BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

GRAPHICAL SIMULATOR OF SUPERSCALAR PROCESSORS

GRAFICKÝ SIMULÁTOR SUPERSKALÁRNÍCH PROCESORŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN VÁVRA**
AUTOR PRÁCE

SUPERVISOR **Doc. Ing. JIŘÍ JAROŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2020/2021

Master's Thesis Specification

Student: **Vávra Jan, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Embedded Systems
Title: **Graphical Simulator of Superscalar Processors**
Category: Computer Architecture
Assignment:

1. Familiarize yourself with the architecture of current superscalar processors.
2. Review existing graphical simulators of superscalar processors.
3. Design a graphical user interface to visualize the operation of superscalar processor. Focus on out-of-order instruction issue, register rename, branch prediction, and reorder buffer. Take into consideration possible customization of the processor configuration (number of registers, the size of issuing window, reorder buffer, etc.).
4. Implement designed solution.
5. Implement a way to input user source code written in assembly or other higher languages.
6. Evaluate and discuss about the usability and illustrative nature of developed simulator.

Recommended literature:

- According to supervisor's advice.

Requirements for the semestral defence:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Jaroš Jiří, doc. Ing., Ph.D.**
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: April 22, 2021

Abstract

The focus of this thesis is implementation of the superscalar simulator. The implementation follows research of existing simulators and tries to implement missing features from them. Simulator uses RISC-V instruction set architecture, but architecture can be swapped for any RISC instruction set. Simulator implements deterministic branch prediction. Parts of the simulation can be configured. The simulator application also contains a text editor for inputting source code.

Abstrakt

Práce se zabývá implementací simulátoru superskalárního procesoru. Implementace se odvíjí od existujících simulátorů a jejich chybějících částí. Simulátor umí vykonávat instrukční sadu RISC-V, ovšem je umožněno přidání jakékoli RISC instrukční sady. Simulátor má deterministickou predikci skoku. Části procesoru lze upravovat. Součástí je i editor kódu pro danou instrukční sadu.

Keywords

simulator, superscalar, processor, interactive, Java, OOP, RISC-V, branch prediction, Gshare, Tomasulo algorithm, data hazards, load bypassing, load forwarding

Klíčová slova

simulátor, superskalární, procesor, interaktivní, Java, OOP, RISC-V, predikce skoku, Gshare, Tomasulo algoritmus, datové konflikty, load bypassing, load forwarding

Reference

VÁVRA, Jan. *Graphical Simulator of Superscalar Processors.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Jiří Jaroš, Ph.D.

Rozšířený abstrakt

Vysvětlení jistého vnitřního chování u určitých objektů je poněkud obtížné bez jakékoli vizualizace, ať se to týká buď lidského těla, automobilového motoru, nebo funkcionality procesoru. Jednou z cest je využití spousty obrázků a animací, nicméně toto jsou statické objekty, ukazující pouze jeden konkrétní problém. Mnohem lepší cestou je vytvořit simulátor, který napodobuje chování daného systému, se kterým lze interagovat, ať je to buď ubráním či přidáním jistých komponent nebo změnou vstupních proměnných.

Cílem této práce je tedy napsat simulátor, který by simuloval chování určitého superskalárního procesoru. Tento simulátor by měl umožnit vkládat vlastní zdrojové kódy, psané v jazyce symbolických instrukci (assembler), jejichž chování by následně bylo zobrazeno ve výstupu simulátoru. Určité komponenty simulátoru, například velikost seřazovací paměti nebo počet funkčních jednotek, by měly být nastavitelné, aby umožnily uživateli pozorovat chování různých nastavení. Další věcí, kterou by měl simulátor obsahovat, je simulace větvení instrukcí a vykonávání paměťových instrukcí mimo pořadí, což ve většině existujících simulátorů chybí nebo je řešeno stochasticky.

V rámci této práce bylo třeba nastudovat instrukční závislosti, chování jednotlivých komponent superskalárního procesoru, přístupy pro predikci skoků a vykonávaní paměťových instrukcí. Dále bylo třeba vybrat vhodný procesor, jehož algoritmy jsou volně přístupné a sloužily by jako ukázkový příklad pro studenty kurzu AVS. Také bylo zapotřebí projít již existující simulátory a zhodnotit jejich silné a slabé stránky, kde chybějící části by byly použity jako inspirace pro vyvíjený simulátor. Na základě této rešerše byl sestaven návrh, kde byly vytyčeny body, které by výsledná aplikace měla splnit.

Výsledkem je simulátor postavený na algoritmech používaných procesorem RISC-V BOOM, který používá instrukční sadu RISC-V. Aplikace simuluje například přejmenování registrů pomocí Tomasulova algoritmu, predikci skoků pomocí metody Gshare a load forwarding a bypassing. Simulátor má i konfigurovatelnou instrukční sadu a soubory registrů, k jejichž zápisu a uchování byla použita notace JSON. V rámci simulátoru lze vkládat vstupní kód pomocí předpřipraveného textového editoru přímo v aplikaci, který umožňuje zvýrazňování klíčových slov.

Rozšířením této práce by mohlo být okno pro zobrazení paměti, kde by byl pozorovatelný obsah jednotlivých buněk paměti o velikosti 1-4 byte. Dalším vizuálním rozšířením by mohl být mód pro vizualizaci posílání instrukcí mezi jednotlivými částmi procesoru, jako je například vytvořeno v aplikaci Cisco Packet Tracer.

Graphical Simulator of Superscalar Processors

Declaration

I hereby declare that this Diploma's thesis was prepared as an original work by the author under the supervision of assoc. prof. Ing. Jiří Jaroš Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jan Vávra
May 18, 2021

Acknowledgements

I would like to thank my supervisor Jiří Jaroš for his kind and patient attitude throughout the duration of the task and for all of his advices, and remarks, without which this work wouldn't be possible.

Contents

Chapter 1

Introduction

Explaining the inner workings of certain objects to people, who know nothing about them, might be a complicated task. Even after showing the basics and familiarizing them with what goes "under the hood", many people might still not understand, because they need to see it in action. This is the problem many teachers face during their lectures. But there is a solution. What if there is a program, that allows people to interact with the object and see how it behaves?

These apps are called simulators. There are many such apps, which show how an engine of a car works, how a certain circuit is interconnected and what voltage is on each connection, and so on. The same goes for a processor architecture, where students could play with instructions and see how they are passing through each part, how does it behave, etc. The problem is that finding a proper simulator might be tricky because not all of them have all features required for a certain subject focus. In this thesis, I shall focus on implementing such a simulator for a superscalar processor.

The goal of this thesis is to create a simulator, that is based on a modern superscalar processor, where the processor state of each step of the execution can be seen. It needs to focus on the out-of-order program execution and the releasing in the correct order. The simulator needs to allow some sort of configuration, whether it is size is the buffers, initial state, or by adding more function units if needs be. The simulator should allow compiling the user-created source code in an assembly language of chosen architecture. Also, the architecture should be allowed to be extended, letting the users extend and play with it at will.

The thesis is divided into separate chapters. In the chapter 2, there is a summary of how scalar and superscalar processor works, form what parts it is built and how they interact. In the chapter 3, I go over existing simulators and review each one of them, focusing on how the simulation works, and existing and missing features. In the chapter 4, I introduce the proposition of my solution for implementing a superscalar simulator.

Chapter 2

Summary of processor architectures

This chapter summarizes the required knowledge to understand the pipelining process, the difference between the scalar and the superscalar processor, used algorithms inside the superscalar processor, and some examples of processors.

2.1 Pipelining

The simplest processors, called subscalar processors, executed instructions one after the other while waiting for the instruction before finishing. So if we create an example with 2 instructions, where both of them take 6 steps to finish, executing a single two instruction program would take 12 steps in total. The clocks per instruction would be $(6+6)/2 = 6$ CPI. This approach was way too restrictive. But there was a way to execute these instructions in parallel, more precisely the steps were overlapped between consecutive instructions, which coined the term *pipelining*.

The pipeline processors, also called scalar processors, were able to significantly shorten the execution time of programs by dividing processors into stages that could be done separately. So we could be loading a new instruction while also computing another one in another stage. If we got back to the example, we would move from 12 steps to just 7 steps, so the CPI would be $(6 + 1)/2 = 3,5$ CPI, which is a 58% speedup compared to the subscalar processor.

The execution steps are similar to the steps in the von Neumann machine model being in order: fetching the next instruction, decoding it, executing it, saving the result, and moving to the next instruction. For pipelining to be effective, it needs to suffice some conditions. Namely, the steps should approximately take a similar time to execute, the instruction needs to go through all stages to utilize all stages, the stage should be able to hold its results, because not all states take the same time to finish, and all stages should be able to react to disruption in the execution and be able to save the state of the pipeline, so when the interrupt is removed, the pipeline can continue from a consisted state. [4] [7]

Pipeline stages

When talking about pipeline processors, we usually refer to the five-stage instruction execution pipeline. The basic schematics of such a pipeline can be seen in the Figure 2.1. The

stages are from left to right as follows: Instruction fetch, Instruction decode, Execution, Memory access, and Writeback.

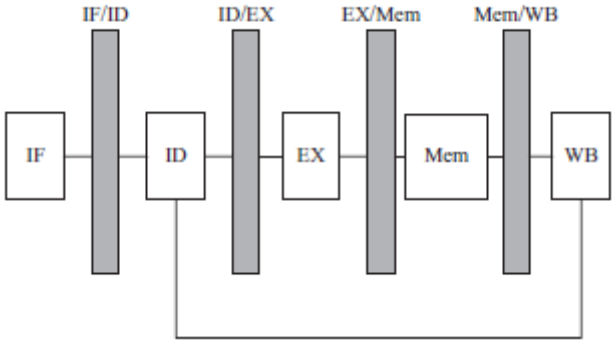

Figure 2.1: An abstract five-stage pipeline.[4]

In the instruction fetch stage, an instruction is fetched from the instruction cache (I-cache) at the address given by the Program counter (PC). In the case of non-branching instructions, the PC is incremented by one, and when finished, a new instruction is loaded. When a branching instruction is processed and the condition is true, the PC is moved to the address pointed by the instruction, and the pipeline is flushed.

The Instruction decode decodes the fetched instruction to know the type and operands of the instruction. In this step, the immediate constants are also extended to the required amount of bits.

The Execution stage performs the actions of the provided instruction. If the instruction is arithmetic, the stage will perform arithmetic or logical operations. If it's a load or store operation, the effective address is calculated for loading or storing a value. In the case of a branching instruction, a new value for PC is computed.

The Memory access stage does nothing if the instructions are not load nor store, nevertheless, all instructions must pass through this step. If the instruction is a load operation, data will be fetched from the Data cache (D-cache). If the instruction is the store operation, data is modified on the index got from the Execution stage.

Lastly, the Writeback stage stores calculated or fetched values in the result register. This applies to all floating-point and fixed-point instructions, logical instructions, and load instructions. In the case of a branch or store operation, nothing happens. [4] [10] [17]

Hazards

As already mentioned, the subscalar processor was restrictive, on the other hand, there couldn't arise any problems, because instructions were executed separately. By breaking this restriction in the scalar pipeline processor, we have to face new problems. These problems are called hazards. We differentiate between 3 known hazards.

The first one is *Stuctural hazard*. This hazard can be encountered when two or more stages compete for the same shared resources. Example being shared cache for instructions and data, where stages IF and MA would compete for access. This type of hazard can be easily prevented by separating the problematic component or by adding more hardware components.

Another hazard is *Data hazard*, which happens when two or more consecutive instructions have data dependencies between them, where one instruction is trying to access data

that are not ready yet. This type is again split into 3 categories: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR).

Read-after-write occurs, when instruction at address i has the same output register as input register of an instruction $i+1$. The instruction $i+1$ will read old data before instruction i has a chance to produce the new ones. The example of such instructions could be: *R3 = R1 + R2, R4 = R3 + R10*. The RAW can be removed either by stalling the pipeline or forwarding computed data. Stalling the pipeline stops any instructions which depend on the previous one. This approach is not optimal, because it leads to a longer execution time. Forwarding is done by adding new paths to stages where the data are already present. Of course, this leads to extending the output multiplexer and adding new busses between stages. In the example, the data from the first instruction are available at stage EX, so by adding path EX to EX, the instructions can execute right after the previous one finishes its EX stage. The same forwarding can be done from stage MA to EX if the first instruction were to be load. [12]

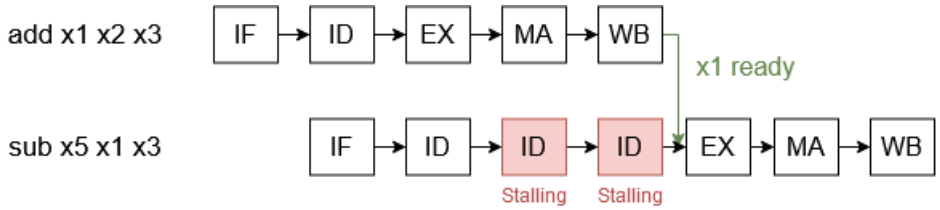

Figure 2.2: RAW dependency between two instructions

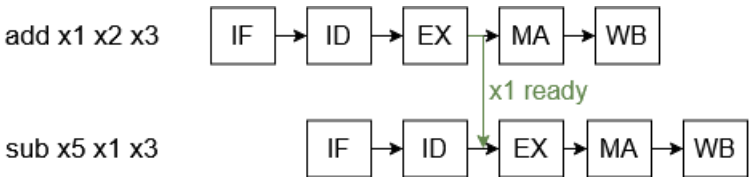

Figure 2.3: RAW dependency on the same instructions with forwarding

The other two hazards, write-after-read and write-after-write, do not happen inside the pipeline processor, but usually happens inside a superscalar processor, which will be mentioned later. Write-after-read is encountered, when the instruction at address i has the same input register as the output register of the instruction at address $i+1$. Write-after-write happens, when two consecutive instructions have the same output register. Both of these hazards are solved by renaming the registers or by stalling, where preferable is renaming.

The last of the three hazards is the *Control hazard*, which is connected to the flow of the program. These are triggered by unconditional and conditional branch instructions. Without any hardware support, the target address is calculated in the EX stage, and the MA stage writes it to PC. That would lead to a three-step delay before any other instruction can proceed. We can reduce the delay to 1 if we add a new adder in the ID stage, which would compute the address and send it to the PC in one step. Also, we can add a test for zero in this stage for the conditional branch instructions, which do jump if a source register is zero or not. For other branch instructions, we would have to wait until the EX stage is done, where the comparison will be calculated. Still, that is a 1-2 step delay compared to 3.

