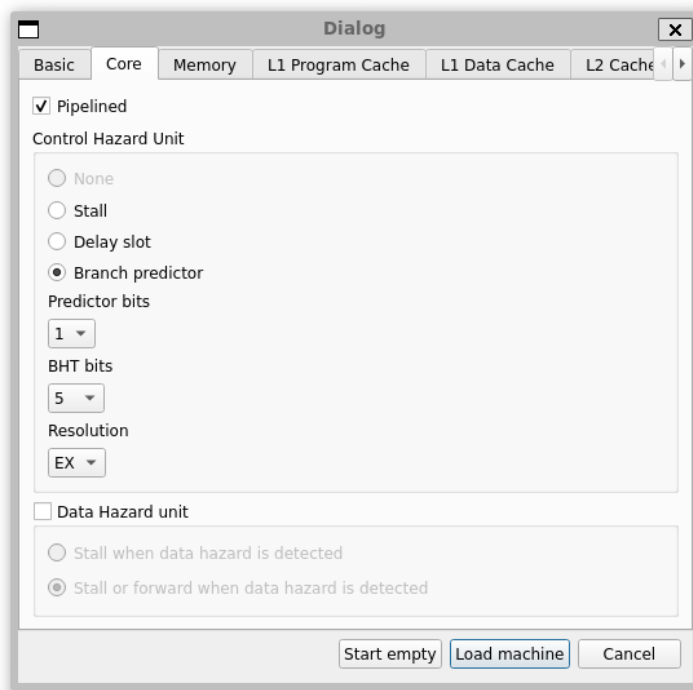


Figure 4.2:

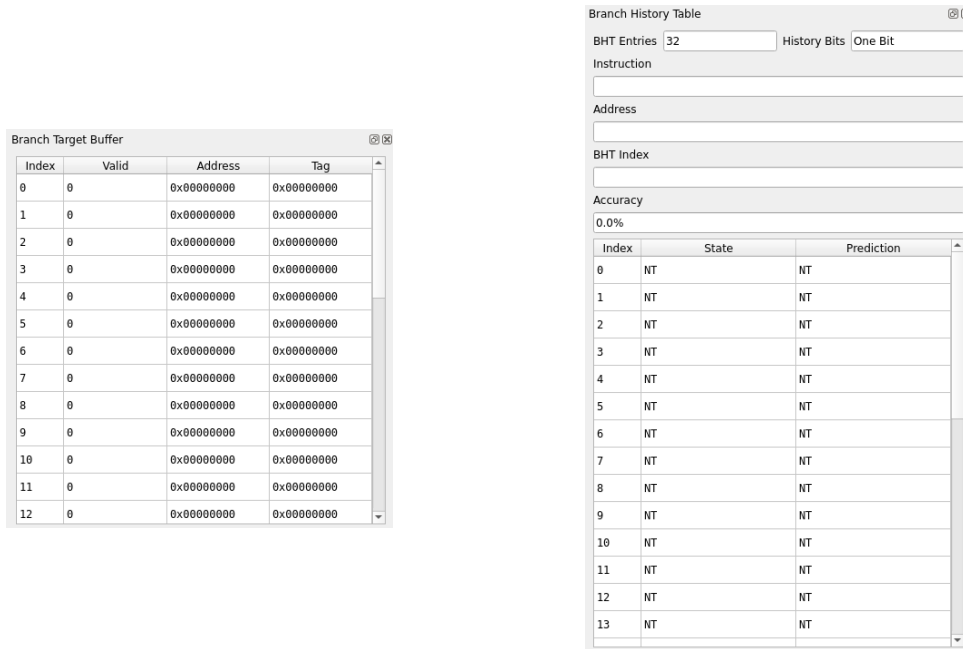


Figure 4.3: QtMips-Di BTB (left) and BHT (right) implementation.

The user can again, only choose between 1-bit and 2-bit Smith predictor, but this time, the smallest size of BHT is 5 bits, which can lead to unnecessarily large tables when the simulator is only used to run short programs with few jump instructions. Another missing feature is that the predictor does not seem to keep track of how many predictions happened, and it does not keep track of the accuracy of individual BHT entries. The only available statistic is the total accuracy. The meaning of the "Resolution" selector is not clear from the user interface alone. The implementation also seems to stall unexpectedly, when running the pipelined version of the processor.

Chapter 5

Implementation

QtRvSim is an open-source RISC-V simulator [9]. It is implemented in C++ language extended by the Qt framework libraries. The main screen of the simulator can be seen in Figure 5.1.