For now, the instruction that was input into the pipeline during the delay was the NOP instruction (i.e. no operation). What if the instruction wasn't just a filler instruction, but some useful and maybe correct instruction that should be executed after the branch instruction. This is where predictions come into play. We differentiate between two possible predictions: negative or positive. In the case of negative prediction, we are expecting that the jump won't happen, so we insert another instruction in the program's order. The positive prediction does the exact opposite. [4] [19] [17]

2.2 Superscalar processor

The time to process the program is defined by the equation $t = \frac{IC}{(IPC*f)}$, where IC is the instruction count, IPC is instruction per clock, and f is frequency. To make programs faster, either IC needs to be smaller, therefore the program needs to be optimized, or IPC or f has to be bigger, resulting in the overall faster processor. In the scalar processor, we had an ideal IPC of 1, but in practice, the Execution stage (EX) took longer, because of more complex computation, such as complex integer calculation being multiplication or division or floating-point arithmetic, so the IPC is in reality smaller. The Superscalar processors make the pipeline wider, meaning that it no longer processes instructions linearly, making the IPC value bigger. [19]

The Superscalar processors are divided into 2 parts: the front-end and the back-end. The front-end covers pipeline stages IF and ID, but now it can fetch multiple instructions at once. The number of fetched instructions corresponds to the number of ways the processor has, making it an m-way superscalar processor. The back-end of the pipeline covers EX, MA, and WB stages with the difference that now these stages process instructions concurrently. [4] [19]

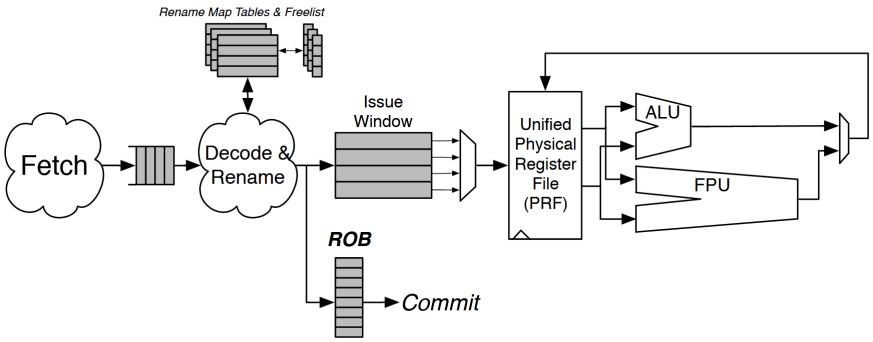

Figure 2.4: An abstract superscalar processor.[1]

The Superscalar processors are divided into two categories. The first one is a static, in-order superscalar processor, where the front-end releases instructions in the strict program order and data dependencies are resolved before passing them to the back-end. The second one is the out-of-order or dynamic superscalar processor, where instructions enter back-end before previous instructions in the program order. To make instructions leave the back-end correctly, the Writeback stage is replaced by the Commit stage, which makes instructions leave in the order given by the source code. Both of these versions have an in-order front-

[1]Source: https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf

end, meaning that the instructions are fetched and decoded in the program order. [4] The following section will be focusing on the separate parts of an out-of-order superscalar processor.

2.3 Superscalar's front-end

The focus of this section is to go through all front-end parts of the superscalar processor and explain them in order from fetching the instructions to the reservation stations, where instructions left for the back-end of the processor.

Instruction fetch

The Instruction fetch has to give m instructions every cycle. But there are several problems. The first one is determining the address from where to get the next instruction when there are several branching paths. The solution is to use ALU near the fetch stage, for address calculation, but the processor is still dependent on previous predictions. Even if we know precisely where to jump, the processor can load one cache line in one cycle, which creates an issue when instructions are way far apart. That limits the processor to one branch prediction per cycle. This limitation can be eliminated by using *trace caches*.

Trace cache allows the processor to fetch instructions with multiple branch predictions, called traces, in one cycle, assuming that the predictions will be correct. The traces are recorded in the order in which they were decoded and saved for later purposes. Trace cache entries have a tag and a data, where the tag is the address to the first instruction in the trace. If in case the processor finds itself at the beginning of a trace, the PC will index from the trace cache rather than from the I-cache. There are different types, such as concurrent trace caches, which access instruction cache and trace cache in parallel, or sequential trace cache, which access these caches sequentially. [4] [6]

Branch prediction

Branch predictions are done during stages of fetching and decoding. The prediction itself can be either static or dynamic. Static predictions are based on statistics from multiple program runs. From known statistics [19] [4], around 83% of jumps will be performed, so if the processor was set to static positive prediction, it would be correct 83% of the time. Unfortunately, the costs of misprediction grew over the years, so this approach is obsolete. [4]

The dynamic prediction is based on previous predictions that happened earlier in the execution of a program. The simplest solution to this is to use *bit predictors*. The bit predictors are of sizes either zero, one, or two. The three or more bit predictors do give only minimal improvement compared to the previous 3 and only give worse storage costs. The 0-bit predictor functions almost the same way as the static one, it is determined by the first branch instruction and for the next, it will use saved prediction from the first one. The 1-bit predictor changes prediction depending on the last branch instruction. The bit represents 2 states, either "taken", meaning that the previous branch condition was true and PC was changed to target address, or "not taken", where the branch condition was false. The 2-bit predictor is a bit more complicated. It uses 4 states, which can be seen in the figure 2.5. The "strongly taken" state is achieved after 2 consecutive branch taking. The "weakly taken" state is reached when we receive a false prediction in the "strongly

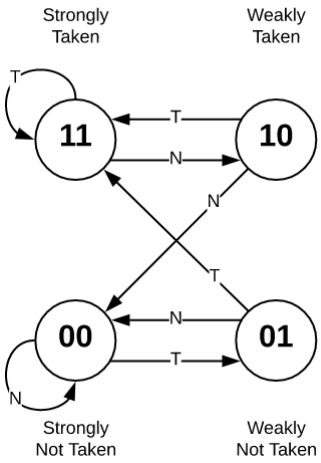

Figure 2.5: Two-bit predictor finite state graph [21]

taken" state, therefore the branch is not taken. The "strongly not taken" is the opposite to "strongly taken" and the "weakly not taken" is to the "weakly taken". [4] [21]

The processor keeps these predictions in a cache called a *pattern history table* or PHT. This cache can be indexed using k selected bits from a PC because using the whole PC would be way more costly than the payload. Unfortunately, this leads to address aliasing, where two different branch instructions can have the same tag, and therefore use the same predictor. We can get better results if the prediction would move from being local to being global, which would take into account all previously evaluated branch instructions. For that, we can use *global shift register* or GSR. The GSR remembers the history of k instructions by creating a bit vector, where 1 represents "branch taken", and 0 "branch not taken". This is then used to index predictions in the PHT. This works well if the k is big enough, meaning if there is a small number of entries in the PHT or the GHR vector is too small, this approach is not suitable and the local predictions should be used instead. This approach also suffers from not knowing the actual position in the program. There is an easy solution using a hash function, where the hash is calculated from GSR and part of the PC. The hashing function can be for example an XOR gate. This exact solution is named *GShare* predictor. [4] [21]

There are also other similar solutions, for example, McFarling predictor, which is a Gshare with a "meta-predictor", a two-bit counter, which selects from one or more PHT, that gives a prediction result based on a majority vote. Another variant is the Bi-Mode predictor, which again uses several PHTs with Gshare for indexing and a meta-predictor that is indexed by the PC telling, which PHT should be used. [1]

For a successful prediction, the processor needs to know the target address of a jump. Without it, the m-way processor would have to have m additional adders not to cause stalling during the instruction fetch stage. For that, the superscalar processors use *branch target buffer* or BTB, where the branch target addresses are kept. There is an option to either use BTB with integrated PHT, which would make the BTB much larger, decoupled BTB-PHT, where addresses and predictions are stored in separate caches but indexed the same way, or to use more PHT and use PC to address a specific one and use index saved in BTB to address specific row in chosen PHT, which would reintroduce locality to the prediction. [4]

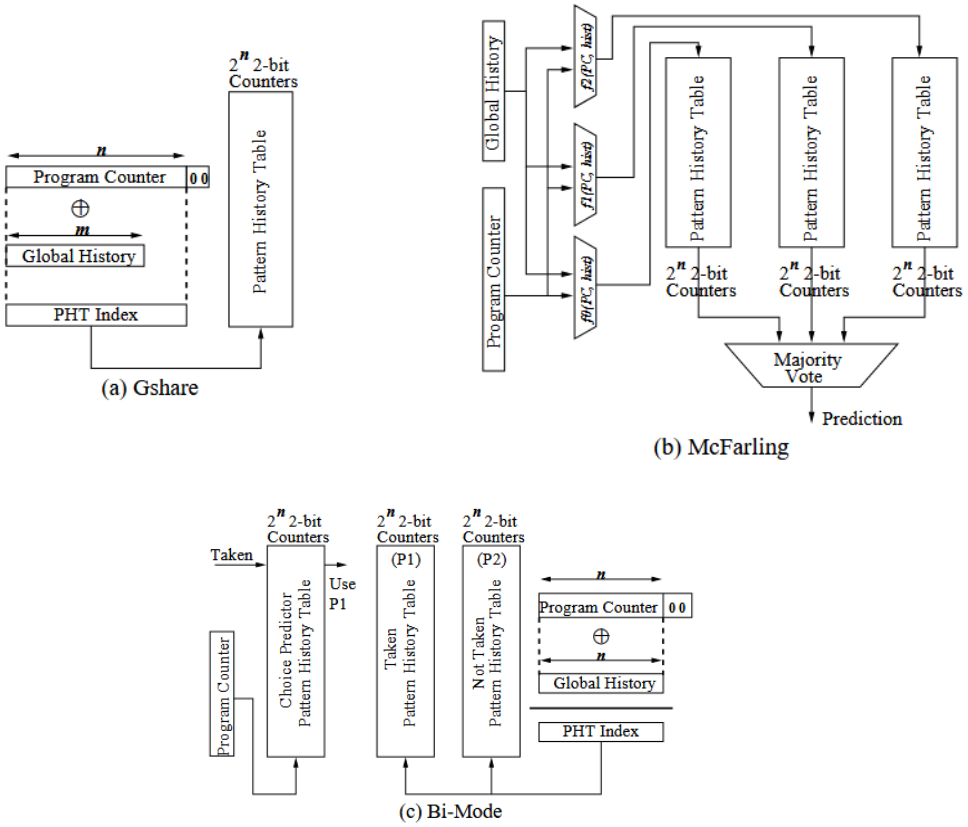

Figure 2.6: Schematics of the Gshare, the McFarling, and the Bi-Mode in respective order. [1]

This also introduces new instances of mispredictions. The first one happens when the direction is mispredicted, which is the most costly one, where the processor needs to nullify all instructions after the branch. The second one happens when the prediction is correct, but the address is missing from the BTB. In this case, the address needs to be calculated and fetching is halted during the calculations. The third case happens when the prediction is correct, the BTB has an address, but the address points to the wrong place in the program. This is called *misfetch* and can occur because of indirect jumps. The penalty is the same as if there would be no address. [4]

Instruction decode

The decode stage takes fetched instructions from the instruction fetch stage and decodes them and allocates all necessary resources for each instruction. This implies that if m instructions are fetched, the processor has to have m decoders not to stall the fetching. But decoders are quite hardware-expensive compared to fetch buffer. Some limitations can be introduced to save up HW-costs for example limit the number of branch instructions or use predecoded bits, that are appended to instruction during fetching so that instruction boundaries are performed only once and it will save work during decoding. The disadvantage is that the size of the I-cache is doubled to facilitate predecoded bits. [4] [21]

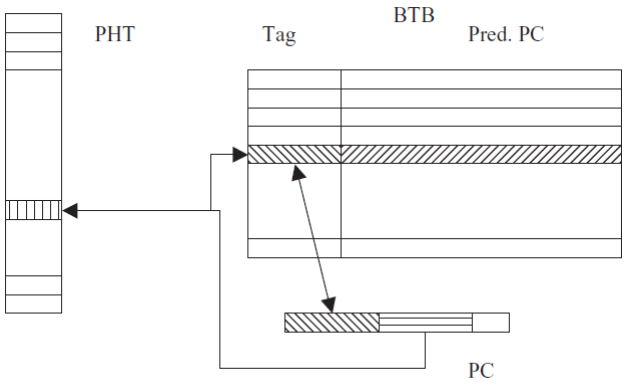

Figure 2.7: Example of indexing into decoupled PTH-BTB [4]

Register renaming

As said in the pipelining section, processors have to have some mechanism to overcome data hazards. The RAW hazard in the superscalar processor is overcome by processing other instructions before this dependency is resolved, which is easy because of the out-of-order execution. On the other hand, the WAW and WAR hazards will start to become a problem. These hazards can be eliminated either by using the Scoreboard algorithm or Tomasulo's algorithm. [4]

The Scoreboard algorithm is the simpler one, a cache called scoreboard keeps track of all issued instructions, their operands (registers), and their validity, represented by the *validity bit*. When the valid bit of destination register is 1, the register will be taken, set to 0 and the instruction will be processed. If the destination register is set to 0, it will wait until it becomes 1 again, which will eliminate the WAW hazard. After the execution, the processor will look into the scoreboard and find all source registers that are equal to the destination, and set them to 1. If it is already set to 1, the instruction has to wait until the validity bit is set to 0, which will eliminate the WAR hazard. When all validity bits are valid, instruction is ready to be executed, which eliminates RAW hazard. In short, the Scoreboard algorithm solves the hazard problem by stalling the execution of the instruction, which is not optimal. The more convenient solution would be to rename the registers so that the later instructions will be executed instead while waiting for the RAW to be resolved. [19]

Tomasulo's algorithm solves the problem of waiting by renaming registers, which eliminates WAW and WAR hazards. This is done by taking the destination register and renaming it to a different one, then changing the name of source registers that follows after and are equal to the original naming. [20] Those values are then kept either inside of the reorder buffer, which is called "implicit renaming", where register file entries are according to ISA specification and the register file is called *architectural* (ARF). This approach is costly in terms of reorder buffer size because it needs to hold all partial results of instructions. Another approach is to use an additional register file, called *Rename register file* (RRF), which would accommodate speculative registers. A different approach is to have *Physical register file* (PRF), which will accommodate more registers than specified by the ISA, having both architectural registers as well as the speculative ones. In both previous cases, we need an additional cache, called Rename Map Table, which would hold mappings of speculative

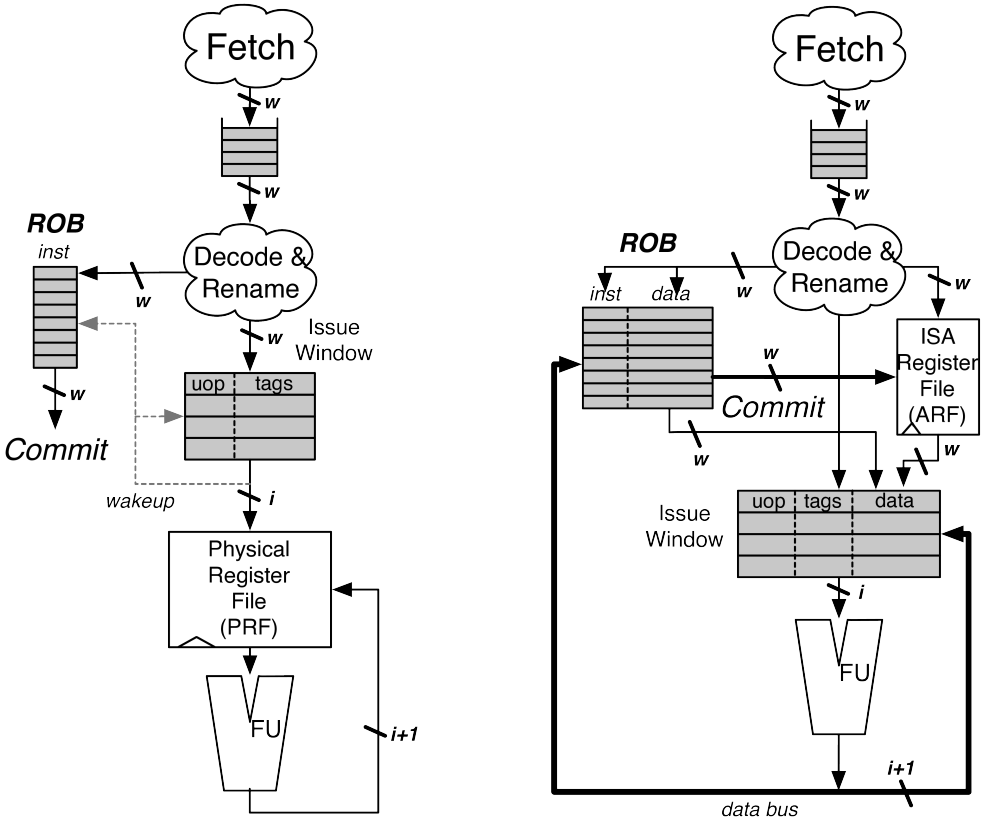

Figure 2.8: Example of an explicit (left) and implicit (right) renaming diagram. [21]

registers to the architectural ones. This can be part of the ARF, reorder buffer, or it can be a separate cache. This is called "explicit renaming". [19] [21]

The "explicit renaming" behaves according to the following algorithm. When the registers are at their initial state, architectural registers are set to *assigned* and the speculative ones are set to *free*. When the free register is used as a destination register, it will become *allocated*. When the result value is produced at the end of the execution stage it will become *executed*. After committing the register becomes *assigned*, which implies that the speculative register becomes architectural. Releasing will happen after no instruction will reference this speculative register and it can become *free* again and wait for the new assignment. [4]

Reservation stations

After registers in the instruction have been renamed and placed into the reorder buffer, the decoded and renamed instruction is dispatched to *reservation station*. The reservation station is holding all dispatched instructions, maintaining information about the type of the instruction, source operands and their valid bits, the name of the result register, and the entry in the reorder buffer, where the result is stored. The reservation station's purpose is to detect the readiness and schedule instructions. [4] [19]

Scheduling is often done by issuing either the oldest instruction in the station or the instruction that would stall most instructions. Detecting the readiness is associated with valid bits of source registers, so the station knows if the result can be computed. The

processor eliminates wasteful stalling by enabling other instructions to pass by monitoring their validity bits. [4] [2]

When the destination architectural register is mapped to a speculative one, the valid bit is flipped to 0. When the instruction is dispatched, every register with validity bit in 1 is replaced by the value inside the register. If every source register validity bit is set to 1, the instruction is ready to be issued. If the valid bit of any register is during dispatch in 0, the name is passed and will wait until the register value is delivered. When an instruction completes, the result and the name of its destination register are broadcasted to all reservation stations. If any reservation station has an instruction, which has the same register name as the broadcasted one, the register tag is replaced by the value inside the register and set validity bit to 1. [4] [19]

2.4 The back-end and commit stage

This section is focused on the back-end of the superscalar processor and explains the execution of memory access instructions, and also introduces the reorder buffer (ROB).

Reorder buffer

The Reorder buffer, or ROB, gives an illusion that the instructions are executed in order. It stores information about dispatched, issued, and already done instructions waiting to be committed. The ROB operates as a circular queue where the order is given by the executing program. The information about instructions is stored in the tuple where the main parts of the tuple are: is instruction busy (busy bit), is instruction valid (valid bit), speculative bit, and a rename state, where the remapped speculative and the original architectural register is stored. Instructions that are not speculative, not busy, and valid can be committed. In case of speculative instructions, ROB needs to wait until the branch is evaluated and after either speculative bit is changed to non-speculative in case of correct prediction, or valid bit is changed to false in case of incorrect one. [21] [19] [9]

Memory access instructions

As already mentioned in the Pipelining section, store and load need two stages to finish, address computation and memory access. This can lead to RAW hazards. Apart from hazard created between load and arithmetic instructions or arithmetic instructions and a store, there can be an instance, where a store is followed by a load. In the following paragraphs, a solution which modern processors implement is explained.

For evaluating a store instruction, the processor needs to have an address where to save data, and the data itself. For that, the processor needs the *store buffer*, which stores information about the state of the store instructions. It is organized into a circular queue, as the reorder buffer. To track the state of the store instruction, for each entry, there is a set of flags that indicated one of these states: entry is available, the entry has an address without result, the entry has an address and a result and is waiting to be committed, and entry has been committed.

For load instruction, the processor has a similar buffer called *load buffer*, where the address and the value are kept until the entry can be committed. The problem arises if a store is to be executed before a load. The processor can check whether a certain load has the same address as the store thanks to the load-store buffers. If the address does

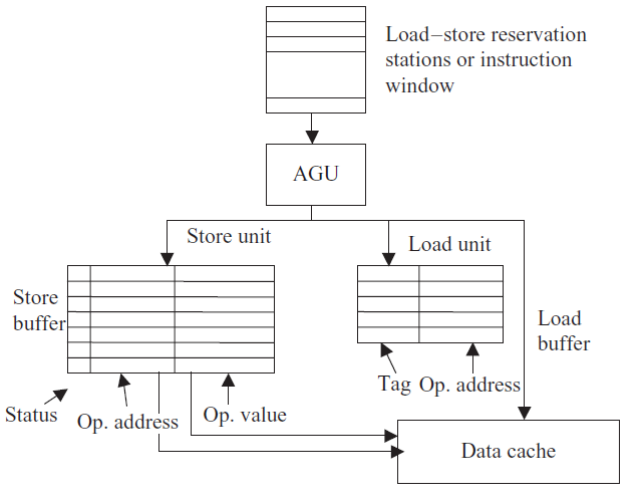

Figure 2.9: Load/Store buffer diagram. [4]

not match, the load instruction can pass the store, which is called *load bypassing*. If the opposite is true, the load cannot access a memory until the store is committed. In case that the load is after store and the store already has all values ready and is either waiting or is already committed, then the load instruction doesn't have to access memory and can take the value from the store buffer. This approach is called *load forwarding*. In the case of the same addresses but no result, the load entry will wait for the store instruction to get the data and after that, depending on the load, the instruction can be completed.

The processor executes out-of-order, there can be an instance, where the load gets issued before the store, even though the store should be executed first in program order. This creates a problem since after processing a store, the processor has to nullify the result of the load instruction if there is a dependency. So the processor can only process load speculatively and only if load instruction is on the front of the load buffer and the ROB, then the instruction is not speculative anymore. But how to approach scheduling load instructions, when this problem is known? There are three approaches. The first one is a pessimistic approach. The processor waits until it is known that there is no RAW hazard, so the instruction can be safely processed. Another approach is optimistic, where the load instruction is processed immediately, but also has a recovery mechanism in case of data dependency. The last approach uses prediction to estimate, whether to evaluate load or not. That allows programs with safe loads to be processed optimistically, meanwhile programs with strong dependencies to be more pessimistic. [19] [4] [21]

Chapter 3

Overview of existing simulators

As this thesis focuses on the implementation of the superscalar processor simulator, it would be wise to review already existing simulators and find out the features they provide, they have in common, their limitations. The new simulator should follow the best design practice from all of them and implement some, or even all, of the missing features.

3.1 VSIM

VSIM simulator is currently used in the Computation Systems Architectures course[1] for showing the basics of superscalar processors. It was developed in 2001 and offers five architectures of superscalar processors of that era, namely Compaq Alpha 21264, Hewlett-Packard PA-8500, IBM Power3, Intel Pentium Pro/II/III, and MIPS R10000. [15]

The main window shows function blocks of different architectures, depending on the selection of the processor. The function blocks cover the decode and dispatch unit, reservation station, floating-point unit, reorder buffer, register files, and so on. The flow of the simulation is shown by arrows between blocks.

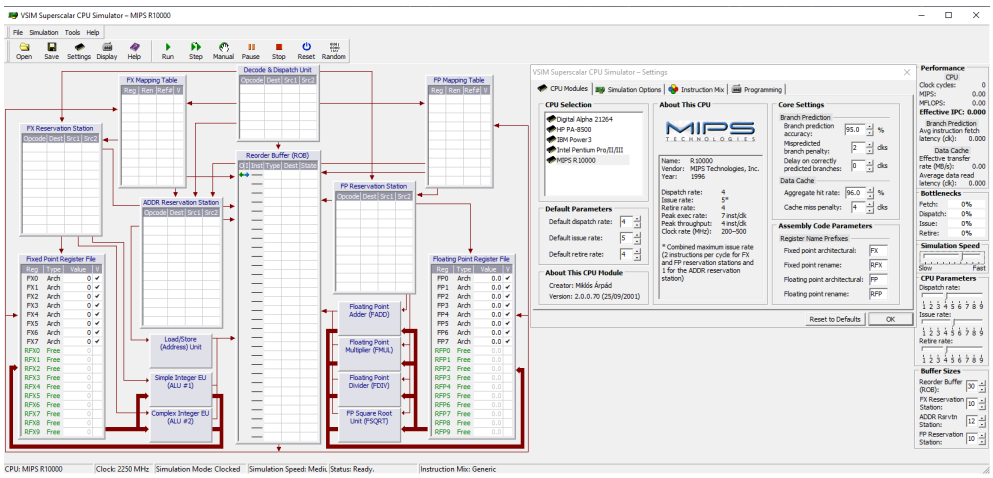

Figure 3.1: The main windows of VSIM with a setting dialogue window.

The simulator offers 3 options for the instruction stream. First, it can create a random stream of instructions for a simple demonstration of the instruction flow between blocks.

[1] https://www.fit.vut.cz/study/course/13577/.en

Second, it can run a single instruction to show more detailed processing of that instruction, or finally, it can run a user-defined assembly program. When running, this stream will be executed sequentially in architectural blocks. The simulation can be run in 2 modes. The first one is the educational mode, which will show step by step how the instructions are decoded, dispatched, issued, executed, and retired. In each step, explanatory tooltips with what is happening in the current step are also shown. The second mode is the performance mode, in which the simulation runs continuously and gathers performance metrics about the selected architecture with a set of given parameters. In this mode, simulation speed can be set.

The simulated processor is customizable by prepared parameters and the user can set up, e.g., how many instructions are dispatched into the processor, how many instructions are issued into the function units, or how many instructions are retired from the reorder buffer. The user can also change the size of the decode and dispatch unit, reservation stations, or the reorder buffer. Since the data cache and branch predictions are only simulated, there are also changeable probabilities of branch prediction accuracy or data hit rate. The instructions have modifiable delays and frequencies, in which they appear in the auto-generated instruction stream.

Unfortunately, there is no way of adding new processor models to this simulator. Even the ISA of each processor cannot be extended. Another thing, the simulator does not have, is a deterministic branch prediction. The simulator simulates jump predictions based on the probability value, which is snot how real processors behave. The same can be said about memory access.

3.2 QtMips

QtMips is an interactive simulator developed at FEL ČVUT as a diploma thesis. It shows the inner workings of a single MIPS pipeline on a user-defined code. The simulator offers detailed views into separate caches and the processor can also be modified. [14]

QtMips has 4 built-in and one user-defined presets of the MIPS processor. The presets are: Without the pipeline, without pipeline with instruction and data cache, pipeline model without hazard unit and cache, and a pipeline model with hazard unit and cache. All of these parameters can be set in a user-defined processor, plus users can set sizes of instruction and data caches, their associativity, and replacement policy. In case of hazards, there are 2 choices, either the pipeline will be stalled or will try to forward the result of the previous instruction.

The main window offers many tabs with a detailed view into current values of registers, memory, and compiled instructions with the raw value of each instruction. The app offers a terminal window for string outputs and RGB peripherals for sending RGB data to a simple frame buffer.

The simulation window shows the configured MIPS processor, divided into two parts if the "no pipeline" version is selected, or into five parts, in the case of the pipeline version. When instructions are being processed, the values can be seen in processor registers in separate stages. If caches are allowed, instruction and data cache will be shown in the simulation and the user can open cache details where the configured n-way cache with cached data can be seen.

The simulation code is a subset of MIPS instruction set architecture, with added pragmas for displaying certain windows for demonstration purposes. The instruction subset also specifies which addresses are used for reading and writing peripheral data. The simulator

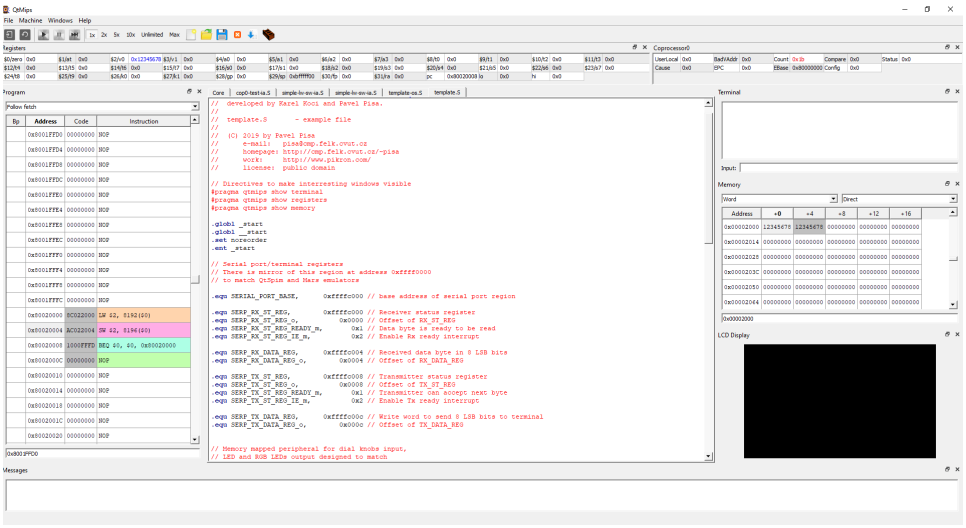

Figure 3.2: QtMips main window.

offers a template file with constants addresses of peripheral data. The code itself can be edited inside the application IDE with a simple keyword highlighting and the support for saving and loading the code.

The simulator only works as an in-order processor. However, most of current processors process the instructions out-of-order, so this simulator can only serve as a basic example. And same as the previous simulator, it lacks demonstration of branch predictors or load-store units of modern processors.