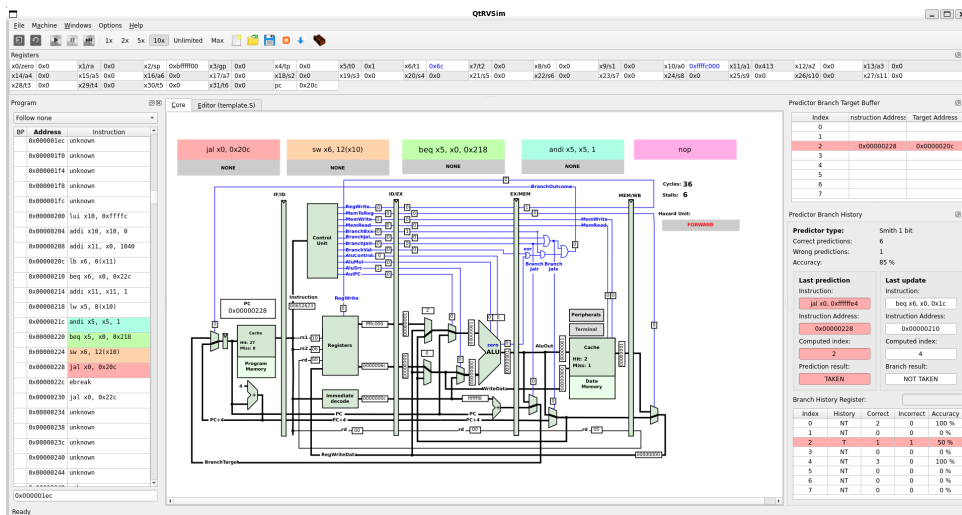


Figure 5.1: Main window of the QtRvSim processor simulator [3].

5.1 Code structure

The first task was to get familiar with the structure of the project. The first code that is run is the main function in the `main.cpp` file. An instance of the class `MainWindow` is created there, which contains all the internal logic, all the user interface, and handles communication between the two. The first relevant part of the user interface is a dialog window which is used to set up and start the simulation.

Then, there are dock widgets, which contain information about various parts of the simulated processor, like the value of registers, statistics of cache, and so on. New dock widgets will have to be added to show the state and statistics of the implemented predictor. In the internal logic of the program, starting a simulation creates an instance of the `Machine` class, which in basic

For JAL and branch instructions, the target address is taken from a separate addition that happens in the execute stage, but for JALR instruction, the target has to be retrieved as a result of the ALU. Similarly, for the branch instruction, the result of the jump has to be retrieved from the ALU as the result of a comparison, but for the JAL and JALR instructions, it is known automatically.

Some types that are required outside of the `BranchPredictor` class were defined in a separate file, `preditor_types.h`, to simplify importing in different parts of the project.

■ 5.3.2 Branch Predictor components

The `BranchPredictor` wrapper class contains three main parts:

- `BranchTargetBuffer` keeps track of instruction-target address pairs.
- `Predictor` implements the guessing of whether a branch will be taken or not.
- `BranchHistoryRegister` keeps track of global jump and branch history.

■ Branch Target Buffer

A fundamental part of the implementation is the `BranchTargetBuffer` class, which implements the buffer as a vector of struct values, where each value represents one entry, or one row, in the table. The interface of this class are the functions `get_instruction_address(...)` and `update(...)`. The first two only take the instruction address as an argument and use it to index the table to read the correct row and return the instruction or the target address respectively. The update function also takes the instruction address as an argument to index the table, but it also requires the corresponding target address it should write back into the table.

The function used to calculate the index from the instruction address can be seen in Listing 5.1. All bits of the address are first shifted by two bits to the right since all instructions in the 32-bit RISC-V architecture are aligned to 4-byte memory words, and so the two least significant bits hold no useful information in this case. Then a mask is created by placing a single bit above the most significant bit of the required part of the address, and subtracting one from it, resulting in a mask that has n ones and then all zeroes. The shifted address and the mask are then simply combined using the logical AND operation, and the result is the index.