3.3 OpenDLX

OpenDLX is a simple pipeline simulator which can be used to explain the basics of pipelining. It was developed at the University of Augsburg in 2013. The simulator is capable of hazards handling and jump prediction. [18]

The simulator processed each instruction in basic pipeline stages (Fetch, Decode, Execute, Memory access, Writeback), which are displayed in the "Cycles and pipeline" window. Each row represents an instruction and each column one cycle of the simulation. Detailed messages about separate cycles can be seen in the Log window where the information about what happened in each step in one simulation cycle can be found. The statistics are being gathered and displayed in another window, where the information about the number of correctly predicted jumps, hits or misses in the branch target buffer, how many instructions were forwarded in case of a hazard can be found. There are also windows for the content of the register file and memory. The branch predictor can be configured to either static with a taken or not taken state, dynamic 1-bit predictor, or 2-bit predictor, where initial bits can be configured.

The ISA is a subset of MIPS instruction set architecture. The code can be edited inside the application using a basic text editor. The simulator package comes with sample programs with basic constructs in MIPS ISA.

This simulator offers only a simple insight into the pipeline processing. It does not show a detailed view of the processor like the previous two. The simulator has a deterministic

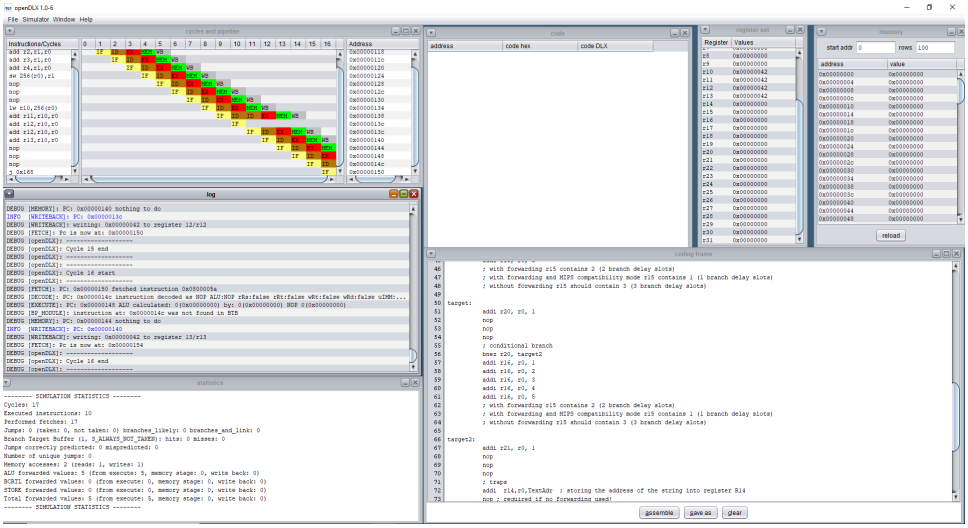

Figure 3.3: OpenDLX simulation window.

branch predictor, but it lacks any visualization of this unit apart from deriving it from executed instructions in the pipeline window.

3.4 Ripes

Ripes is a graphical processor simulator built around the RISC-V instruction set architecture. It offers similar functionality as QtMips with a few enhancements. [16]

The application is split into 3 tabs: Processor, Memory, and Editor. The processor tab shows the simulation window with the RISC-V pipeline, values inside the register file and instruction memory, console for program outputs, and basic statistics about processor performance. The processor can be changed to a single cycle processor, default five-stage pipeline, pipeline processor with hazard detection without forwarding, and pipeline processor with hazard detection and forwarding. The layout of the processor can also be either simplified for only showing the necessary components, or can show an extended version with control, hazard, and forwarding unit. The processor can only simulate RISC-V I (integer operations) and M (multiplications and divisions operations) extensions. Apart from the processor view, users can access a dialogue with a pipeline stage table where stalls between stages can be seen as well as stages already completed for each instruction.

The memory tab displays values stored inside the memory. It also shows data and instruction caches. Caches can be customized by changing the number of lines, ways, and blocks, the cache have. The replication policy can be configured too, as well as write hit and miss policies. During and after the simulation, the statistics about cache performance and a graph of the cache hit ratios can be seen.

The code tab offers a basic IDE for editing assembly code in RISC-V ISA with keyword highlighting. The input code can also be written and compiled in C, but for that one needs official SiFive Freedom RISC-V tools with a C compiler available on SiFives GitHub[2]. When running, the simulation will display compiled code with highlighted instructions and their

[2]https://github.com/sifive/freedom-tools

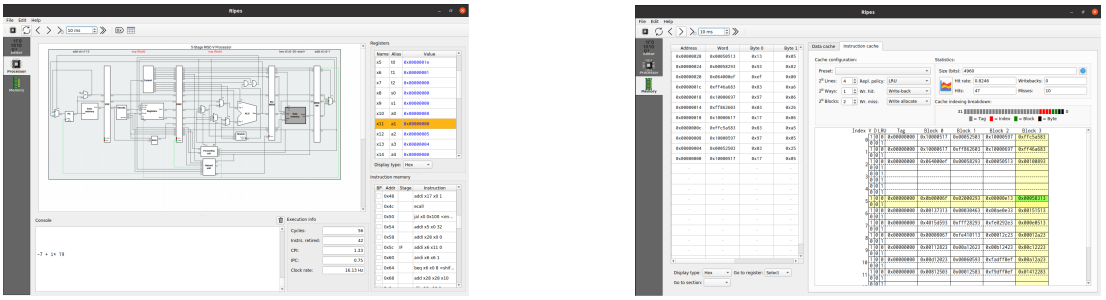

Figure 3.4: Processor and Memory windows of Ripes simulator

current pipeline stage in the right window, where the highlighted instructions are currently being simulated.

As QtMips, it has the same lack of features, being only in-order processing, static branch predictions, and simple load/store implementation, which is enough for scalar pipeline processors, but not for modern ones.

3.5 Jupiter

Jupiter is a runtime RISC-V simulator. It simulates the IMF extensions of the RISC-V instruction set architecture. This simulator focuses on the programming side of the things, compared to other simulators. [5]

The simulation window shows compiled instructions, register files (integer and float), memory values, and data cache. The data cache is configurable, offering a setting of the size of the cache block, number of blocks, and associativity. One can also change whether the cache is fully associative, directly mapped, or n-way associative. Registers in the register file are tagged by their standardized names, which is called the RISC-V application binary interface. The simulation itself can be either run as a program where you can see states of the program when being interrupted or at the end, or you can step over the instructions and see direct changes in registers, memory, or cache. The user can either step forward or even backward in the instruction flow. The inputs and outputs are introduced through the console inside the simulation window. The simulator offers a simple code editor of the RISC-V assembler with save and load functionality.

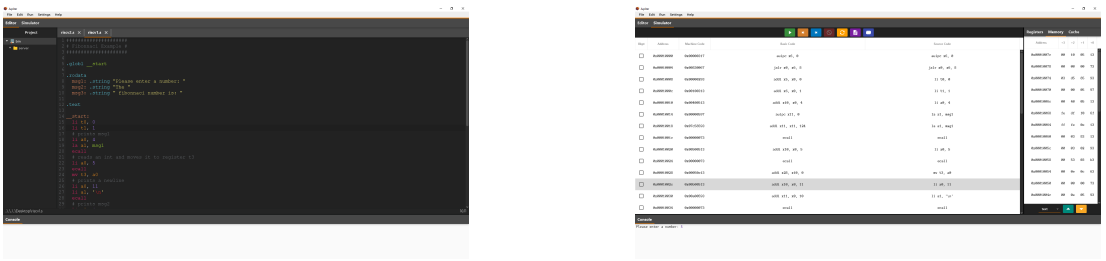

Figure 3.5: Code editor and simulation window of Jupiter

The simulator can only interpret instructions with some simple views into register files and memory. This can be used in courses which study assembly language and its use. However, for our purposes, it is lacking any kind of visualization for the processor's instruction flow, either pipeline processing, like in QtMips, or out-of-order processing, like in VSIM.

3.6 RISC Simulator by Peter Higginson

This web-based simulator[3] was developed in 2016 and is trying to visualize how a RISC-based processor inspired by ARM Cortex-M3 design works. [11]

The simulator page shows a basic processor block layout with one register file, arithmetic unit, PC counter, control unit for decoding and dispatching, and a memory block. During the simulation, a data bubble displays the data flow in the processor. Users can choose the number format for data from signed, unsigned, or hex format.

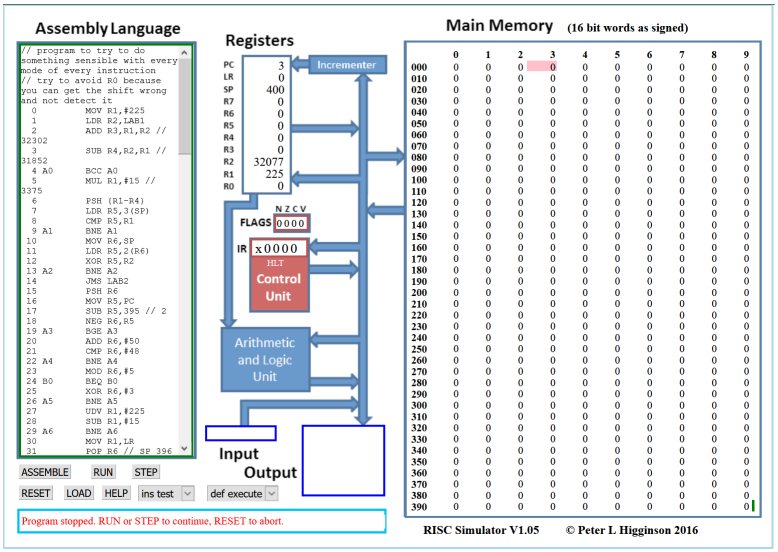

Figure 3.6: RISC Simulator by Peter Higginson.

The assembly language is a subset of the instruction set used by ARM with a few extensions. One of them being a special multiplication, which works similarly as in Intel or AMD processors where two x-length variables will create a new double x-length result. The same extension is made for divide instruction, which again works the same way as in Intel or AMD processors. The code can be inputted and edited through a text box at the website.

This takes a unique way of processing visualization, nevertheless, it shows only the basic blocks. The code has a small instruction set and offers only integer instructions. The simulator cannot be configured in any way.

3.7 BRICS-V simulator

The BRISC-V simulator is a runtime web-based simulator[4] developed at Boston university simulating RISC-V assembly including all extensions from RISC-V ISA. [13]

The simulator is composed of multiple panes, the code pane, register file pane, memory pane, instruction breakdown pane, and console pane. The register file pane displays separate registers and their values. The registers are labelled by their standardized names. The memory pane shows memory values either in binary, hexadecimal, or decimal format. The instruction pane is a unique feature. In this pane, you can see the breakdown of instruction

[3] https://peterhigginson.co.uk/RISC/
[4] https://ascslab.org/research/briscv/simulator/simulator.html#

into separate parts defined by ISA with values of a currently executed instruction. The console pane operates as both the input and output of an assembly program.

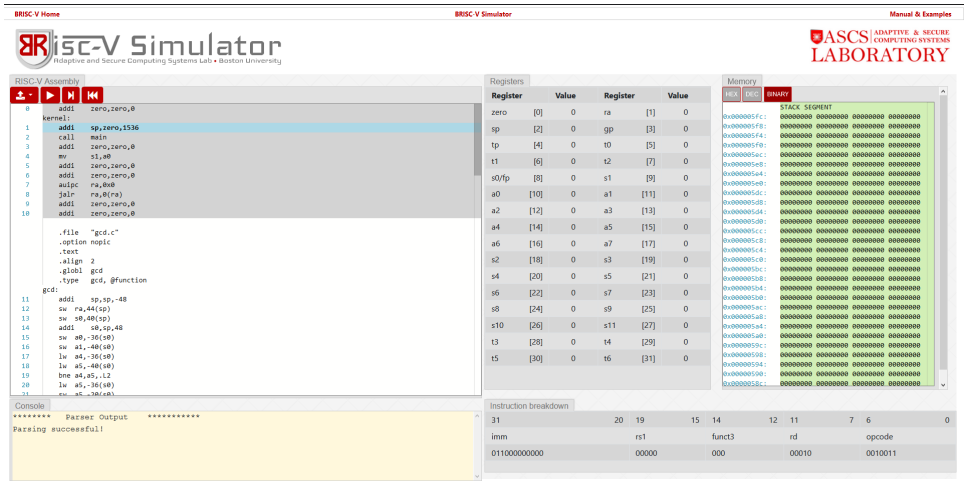

Figure 3.7: Boston university RISC-V simulator.

As mention before, the simulator interprets the RISC-V assembly code with all extensions including compressed format instructions, single, double, and quad precision instructions, and 32, 64, and 128 integer instructions. The editor pane supports saving, loading, and also includes example programs for demonstrations.

The advantage of this simulator is that it implements the whole RISC-V ISA and shows raw values for the instructions, which is unique compared to other simulators. The disadvantage is that it does not show the inner workings of a RISC-V processor, only the register file and memory values are shown. The processor cannot be configured.

3.8 Summary

This chapter reviewed some of the existing simulators that can be found and used for educational purposes. These simulators can be classified into interactive interpreters with memory and register details, or into interactive configurable processors, where users set parameters of their processor and see instruction phases in action. Examples of the first category are OpenDLX, Jupiter, and BRISC-V simulators. Examples of the second category are VSIM, QtMips, Ripes, and RISC Simulator by Peter Higginson. I have concluded that the further development should focus on interactive configurable simulators because they can show the inner workings of a processor where each dependency and function block can be demonstrated for the students.

Chapter 4

Proposal of implementing system

According to the findings in the previous chapter, I would like to propose my version of the superscalar processor simulator. This version should implement most of the missing features identified in the previous chapter. The look should be inspired by the VSIM simulator with some added features such as an internal text editor for coding, load/store unit simulation, and branch simulation. The simulator will be built around the RISC-V ISA [3], moreover, the user should be able to add new arithmetic instructions. The simulator should have a configurable number of ways in fetch, the delays for each function unit, branch predictor, and others. The register file and the program instructions should be visible in the simulation window, as many other simulators had similar features. The simulator should be implemented using the object oriented programming paradigm and and be able to run on various operating systems. That's why, Java programming language was chosen as the implementation language.

4.1 Decomposition of the implementing system

The implementation of the simulator shall be divided into 4 layers: Loader, Code, Blocks, and UI. Each layer should depend on the lower layer to be fully functional. Also, the implementation should go from the lowest layer to the top. The layered model can be seen in the figure 4.1.

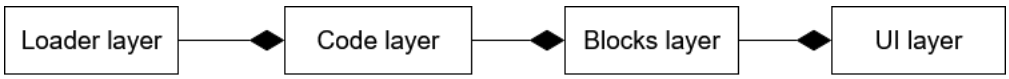

Figure 4.1: Abstract layer diagram.

The Loader layer shall consist of loading the ISA instructions and architectural register files. These should be stored in a suitable serializable format. The ISA should be organized as a folder containing all allowed instructions. Each instruction must contain the information about its syntax and how it should be interpreted. The register file must contain information about the data type of the whole file, and the registers names.

The Code layer shall implement the parser and interpreter logic. Since the parser needs to know how to parse each instruction, it needs the information from the loader. The same applies to the interpreter where the interpreter needs to know how to interpret each instruction. The interpreter should be divided into 3 minor interpreters for arithmetic instructions, load and store instructions, and the last branch instructions.