Listing 5.1: Index calculation for the BTB

```

1 uint16_t calculate_index(Address instruction_address) {
2     return (
3         (uint16_t)(instruction_address.get_raw() >> 2))
4         & ((1 << number_of_bits) - 1
5     );

```


- Always taken predictor, will always predict all instructions stored in the BTB to be taken.
- Always not taken corresponds to the original functionality.
- Backward taken forward not taken, or BTFNT, will compare instruction and target address, and if the jump is backward in memory, it will predict it as taken, otherwise it will predict it as not taken.
- Smith 1-bit
- Smith 2-bit
- Smith 2-bit with hysteresis

The Always taken, Always not taken and BTFNT predictors are static, since they use only immediately available information to make a prediction. The three Smith predictors are dynamic, since they keep track of branch history, and use it to make the next prediction.

The index of the branch history table is created by combining bits from the instruction address and the branch history register. The principle is similar to the branch target buffer, shown in 5.1. The only difference is that the bits of the BHR were added before the address bits using the OR logical operation to create the index.

■ Machine Config

To simplify configuration by bundling all relevant configuration parameters together, a `MachineConfig` is already provided in the simulator implementation. It was necessary to extend this class with branch predictor parameters. The added parameters are

- `bool bp_enabled` - To keep track of whether the branch predictor should be enabled.
- `PredictorType bp_type` - Stores the predictor type as an enumeration.
- `PredictorState bp_init_state` - Stores the initial state and is based on the predictor type.
- `uint8_t bp_btb_bits` - Stores the number of BTB bits.
- `uint8_t bp_bhr_bits` - Stores the number of BHR bits.
- `uint8_t bp_bht_addr_bits` - Stores the number of BHT bits that are to be extracted from the instruction address.
- `uint8_t bp_bht_bits` - Stores the number of BHT bits, and is equal to the sum of the number of BHR bits and the number of BHT address bits.

5.4 User Interface implementation

The user interface was partly implemented using the QtCreator graphical design tool, in the case of the configuration dialog, and partly implemented directly as instances of Qt objects in code, in the case of both dock widgets. The communication between the user interface and the internal implementation was done using the system of signals and slots provided by the Qt framework.

The internal classes contain special functions labeled as "signals", which act as events. They are usually called in the parts of the program where some action has finished, and the user interface has to be notified, and/or has to receive some updated data. An example is a signal which is called when a prediction happens, to notify the interface that it should show the prediction result to the user.

The other end of this data exchange is called a slot, and it is essentially a handler function which is called every time the signal is called. Each signal has to be connected to a relevant slot somewhere during interface setup, using the `connect(...)` function.

5.4.1 Configuration dialog

When the simulator starts, a configuration dialog window opens, allowing the user to set simulation parameters. In the "Core" tab a new section was added dedicated to the branch predictor settings. An overview of this section can be seen in Figure 5.2.

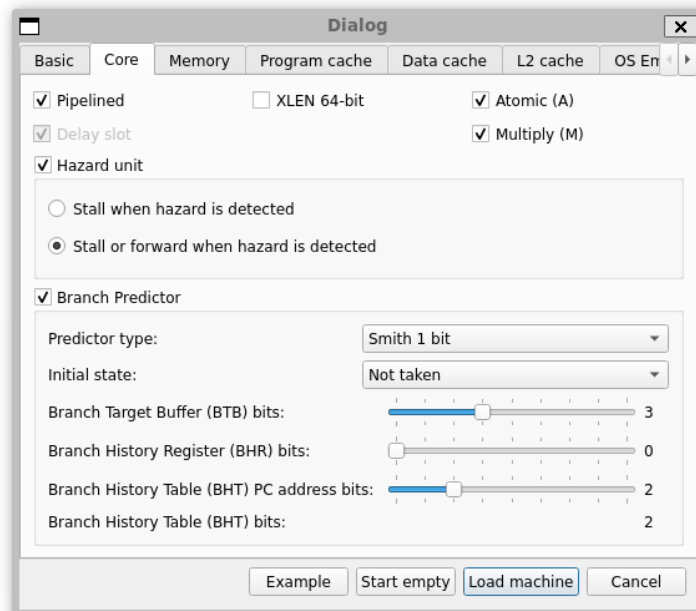


Figure 5.2: Simulator configuration dialog

The configuration dialog is used to set predictor values, as they are described in Section 5.3.2.

5.4.2 Dock widgets

The user interface was implemented as a set of dock widgets. Two new entries were added to the "Windows" menu to open the BTB and BHT widgets, called "Branch Predictor (Target table)" and "Branch Predictor (History table)" respectively. The entries are shown in Figure 5.3. Clicking on them opens their respective widgets on the right side of the simulator window, similar to the existing terminal and memory dock widgets.

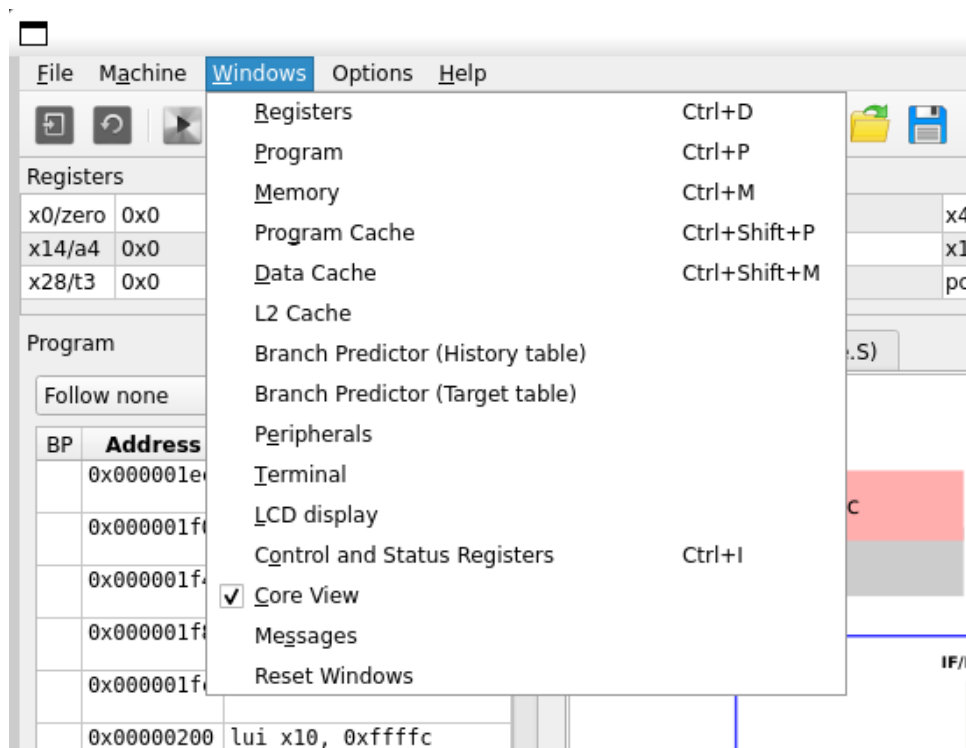
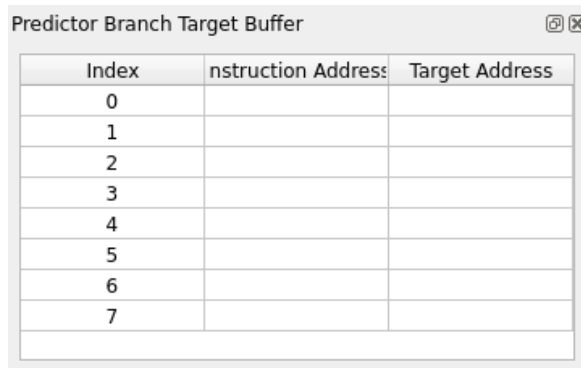


Figure 5.3: Menu with available dock widgets.

BTB Widget

This dock widget contains a table that shows the contents of the BTB and can be seen in Figure 5.4.