The Block layer shall implement the logic for each "block" of the superscalar processor, namely ROB, BTB, FUs, reservation stations, etc. All of these blocks must be connected to a central timer class simulating clock ticks. Each FU should have a configurable delay to simulate processing of instructions with different time complexity such as addition, multiplication and division.

The UI layer shall implement graphical views for the application. Each block from the Block layer must have a UI wrapper displaying important values to the user. This layer shall also implement interactions with the simulator, dialogue windows for configuring the processor, and also implement transitions between the text editor and the simulation window.

4.2 Simulator mockup

The processor model can be seen in Figure 4.2. The model was inspired by the model of RISC-V BOOM[1], which is an "explicit renaming" model with the physical register file. The view of the block should have the same ordering as the implemented data structure in the block. If there is no specific ordering in the data structure, the updated instructions should be displayed at the top in the view's list. The FU must display the instruction that is being processed. The load-store FU should detail load/store buffers. The main purpose of this simulator is to show interactions between blocks and reactions on different programs, so the main memory shall be simulated with delays on the load-store unit.

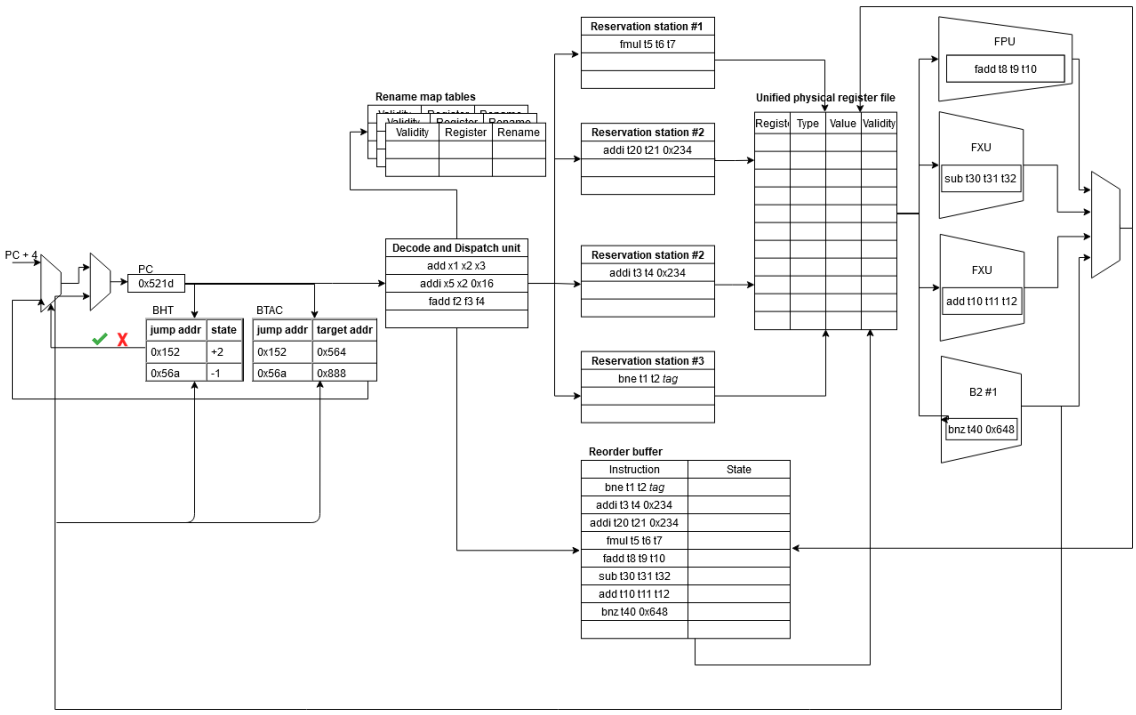

Figure 4.2: The mockup of the processor simulator.

The mockup's main window is featuring a control panel for controlling the execution, detail of architectural register files, and a left sidebar with the currently executed program.

[1] https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf

The left sidebar should show which instructions are currently being fetched into the processor in a certain step. The values in registers should be either hexadecimal, signed decimal, or unsigned decimal. To move between a code editor and a simulation window, there are tabs to switch between these two windows.

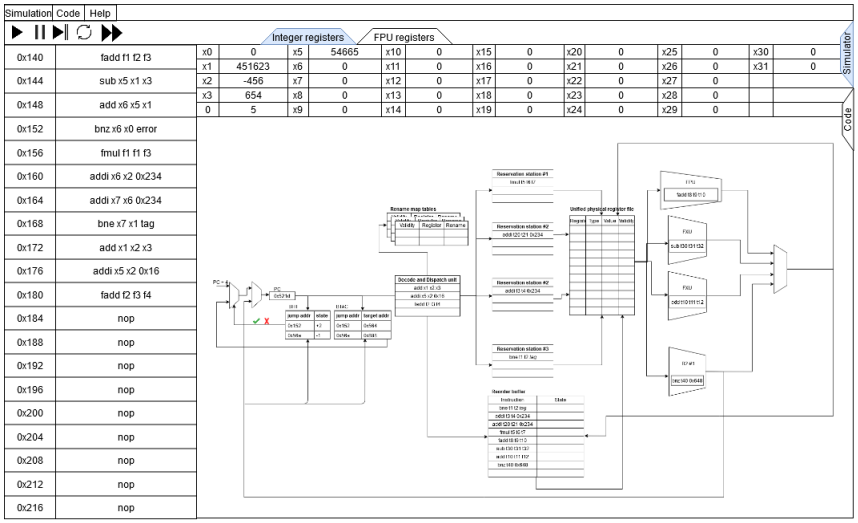

Figure 4.3: The mockup of the main simulation window.

The code editor shall have a different control panel from the simulation window to open, load, save, and compile user program. Compilation results shall be seen in the console window below the code. The console should show either success on a successful compilation, or point out rows with errors with the type of the error. After successful compilation, there should be a dialogue serving as a shortcut for switching to the simulation window.

Figure 4.4: The mockup of the code editor window.

Chapter 5

Implementation of the proposed system

Based on the proposal and findings from previous chapters, I implemented the system in the Java language. This chapter goes through the important features of the interactive simulator in a bottom-up manner starting from the Loader layer explaining how initial register and instruction load works, then moving to the Code layer to explained how the source code gets parsed and interpreted, after that going up to the Blocks layer to explain the algorithms used inside the simulator, and finally, moving to the UI layer to address simulation state is visualization.

During the development, the Gitlab from the SC@FIT research group was used to commit and version the application source code, where each feature were tracked using the Gitlab issues. Each issue have its own description, branch name and most of them have a class diagram, showing the relations between classed, that were implemented in the specified branch. Each issue also had a checklist, where the progress was tracked.

5.1 Loader layer

The Loader layer is composed of 3 classes, the main loader singleton class (`InitLoader`) and its sub-loaders for instructions and registers. Most of the higher-up classes have their own reference to this class, either because they need to know available instructions, or the structure of the register files. For that reason, the loader class needs to be the first thing that is created and called even before the simulation window is rendered. Both register files and instructions are saved in a JSON file, not just because of its human readability but also because it is meant to be modified and extended by the users.

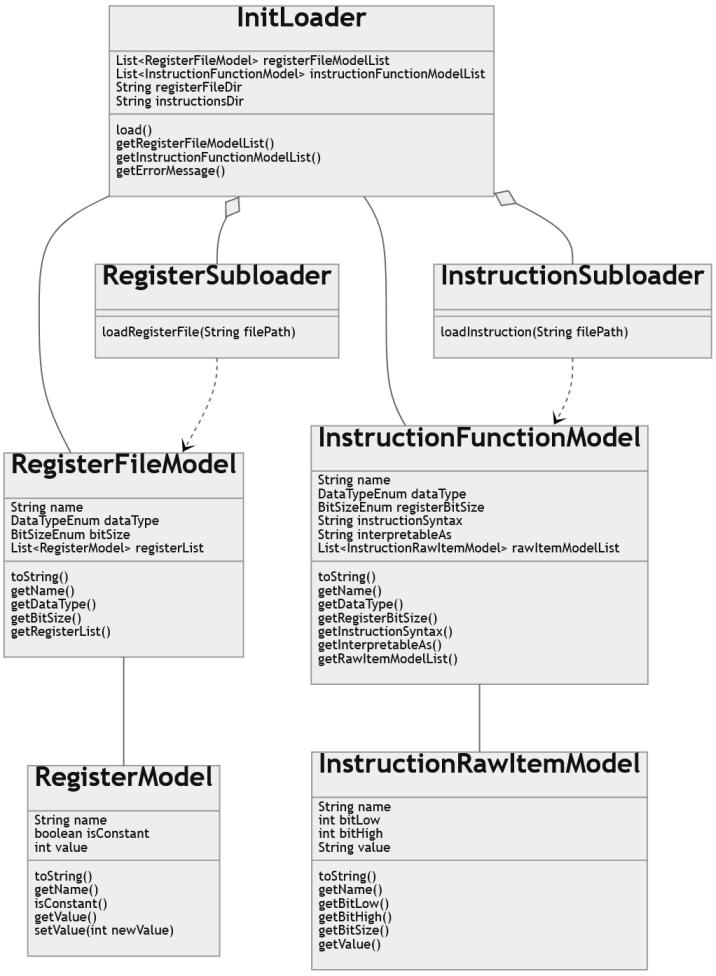

Figure 5.1: Loader class diagram

Register file structure

A register file is a JSON object composed of these subobjects:

- `name` - displayed name of the register file in the UI layer
- `dataType` - the data type of each register
- `registerList` - the list of all available registers in the register file

The allowed data types in the `dataType` field are: the integer (`kInt`), the long integer (`kLong`), the float (`kFLoat`), and the double (`kDouble`). The register file with larger data type can also fit values of a smaller data type. For example, the integer date type can only fit 32-bit integers inside the registers, but the long data type can fit both 64-bit and 32-bit integer values.

The register list entries are also JSON objects holding information about its name (`name`), value (`value`) and if it is constant (`isConstant`). The `name` is again the dis-

play name of the register, the *value* is a initial number value inside the register, and the `isConstant` marks, whether the register value is constant or a variable.

As a example, let's take the integer register register from the RISC-V. Depending on the ISA extension, the file contains either 32-bit (`kInt`) or 64-bit (`kLong`) wide registers. It contains 32 different registers (and a PC counter register, which is represented differently in the simulator and therefore should be omitted), with the first one being constant 0. Such register file can be seen in the Figure 5.2. Each new register file is stored in a separate file for a loader to load at the start of the program.


```
{
  "name": "Integer physical register",
  "dataType": "kInt",
  "registerList": [
    {"name": "x0", "isConstant": true,"value": 0},
    {"name": "x1", "isConstant": false,"value": 25},
    {"name": "x2", "isConstant": false,"value": 0},
    {"name": "x3", "isConstant": false,"value": 0},
    {"name": "x4", "isConstant": false,"value": 0},
    {"name": "x5", "isConstant": false,"value": 0},
    {"name": "x6", "isConstant": false,"value": 0},
    {"name": "x7", "isConstant": false,"value": 0},
    {"name": "x8", "isConstant": false,"value": 0},
    {"name": "x9", "isConstant": false,"value": 0},
    {"name": "x10", "isConstant": false,"value": 0},
    {"name": "x11", "isConstant": false,"value": 0},
    {"name": "x12", "isConstant": false,"value": 0},
    {"name": "x13", "isConstant": false,"value": 0},
    {"name": "x14", "isConstant": false,"value": 0},
    {"name": "x15", "isConstant": false,"value": 0},
    {"name": "x16", "isConstant": false,"value": 0},
    {"name": "x17", "isConstant": false,"value": 0},
    {"name": "x18", "isConstant": false,"value": 0},
    {"name": "x19", "isConstant": false,"value": 0},
    {"name": "x20", "isConstant": false,"value": 0},
    {"name": "x21", "isConstant": false,"value": 0},
    {"name": "x22", "isConstant": false,"value": 0},
    {"name": "x23", "isConstant": false,"value": 0},
    {"name": "x24", "isConstant": false,"value": 0},
    {"name": "x25", "isConstant": false,"value": 0},
    {"name": "x26", "isConstant": false,"value": 0},
    {"name": "x27", "isConstant": false,"value": 0},
    {"name": "x28", "isConstant": false,"value": 0},
    {"name": "x29", "isConstant": false,"value": 0},
    {"name": "x30", "isConstant": false,"value": 0},
    {"name": "x31", "isConstant": false,"value": 0}
  ]
}
```


Figure 5.2: Example of a JSON register file object

Instruction structure

An instruction is a JSON object containing these subobjects:

- `name` - a unique identifier for one instruction
- `inputDataType` - data type of source values
- `outputDataType` - data type of the destination register
- `instructionType` - type of the instruction (arithmetic, branch, etc.)
- `instructionSyntax` - syntax of the instruction
- `interpretableAs` - how the instruction should be interpreted

The unique identifier for the instruction is its `name` since in most ASM languages there are no 2 instructions sharing the same name. Therefore, the `name` can be used for searching through all available instructions and then unambiguously verified and interpreted.

```
{
  "name": "add",
  "instructionType": "kArithmetic",
  "inputDataType": "kInt",
  "outputDataType": "kInt",
  "instructionSyntax": "add rd rs1 rs2",
  "interpretableAs": "rd=rs1+rs2;"
}
```

Figure 5.3: Example of a JSON instruction object

The syntax of the instruction in the `instructionSyntax` field is used during parsing with the combination of the output and input data types to verify and notify the user in the case an invalid parameter is provided. The explanation of the syntax can be found in chapter 5.2 in Code parser section. The `interpretableAs` value is then used when the instruction is evaluated inside the function unit. The `interpretableAs` has different syntax based on the value inside the `instructionType`, where the types are:

- Arithmetic type (`kArithmetic`) - Syntax explained in section 5.2 in Arithmetic interpreter subsection
- Branch type (`kBranch`) - Syntax explained in section 5.2 in Branch interpreter subsection
- Load/Store type (`kLoadstore`) - Syntax explained in section 5.2 in Load/Store interpreter subsection

A simple example can be shown on the `add` instruction. This is a simple addition instruction where 2 register values are added together and the result is stored in the destination register. In RISC-V ISA, `add` is a 32-bit integer instruction where both source and destination registers have the same value. The JSON representation of such instruction can be seen in the figure 5.3.

5.2 Code layer

The Code layer can be divided into 2 parts: the parser, and interpreters. The parser is used to parse the user's source codes and transform them into an internal representation, which is held by the parser on the successful compilation. Interpreters can be further divided into arithmetic, load and store, and branch interpreters, where each of them has their own implementation on how to process the source code line.

Code parser

The task of the code parser is to take an input code, break it down onto separate code lines, try to parse it, and in turn validate each line using the loaded list of instructions. Every code line is first matched with some instruction in the loaded instruction list, and if matched, a `InputCodeModel` object storing the information about the instruction name,

type, data type, code line itself, and unmatched arguments is created. After that, the new object is validated with the matched instruction, and its arguments are matched with abbreviations stored in the instruction syntax.

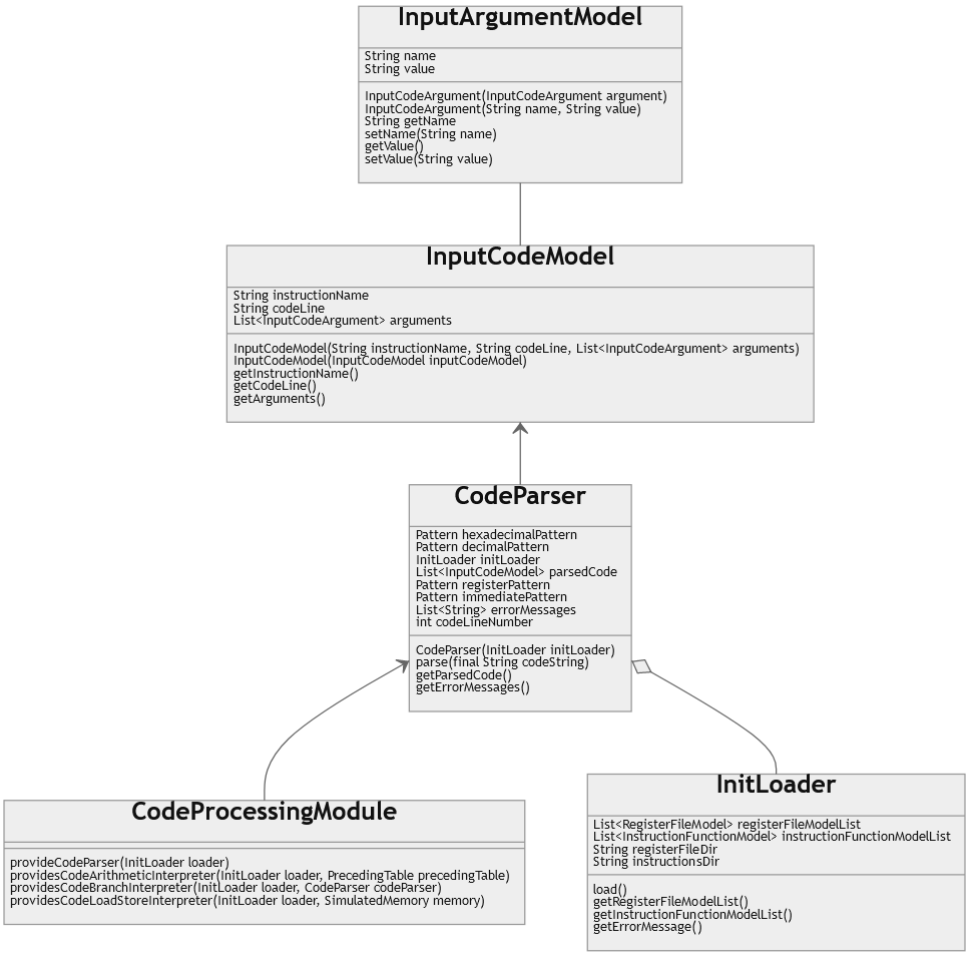

Figure 5.4: Class diagram for the Code parser

The syntax of the instruction is a string value following a simple pattern of name, argument1, argument2, argument3, etc. split by a whitespace. The argument is either the destination register marked by the `rd` abbreviation in the string's syntax, source register marked as `rsX`, where `X` is a natural number starting from 1, or an immediate value marked by the `immX` abbreviation, where the `X` is optional and can be either empty or a natural number starting from 1. The parser uses these abbreviations to validate arguments on each position. The destination or source registers can only be matched with registers in the loaded register files, while immediate values can be matched with either a numerical value or a jump target label in case of branch instructions.

If the instruction is successfully parsed and verified, the created object is stored inside of the parse list, which serves as an instruction cache during the simulation. In case of a failure, the parser stores each error message in a list along with hints where the user made a mistake and what was expected.

Code interpreters

As already mentioned, there are 3 types of interpreters. The Arithmetic interpreter is used to evaluate arithmetic expressions supplied by the instruction interpretable pattern. The Branch interpreter is used to calculate the difference between the code line position in the instruction cache and the target if the condition inside the interpretable pattern is true. The Load/Store interpreter is used to calculate address pointing to a value inside the simulated memory, and consequently, either load from or store the value to the address. The interpretable pattern inside the `interpretableAs` value uses the same abbreviations as the `instructionSyntax` value to link the parsed arguments inside the `InputCodeModel` object with the pattern.

Arithmetic interpreter

The Arithmetic interpreter is the core component of the ALU and Floating point unit. It takes the `InputCodeModel` and interprets it based on the matched instruction. The syntax of the `interpretableAs` field is as follows. The field contains multiple statements split using the semicolon. Each statement has format "*lvalue* = *rvalue*", where *lvalue* is the destination register, a part of the destination register or a temporary value. The *rvalue* is an arithmetic expression evaluated using the precedence table where the same instructions are left-associative. The supported instructions are:

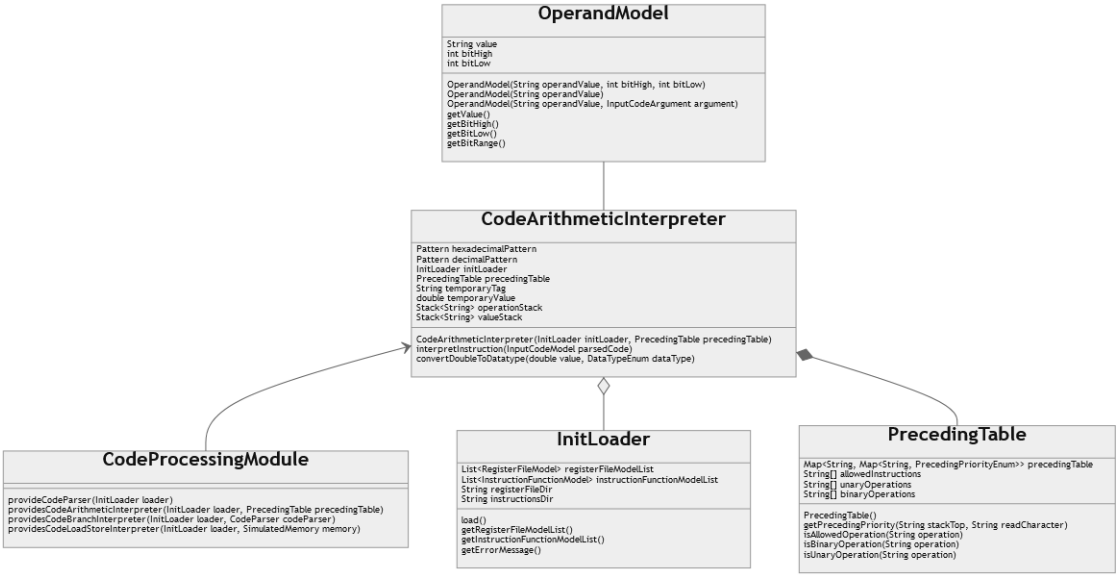

Figure 5.5: Class diagram for the Code arithmetic interpreter

- Addition, subtraction, multiplication, division, modulo
- Logical shift left, logical shift right, arithmetic shift right
- Brackets
- AND, OR operators and unary NOT operator

- Comparison operators greater than, greater than or equal, equal, less than or equal, and less than
- Unary operators for incrementing, decrementing and squaring a value, and a blank operator

An example let's have an instruction *multiply-and-add*, which takes 3 source registers and 1 destination register. The first two registers are added together and then multiplied by the third one. The `interpretableAs` value can look like "`rd=rs1+rs2;rd=rd*rs3`", or alternatively like "`rd=(rs1+rs2)*rs3`".

To make rotation instructions possible, the interpreter can also interpret writes to specific bits. This is done by specifying the input or output bit range in the square brackets starting with the highest bit, separated by a colon. It can also target only one bit by specifying the index of the bit. As an example, let us take an instruction which takes the destination register and sets the lowest bit and the 16 highest bits to 0. Such a pattern would be "`rd[0]=0; rd[32:16]=0;`".

Branch interpreter

The Branch interpreter is used inside of the Branch function unit to calculate the difference between evaluated branch instruction and its target based on the success of the condition. The interpretable pattern format is `(unsigned|signed):`*compareExpression*, where the `unsigned` or the `signed` value specifies how the source register's value should be handled and the *compareExpression* is similar to the single expression in the arithmetic interpreter's pattern, where only comparison operators are used. Since the interpreter expects a single expression, the pattern is not ending with the semicolon. In the case of an unconditional jump, the pattern contains a single word `jump`

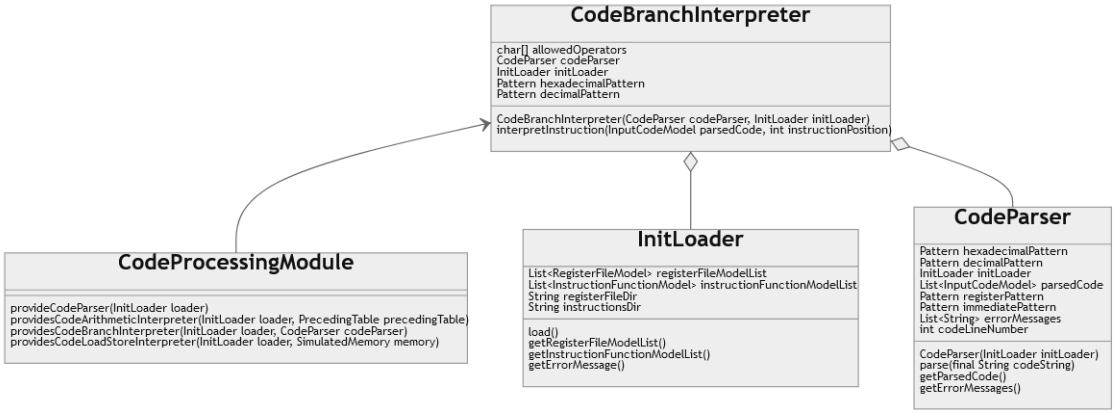

Figure 5.6: Class diagram for the Code branch interpreter

Let us take the RISC-V instruction `BGEU`, which is a "greater than or equal" branch instruction, where the source register values are compared as unsigned values. The pattern for such instruction would be "`unsigned:rs1 >= rs2`".

IF the condition is met or the jump instruction is unconditional, the interpreter calculates the offset between the current instruction and the target label. For that, each branch interpreter is supplied with the compiled list of instructions where the label and the position of the instruction can be searched. After that, the offset is returned as the result. If

the condition is not met, the resulting offset is 1 representing the next instruction in the program order.

Load/Store interpreter

The Load/Store interpreter is used for address calculations in Load/Store function units, for loading a value from the simulated memory in Memory Access blocks, and at the commit stage for storing a value inside the memory in Store Buffer.

The pattern for a load instruction is "`load` *dataType*`:(signed|unsigned)` *what where offset*" and for a store is "`store` *dataType what where offset*", where the first argument specifies how the instruction should be interpreted (`load` for load instructions, `store` for store instructions), *dataType* specifies the data type of the loaded or stored value inside the *what* argument. The *what* argument specifies either the destination register, where the value should be loaded or the source register, which holds the value to be stored. The *where* argument is a source register telling the interpreter the position in the memory where the value should be taken from or should go to. The *offset* is an immediate value which can be used to specify offset from the position pointed by the *where* value. The signed or unsigned flag inside the load pattern serves as a hint telling the interpreter whether the value in the memory should be loaded as a signed or unsigned value.

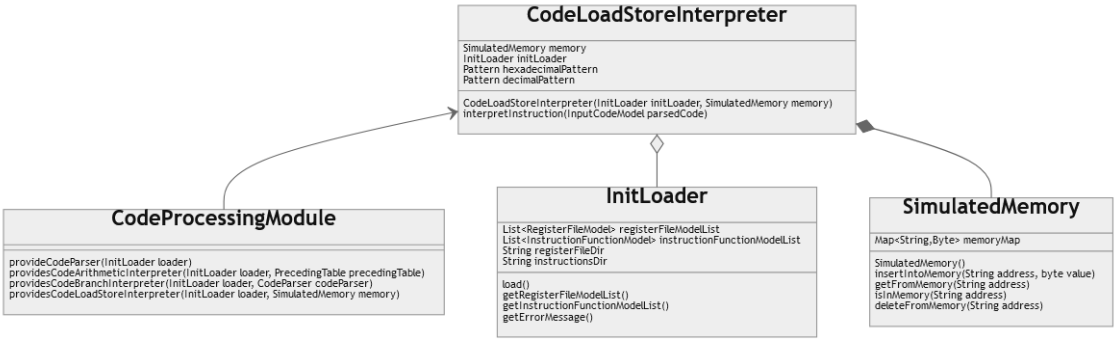

Figure 5.7: Class diagram for the Code load/store interpreter

As an example, we can consider the `LHU` (load halfword unsigned) instruction where the interpreter has to load a 16-bit value from the memory as an unsigned value. The pattern would be "`load half:unsigned rd rs1 imm`".

The interpreter has two functions, evaluating whole load/store instructions (meaning calculating address, recognizing if the instruction is load or store, and processing the recognized instruction), and simple address calculation. The address calculation is used in the Load and Store buffers to simulate load forwarding and bypassing. The whole instruction processing is used only at a certain point in the simulation (Memory Access for load and commit for store instruction), which will be explained in the next section.

The simulated memory is implemented as a hash map. The value that is inserted into the memory is broken into separate bytes, which are then stored on the address specified by the *where* argument. The endianness used by the simulated memory is little-endian, where the specified address is the least significant byte. If there is no value at the specified address or one or more bytes are missing, the simulated memory will generate random bytes which take place of the missing ones, and create the value from those bits.

5.3 Block layer

The Block layer is composed of the block classes, from which the superscalar processor is build. The implementations of the blocks are based on the RISC-V BOOM processor, where the blocks are mimicking the functionality of separate parts as closely as possible. Each block, that is modified at the clock tick, has to implement `AbstractBlock` class, which has 3 main methods that must be implemented. Those are:

- `simulate()` - simulate the behavior of the block during one clock tick
- `simulateBackwards()` - restore the state before the *simulate()* method
- `reset()` - clear all lists, maps and variables inside the block affected by the *simulate()* method

The `simulate()` behavior takes the values inside either the simulated block itself or the neighboring block and transforms them according to the block's logic. Each block's logic will be discussed in detail later in this section.

The `simulateBackwards()` behavior works on the ticket system, where the extended `InputCodeModel` class, called `SimCodeModel`, stores IDs from each block that it passes through during the forward simulation. The ID generation is block specific, where each block either gives out its ID when the instruction leaves or enters the block. When simulating backward, the block checks the ID specific to its stage and decides, whether the instruction should be moved back or not. If the `InputCodeModel` reaches the initial block, the object is unmapped from the simulation and destroyed.

All the blocks are controller using the `BlockScheduleTask` class controlling the order the blocks are executed in. All used blocks have to add themselves to the list of listeners before the simulation can start. The following subsections explain how each of the blocks implements their `simulate` and `simulateBackwards` methods.

Instruction fetch

The Instruction fetch is implemented by the class `InstructionFetchBlock`. The block takes the `InputCodeModel` objects from the parser, loads a certain amount of instructions, and makes them available for the decode and dispatch stage. The PC counter also resides inside this block and is incremented each cycle by the amount of fetched instructions.