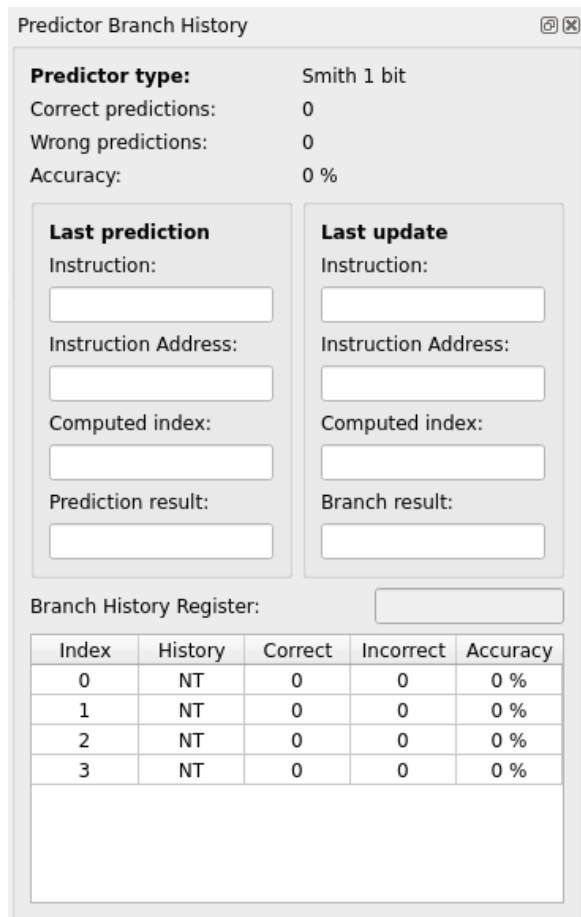


Index	Instruction Address	Target Address
0		
1		
2		
3		
4		
5		
6		
7		

Figure 5.4: Widget showing the contents of the Branch Target Buffer.

5.4.3 Branch Predictor and BHT Widget

This dock widget contains a table that shows the contents of the BHT, predictor information and statistics, and the last performed prediction and update. It can be seen in Figure 5.4.



Predictor type: Smith 1 bit

Correct predictions: 0

Wrong predictions: 0

Accuracy: 0 %

Last prediction

Instruction:

Instruction Address:

Computed index:

Prediction result:

Last update

Instruction:

Instruction Address:

Computed index:

Branch result:

Branch History Register:

Index	History	Correct	Incorrect	Accuracy
0	NT	0	0	0 %
1	NT	0	0	0 %
2	NT	0	0	0 %
3	NT	0	0	0 %

Figure 5.5: Widget showing the contents of the Branch History Table, as well as other important predictor information.

At the top of the widget, the predictor type and its total statistics can be seen. Both the last prediction and last update sections contain the instruction shown as a string, its address, the index of the BHT, and the result of the prediction or the branch. Below is the field containing the branch history register value. At the bottom of the widget is the branch history table. The "History" column shows the internal state of the predictor at that index, as described in Section 3.2.2. Then, for each row, the number of correct and incorrect predictions is tracked, as well as the accuracy of that specific entry.

Chapter 6

Testing

The implementation was tested on several example codes, against similar code run in the RARS [7], MARS [6], and QtMips-Di [8] simulators. The syntax of the programs was changed slightly, to make a version for the MIPS architecture, but structurally it remained the same. The tests were run with both the Smith 1-bit predictor and 2-bit predictor and with 5-bit BHT, as that is both the largest value supported by the MARS and RARS simulators and the smallest value supported by the QtMips-Di simulator. The initial state was selected as not taken, for 1-bit, and strongly not taken, for the 2-bit predictor, as the QtMips-Di does not allow configuring the initial state and defaults to this configuration.

6.1 For loop

Shown in 6.1 is a simple for-loop with 10 jumps. It is expected the 1-bit predictor will make a misprediction at the beginning when the condition is taken, and at the end when the condition is not taken. This is confirmed when running the test on all simulators, the result is 80% in all cases. A 2-bit predictor starting at a strongly not taken state is expected to make one more misprediction at the beginning, as it passes the weakly not taken state. This is again confirmed in all implementations, with accuracy of 70%.

Listing 6.1: For loop

```
1   addi a0, zero, 10
2 loop:
3   addi a0, a0, -1
4   nop
5   bne a0, zero, loop
6   nop
```

6.2 Nested for loops

Shown in 6.2 is a nested for-loop with 5 jumps each. The 1-bit predictor behaves as before, causing one misprediction at the beginning and end of each

loop, resulting in 80%. The 2-bit predictor improves the accuracy of the inner loop, by not switching prediction immediately after the inner loop is finished. Instead, it switches to a weakly taken state and predicts the beginning of the second run of the outer loop correctly. The total accuracy is 86%, where the inner loop had 88% accuracy and the outer loop had 70%, the same as in the previous example.

Listing 6.2: Nested for loop

```

1   addi a0, zero, 10
2   nop
3   outer_loop:
4   addi a1, zero, 10
5   nop
6   inner_loop:
7   addi a1, a1, -1
8   nop
9   bne a1, zero, inner_loop
10  addi a0, a0, -1
11  nop
12  bne a0, zero, outer_loop
13  nop

```

6.3 If conditions

Shown in 6.3 is a for-loop with 10 jumps with three if conditions nested inside, where the last one is dependent on the result of the first two. The dependency of the if conditions is not important during this test, as the registers are preloaded with constant values. Rather a larger number of branch instructions in one program was tested. The result of this test was also the same for all tested simulators, accuracy of 87.5%.

Listing 6.3: Dependent if conditions

```

1   addi a0, zero, 2
2   addi a1, zero, 2
3   addi a2, zero, 2
4   addi a3, zero, 10
5   loop:
6   nop
7   if1:
8   bne a1, a0, if2
9   nop
10  if2:
11  bne a2, a0, if3
12  nop
13  if3:
14  beq a1, a2, ifend

```

```

15     nop
16 ifend:
17     addi a3, a3, -1
18     bne a3, zero, loop
19 end:
20     nop

```

6.4 Results

All tested simulators provided the same results for the tested code, and the summary of prediction accuracy can be seen in Table 6.1. These results match the expectations of how the 1-bit and 2-bit predictors operate. It also serves as a simple check of the correctness of the implemented predictor.

Program	1-bit	2-bit
For loop	80 %	70 %
Nested for loops	60 %	67 %
If conditions	93 %	88 %

Table 6.1: Summary of the 1-bit and 2-bit predictor accuracy in the performed tests.

Chapter 7

User manual

When starting the simulator, the user is greeted with a configuration dialog. To use the branch predictor, open the "Core" tab, pictured in Figure 7.1, and enable the "Branch Predictor" checkbox (1). When the branch predictor is turned off, all branches are assumed to be not taken, and the next instruction is loaded from the memory, every time.

Below, the predictor type can be selected (2) from six implemented types:

- Always taken
- Always not taken
- Backward taken forward not taken
- Smith 1-bit
- Smith 2-bit
- Smith 2-bit with hysteresis

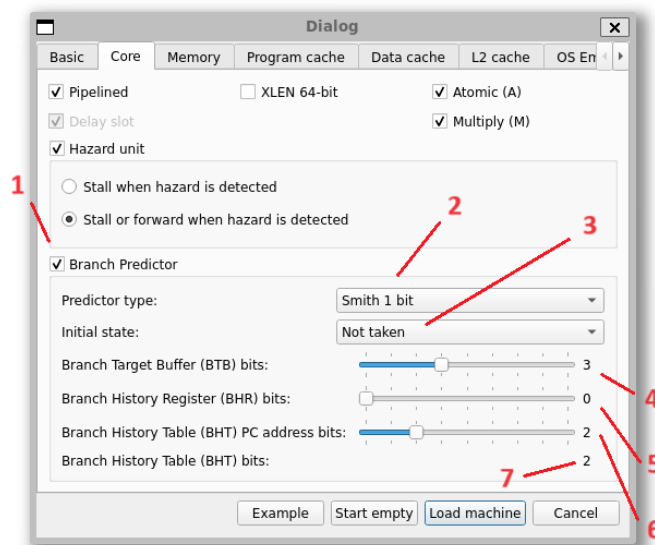


Figure 7.1: Configuration dialog with highlighted branch predictor settings.

Depending on the type of predictor selected, the initial state can also be selected (3). This field is only relevant for Smith predictors, as the first three predictors are static, and do not keep track of history.

The branch target buffer will be used to keep track of destination addresses for encountered jump and branch instructions. The user has to specify the number of bits the table will have using the provided slider (4). If n is the number of selected bits, the number of rows in the BTB will be 2^n .

In case the Smith predictors were selected, the user can also select the number of Branch History Register bits (5) and the number of bits from the instruction address used to index the Branch History Table (6). Values from the sliders (5) and (6) are summed, and the result is displayed below (7). This value is the total number of Branch History Table bits, and the number of rows is again 2^n , where n is the number of bits. This is because the predictor uses the Gselect principle, where the bits from the BHR and the instruction address are simply appended to form the BHT index.

When the simulation is started, the branch predictor information can be shown in the View section, by opening the Branch predictor (History Table) and Branch predictor (Target Buffer) dock widgets, shown in Figure 7.2. The target buffer is represented by a table where each row contains the appropriate index, and then the instruction/target address pair.