The amount of fetched instructions depends on several factors. The most common is the *number of ways*, which is the upper bound of the number of instructions that can be fetched in one clock. This value along with the current PC counter value creates a range in the parser list which new instructions are pulled from.

Another factor is branching. Since the block does not use a trace cache, only one branch instruction can be inside the fetch range, which often results in fetching fewer instructions than the number of ways. The PC counter can be modified during the branch fetch if there is an entry inside the Branch Target Buffer.

The last factor is stalling, which happens when one or more buffers (Reorder, Load, or Store buffer) are full and the instruction inside the decode and dispatch will not fit. In that case, decode and dispatch tells the instruction fetch that it has to stall with the number of instructions that were pulled and decoded. Based on that knowledge, the instruction fetch fetches only a limited amount that fills all available empty ways. If all ways are filled and decode and dispatch was not able to pull any instructions the block will stall.

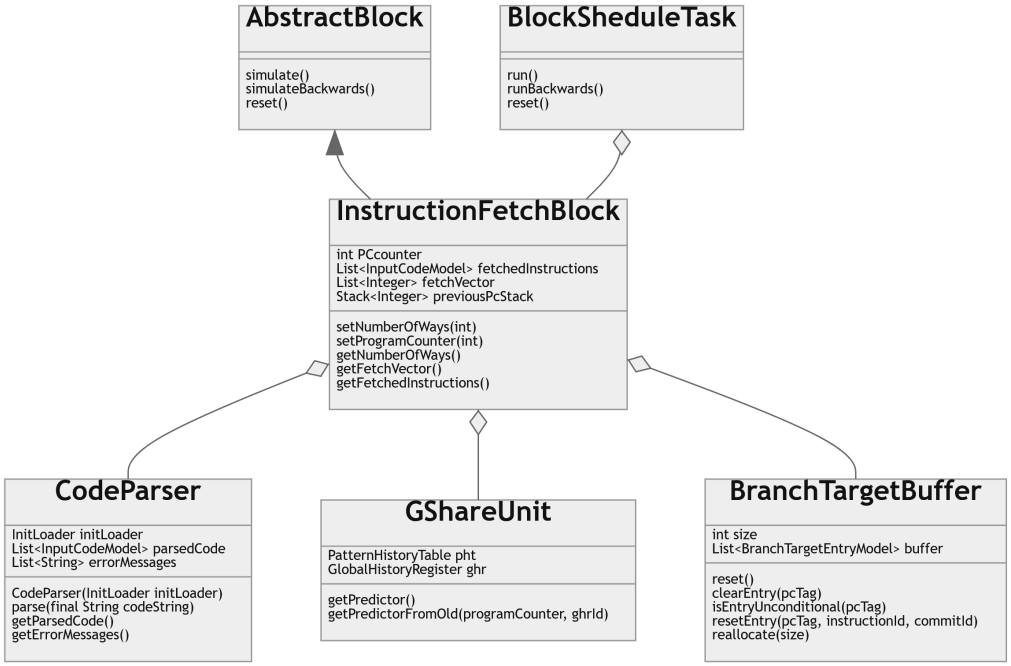

Figure 5.8: Class diagram for the Instruction fetch block

For restoring the previous simulation state, the instruction fetch block keeps a stack of previous PC counter positions, which are then used in the backward simulation. The position is pushed into the stack at the start of the `simulate()` method. When the previous PC counter value is restored, the backward simulation then proceeds to do the same steps as the forward simulation did, calculating the fetch range and then fetching the instructions.

Decode and dispatch

The Decode and dispatch stage is implemented in `DecodeAndDispatchBlock`. The block takes the fetched instructions from the Instruction Fetch block, extends the `InputCodeModel` objects to the `SimCodeModel` objects and renames the registers. This stage works closely with the Rename map table where all the speculative register mappings are stored. This block also filters the `NOP` instructions and unexpected labels.

After filtering, the rest of the instructions are renamed. The renaming has three stages, renaming to generic names, renaming the destination register, and renaming the source registers executed in order. The *renaming to generic* stage changes the registers in the code line string to its generic abbreviations, which is done with the help of the instruction syntax. After that, the *renaming destination register* stage is executed. Here the block uses the Rename Map Table to create a new map entry mapping the speculative register to the provided destination one. The last *renaming the source registers* stage renames the registers from the generic either to the same architectural register, or if Rename Map Table has a map entry for such register, to the last speculative register mapped to this particular architectural register.

Same as in the Instruction Fetch, the number of processed instructions in the Decode and Dispatch block can be reduced by either the number of branch instructions or buffer stalling. The branch instructions can cause mispredictions during the Instruction Fetch

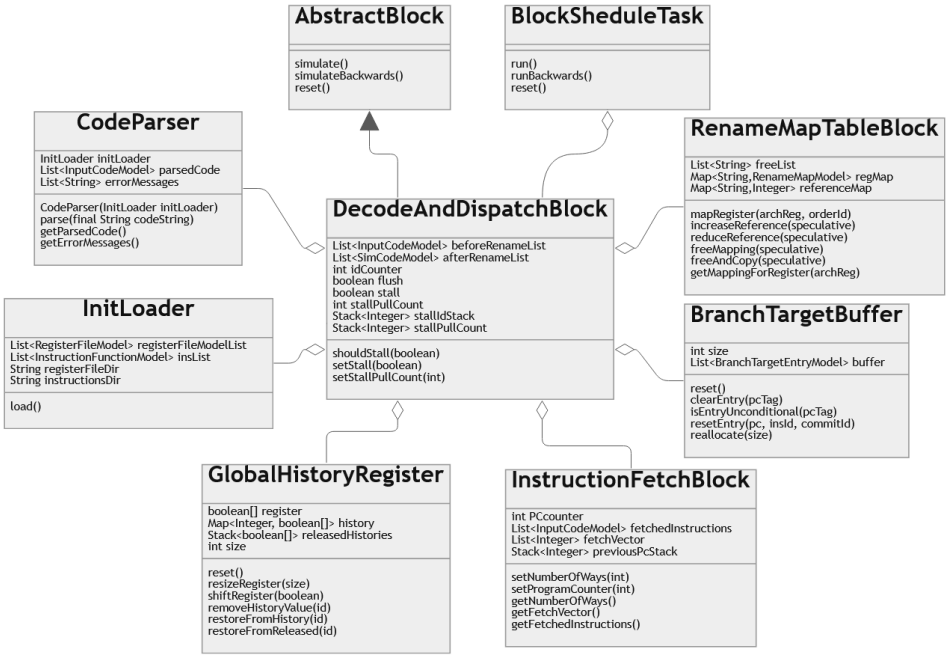

Figure 5.9: Class diagram for the Decode and Dispatch block

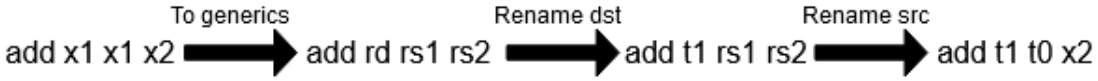

Figure 5.10: Decode stage renaming

stage caused by the Branch Target Buffer having an old target address causing the program flow to fetch instructions from a wrong address. In the case of misprediction, all instructions after the branch instruction have to be removed from Decode and Dispatch and a new value of the PC counter has to be set. The buffer stalling happens when one of the buffers is full. The Reorder Buffer notifies the Decode and Dispatch of such an event with the number of successfully pulled instructions into the buffers. The Decode and Dispatch then remove all pulled instructions and tries to pull new instructions from the Instruction Fetch. The limit is specified by the number of ways the instruction fetch has.

For the backward simulation, the Decode and Dispatch gives out numerical IDs from its ID counter, specifying the order in which the instructions leave the block. The ID is then later used for state restoration a step ago. When the instruction is supposed to leave the block during the backward simulation, it has to unmap all architectural registers from the Rename Map Table. The block also keeps a history of the moments when it was stalled and how many instructions have been pulled from the Instruction Fetch to know the number of instructions to be removed in one step. The moments in history are marked with the values from the ID counter.

Issue Window SuperBlock

This block connects the in-order part of the processor with the out-of-order side. When the `simulate()` method is called, it takes all instructions in the decode block, that were loaded in the Reorder buffer, and transfers them to their appropriate Issue window. For

selecting the appropriate window for the instruction, each window provides a public method for checking whether the instruction can be placed inside the selected window or not. When the suitable window is found, the instruction is moved from the decode to the found window. This repeats until no more instructions can be pulled.

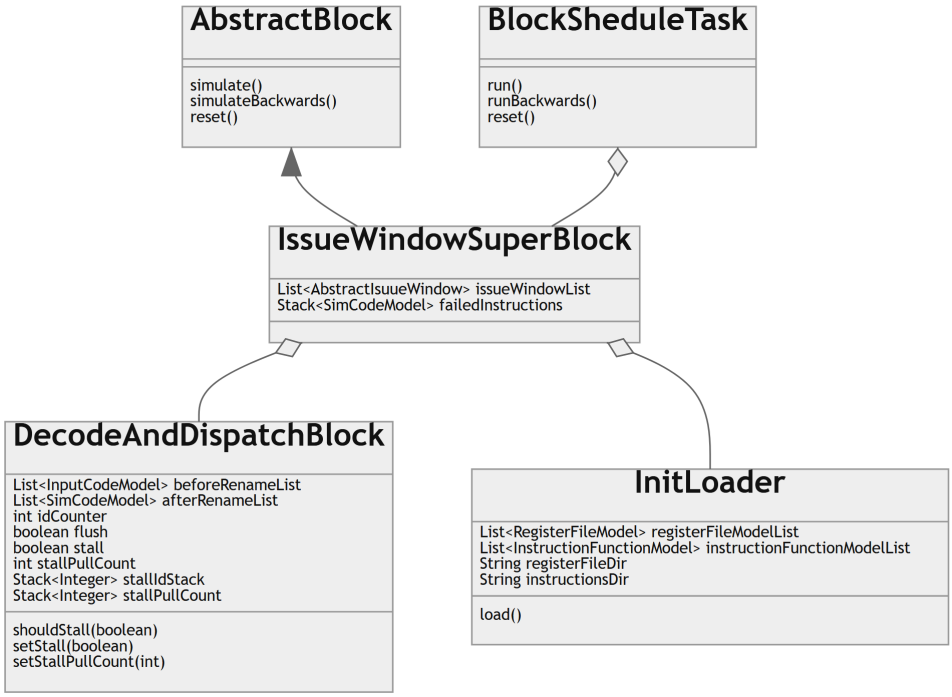

Figure 5.11: Class diagram for the Issue Window SuperBlock

When simulating backward, the superblock checks using the Decode and Dispatch ID counter which instructions can be pulled back to the Decode and Dispatch, where the speculative registers are freed. In case of instruction flush, the superblock stores failed instructions and releases them on Decode and Dispatch ID counter match.

Rename map table

The Rename map table is used to map any architectural register to a speculative one to avoid WAR and the WAW hazards. The table keeps names of all speculative register in a *Free list*. When the decode stage asks for a new mapping of the register, the table checks the *Free list*, takes the first speculative register, and saves the mapping in a hash map. The table also keeps a reference counter to each of the mapped speculative registers. Until the count does not reach 0 and references are not freed, the speculative register holds the result value. As soon as the counter reaches zero, the speculative register is added to the Free list and the value is copied to the mapped architectural register.

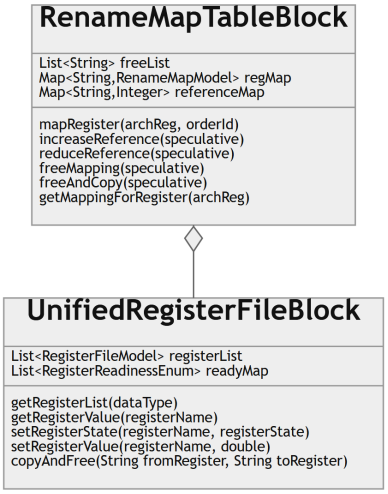

Figure 5.12: Class diagram for the Rename Map Table Block

Unified register file block

The Unified register file block holds all architectural register files as well as the speculative register file, crated based on the size of architectural ones. The architectural register files are pulled from the `InitLoader` during the simulator initialization. The register block also keeps the state of each register in a hash map where the architectural registers are always assigned and only the states of the speculative ones can be modified.

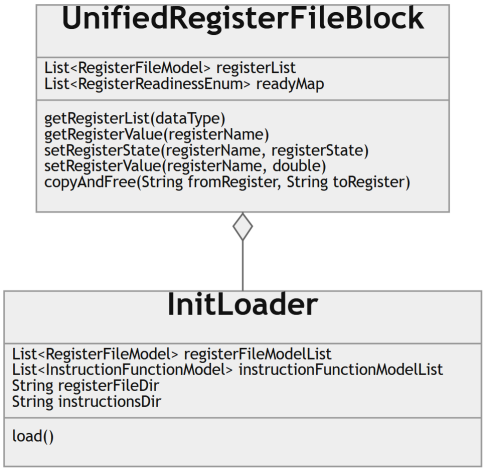

Figure 5.13: Class diagram for the Unified Register File Block

The states of all speculative registers are set to Free during initialization. When the register is mapped in the decode stage, the speculative register is set to Allocated. After an instruction is executed, its destination speculative register is set to Executed. And when the instruction is ready to be committed, the register is set to Assigned and the value is held until all mappings for a certain speculative register are freed. Finally, the register becomes Free and can be allocated again.

Issue window blocks

The issue windows are used to queue instructions before they are ready to be executed inside an appropriate function unit. The issue window works as an Age-ordered queue, where the elder instructions have higher priority than the younger ones. To track the readiness of each instruction, the issue window has list of `IssueItem` objects for each instruction, which is used to track the readiness of an instruction.

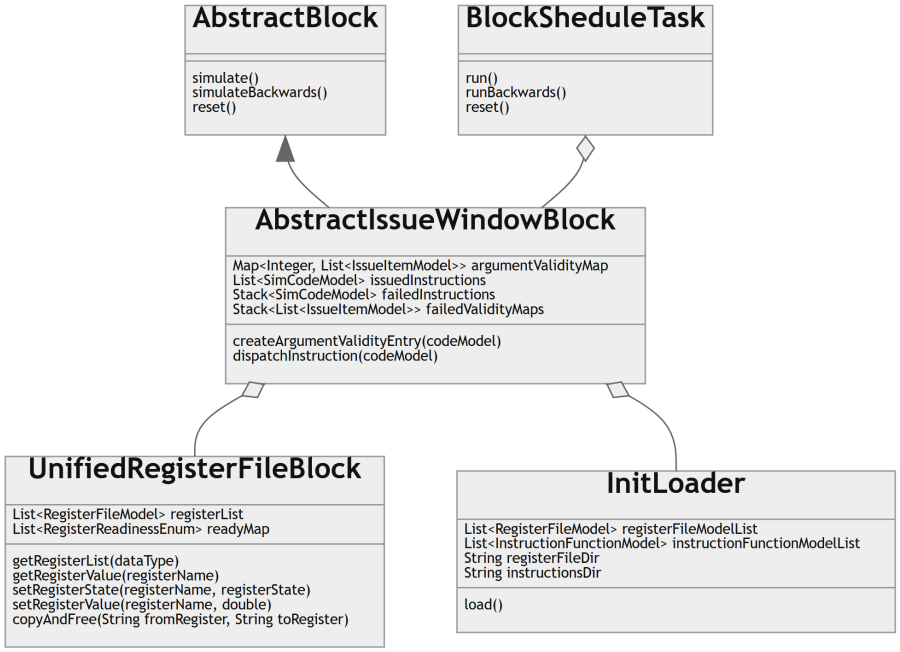

Figure 5.14: Class diagram for THE Abstract Issue Window Block the other windows are created from

The `IssueItem` object list stores information about destination tag, source tags, their values, and validity bits. The validity bits are configured at each clock cycle. The state of the source register are observed, and if the state is either Executed or Assigned, the valid bit is set to true and the register value is copied inside the `IssueItem`. When all validity bits are set, the instruction is ready to be executed. Issuing is done in the Age-order until all function units associated with the issue window are filled with compatible instructions, or no other instruction is ready.