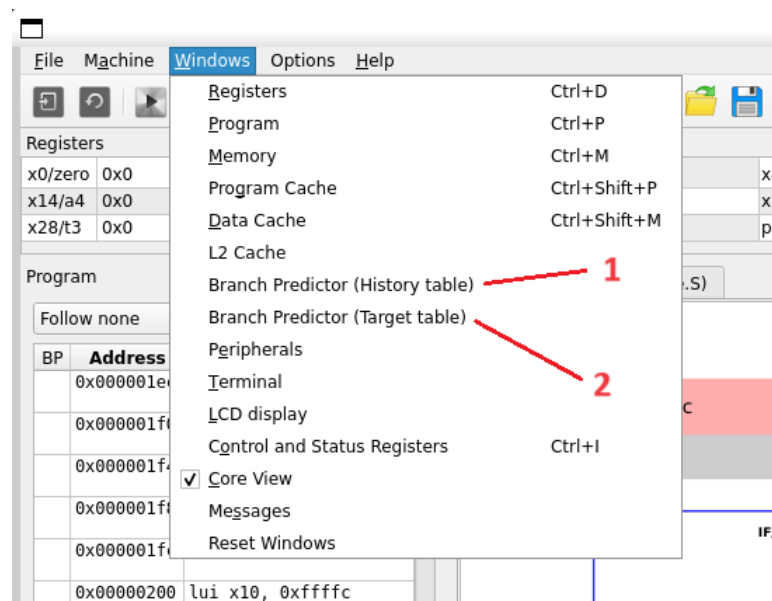


Figure 7.2: Menu where the branch predictor widgets can be enabled.

The history table contains more information about the predictor. At the top of the widget, there are total predictor statistics, that is how many correct and incorrect predictions the predictor made globally. Below are two columns, one for tracking information about the last prediction, and one for tracking information about the last update. Below is the current state of the branch history register. At the bottom, there is the branch history table, showing the current state at a given index, and statistics for that state.

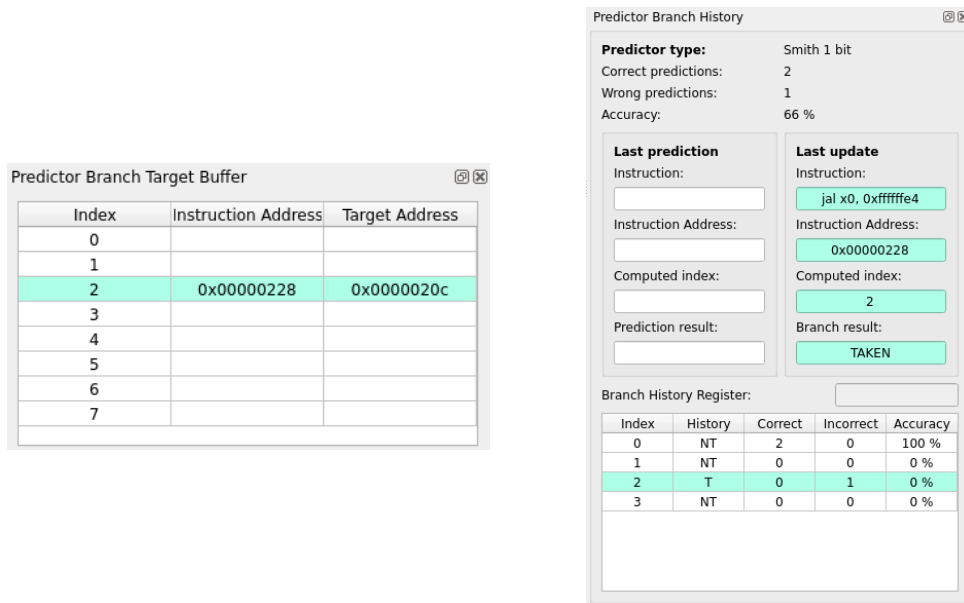


Figure 7.3: The branch predictor widgets highlighting rows used for prediction during the fetch stage.

When the user starts a program that encounters a jump or branch instruction, initially, no prediction will take place. This is because the target address is not yet stored in the target buffer, and so not only the predictor would not know where to jump, but it also does not know whether the currently fetched instruction is a jump or a branch instruction. Once the instruction reaches the memory stage, both the BTB and the BHT are updated using the information from the resolved instruction. This update can be seen in Figure 7.3. The appropriate rows and text boxes are highlighted, and the information is updated. It is important to note that the branch target buffer is only updated if the result of the instruction is for it to be taken. Otherwise, only the BHT is updated, to not potentially overwrite another instruction stored in the BTB, which was taken and is needed.

When the stored instruction is fetched next time, the BTB and BHT rows are also highlighted, this time with the rows used to make the prediction and get the target address. This behavior can be seen in Figure 7.4

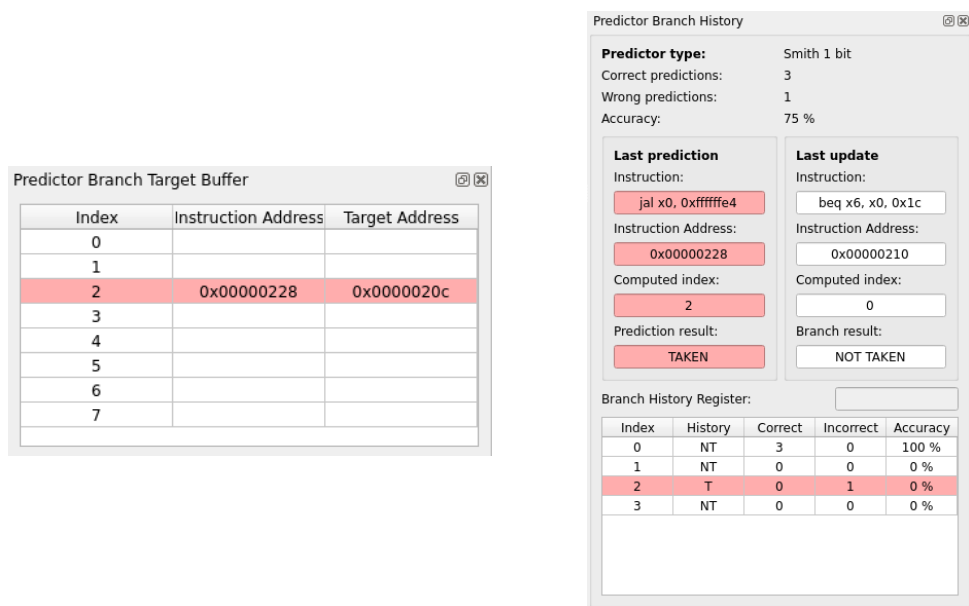


Figure 7.4: The branch predictor widgets highlighting rows that were updated during the memory stage.

Chapter 8

Conclusion

The aim of this project was to implement branch predictor into the existing codebase of the QtRvSim project. Several other simulators and their branch predictor implementations were considered and tested. The explored implementations were found to be lacking certain basic features desirable for education and learning purposes. The new branch predictor was implemented with these features in mind.

The result of this project is a working branch predictor implementation in the QtRvSim open-source project. The implementation was documented and consisted of two main parts. The internal implementation contains six types of predictors, as well as necessary configurations, like their initial state or addressing bits. The user interface consists of three main parts, the branch predictor section in the configuration dialog window, a widget with BTB contents, and a second widget, containing other predictor information, such as statistics and the BHT.

A pull request was opened to merge the implementation into the project. The code was reviewed, and several minor issues regarding the code quality were pointed out. One major issue was found with the branch history register implementation. The index created from the BHR is not in sync between the fetch and the memory stage, causing any updates in between that change the BHR to result in the wrong BHT row being updated. This issue will be fixed before the feature is merged.

8.1 Future work

There is potential for future improvements in the implementation. Since the Branch History Register was implemented as a part of this thesis, the feature could be extended by allowing the user to select between the Gshare and Gselect BHT index computation. Currently, only Gselect is available.

Another feature that might hold at least some value could be the possibility of disabling control hazard detection. The reason is to showcase what happens if the processor does not keep track of incorrect prediction results and does not flush the pipeline when necessary.



Bibliography

- [1] *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. 2019.
- [2] D. A. Patterson and J. L. Hennessy, *Computer organization and design RISC-V edition: the hardware/software interface*. Cambridge: Elsevier, second ed., 2021.
- [3] J. Dupák, P. Píša, M. Štepanovský, and K. Kočí, “QtRVSim - RISC-V Simulator for Computer Architectures Classes,” Haar: WEKA FACHMEDIEN GmbH, 2022.
- [4] J. Dupák, “Graphical RISC-V Architecture Simulator - Memory Model and Project Management.” <https://dspace.cvut.cz/handle/10467/94446>, 2021.
- [5] K. Kočí, “Grafický simulátor činnosti procesoru a činnosti vyrovnávací paměti.” <https://dspace.cvut.cz/handle/10467/76764>, 2018.
- [6] “Mars.” <https://courses.missouristate.edu/KenVollmar/MARS/index.htm>, cited: 24.5.2024.
- [7] “Rars.” <https://github.com/TheThirdOne/rars>, cited: 24.5.2024.
- [8] “Qtmips-di.” <https://github.com/kchasialis/QtMips-Di>, cited: 24.5.2024.
- [9] “Qtrvsim.” <https://github.com/cvut/qtrvsim>, cited: 24.5.2024.