Instructions are dispatched into the issue windows according to their type. There are 4 Issue windows in the simulation being ALU (FX) issue window, Floating point issue window, Branch issue window, and Load/Store issue window. The conditions for dispatching are:

- ALU issue window - Instruction has to be arithmetic and all data types are integers
- Floating point issue window - Instruction has to be arithmetic and at least one data type (input or output) is not an integer
- Branch issue window - Instruction has to be a branch instruction
- Load/Store issue window - Instruction has to be memory instruction (either load or store)

Backward simulation is done by pulling instructions from the function units when the backward simulation inside the function unit is complete. Each window also has its ID counter where the instructions are marked by the counter value when they leave the issue window. The instructions then update their lists of `IssueItem` objects until they are pulled by the Issue window superblock to the Decode and Dispatch. In the case the instruction flush caused by misprediction, each issue window has its own stack for failed instructions with the counter value when being flushed for previous state restoration.

Function units

The function units are the computing core of the simulator working closely with the appropriate interpreters to give out instruction results. The instructions are placed inside the function unit block by the issue window. To simulate the computation delay, each function unit has a set delay. When the unit registers that an instruction has been placed into the unit, its counter is incremented each clock cycle. When the counter reaches the delay value, the interpreter is ask for execution and modification of the destination register. The function units are split into 4 categories being Arithmetic, Branch, Load/Store address calculation, and Memory access.

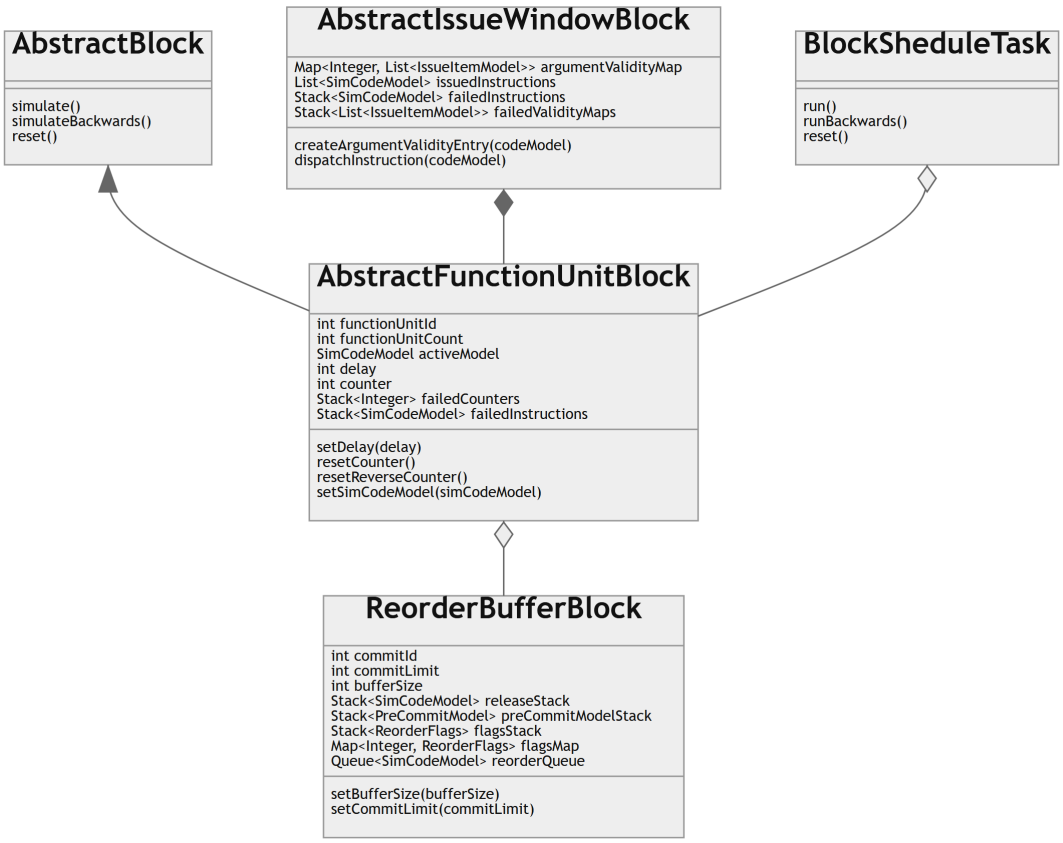

Figure 5.15: Class diagram for the Abstract Function Unit Block other function units are created from

The Arithmetic function unit instances are used by the ALU and Floating point issue windows where each window has its own list of function units. They use an Arithmetic interpreter to evaluate the active instruction inside the unit. Each arithmetic unit has an

array of instructions, which can interpret. This limits access to some instructions which can be used to introduce a difference between multiplication and division function units, for example.

The Branch function unit instances are used by the Branch issue window. They use a Branch interpreter to get the jump offset, and after that, they calculate the exact address of the jump. Because the branch instructions require additional information, the argument list of the `InputCodeModel` is appended during the simulation. The appended arguments are: the PC counter value, if the instruction have taken the branch during the decode stage, and address of the taken branch. The PC counter value is used to compute the real target address, which is appended to the list for later evaluation during the commit stage.

The Load/Store address function units are used by the Load/Store issue window. They use the Load/Store interpreter's address calculation feature to give the address pointing to the first byte. The result is then written into either Load or Store buffer according to the instruction type.

The Memory access unit processes load instructions to return the data at a certain address. Its instances are used by the Load buffer. The instructions are only issued when the data address is ready, and no store address is blocking it. The result is written inside the destination register after a predefined delay, and the data flag inside the Load buffer is set to true.

The backward simulation is done by pulling instructions from the Reorder buffer when the match the Function unit ID counter. Each function unit inside the issue window list generates unique values to correctly restore the previous state. The unique ID generation is done by setting the initial ID counter value to position in the Issue window list. Consequently the set offset is set for each function unit to the count of function units in the list. Therefore, no two ID counter values of different function units will be the same. The ID is set when the instruction enters the function unit and is incremented by the offset each time there is no instruction in the function unit. In the case of instruction flush, each function unit has a stack for failed instructions, instructions are pulled from when the ID counter value matches the ID saved inside the instruction model. The function unit counter is restored with the instruction as well.

Reorder buffer

The Reorder buffer block serves as a queue for keeping instruction in their program order. It is the last block before the changes caused by the instruction are committed into the memory of the register file. The Reorder buffer can commit only a limited amount of instruction specified by the commit limit value in the buffer, which can be configured.

Each instruction has its reorder flags specifying the readiness of the instruction. The flags are: `isValid`, `isBusy` and `isSpeculative`, representing whether the instruction is valid, busy, or speculative respectively. The valid bit represents the validity of the instruction which can be changed by misprediction or bad load forwarding. When it is set to false, the Reorder buffer removes such an instruction from the queue. The busy bit is set to false when the instruction enters the issue window and is set to true when the instruction is evaluated in the function unit. The speculative bit is set to true for all instructions preceding a speculative branch instruction stay speculative until the branch instruction is evaluated. The speculative bit is also set for instructions preceding a speculative load forwarding. When the speculative instruction is committed and there is no misprediction, the reorder buffer starts setting the speculative bits to true until it reaches another speculative

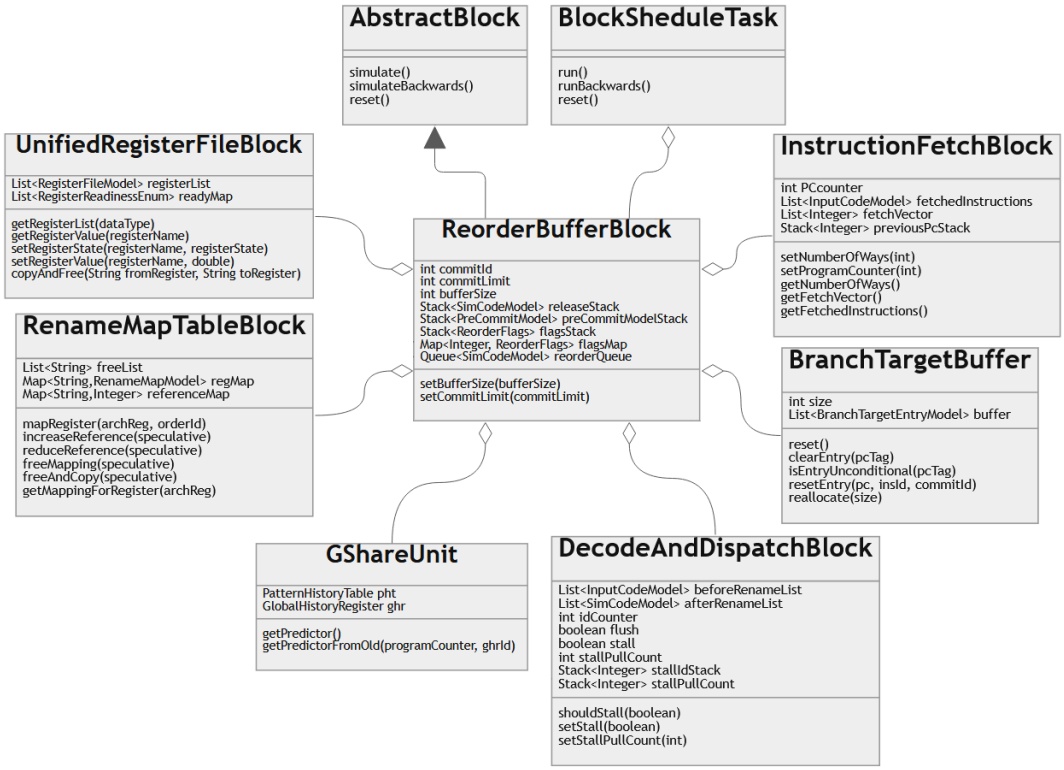

Figure 5.16: Class diagram for the Reorder Buffer Block

instruction. When the Reorder buffer commits the mispredicted instruction, all instructions in the Reorder buffer are set to invalid and flushed from the buffer. The PC counter value is set to the correct position consequently. The position is either the correct branch target or a PC counter value of the first failed instruction. The instruction is ready to be committed as soon as the valid is set to true and the busy and speculative are set to false.

Each clock cycle, the reorder buffer is responsible for pulling newly decoded instruction into the buffer. It also has to create a new flags object for each instruction. Before pulling from decode, the buffer has to ensure the buffers have enough available space by comparing the queue size and its limit. If one buffer failed to pull decoded instructions due to overflow, the Reorder buffer would rise the stall flag and the count of the instructions that it was able to pull. During the backward simulation, the buffer is responsible for removing all instructions that appear in the Decode and Dispatch block.

When the instruction is committed, all the speculative registers inside the instruction have to lower their reference count, and the instruction has to be saved for later backward simulation. The released instruction, either released on commit or removal, are therefore stored in the release stack together with the reorder flags. The released instructions are also marked with the commit ID counter value used in the backward simulation. The Reorder buffer block also stores the register values before commit and their mappings.

Store buffer

The Store buffer keeps track of all in-flight store instructions. The instructions are stored in the queue with additional information about the store address and the state of the source register kept in the `StoreBufferItem` object. The store instructions are loaded into the

buffer in the decode stage and kept there until committed. When being committed, the store buffer simulates the writeback by calling the Load/Store interpreter and writing the data to the memory. All store instructions during its time between the decode and commit stage are used for load bypasses and for stopping the speculative load forwards.

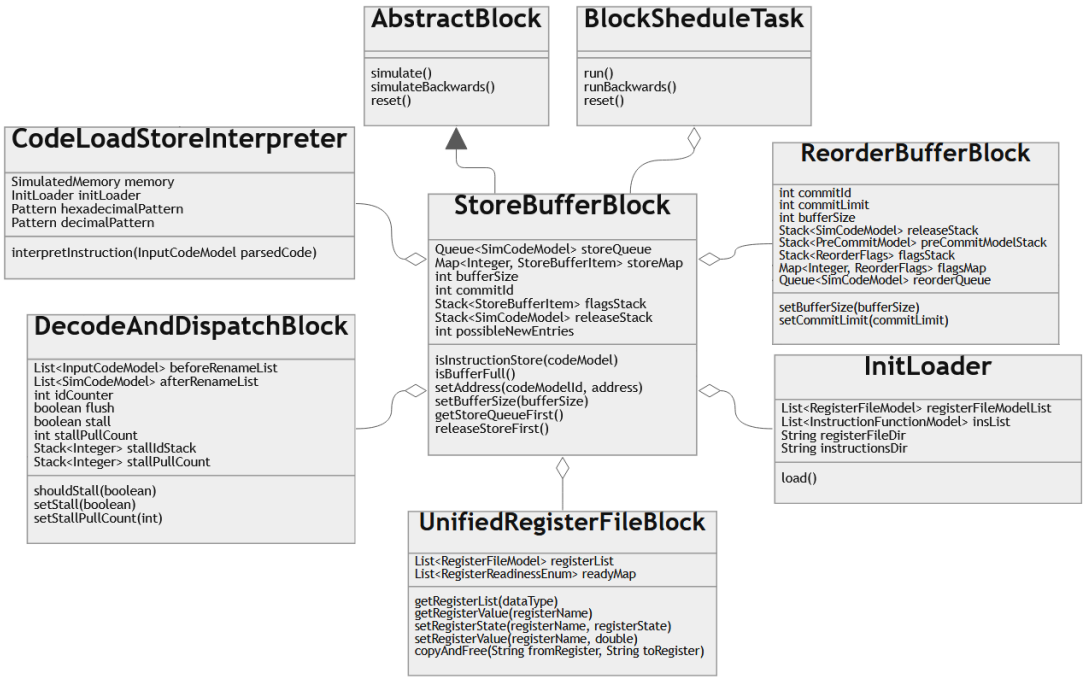

Figure 5.17: Class diagram for the Store Buffer Block

Like the Reorder buffer, the store buffer has to pull store instructions from the decode. Since it is allowed to pull only store instructions, the buffer block has an internal filter permitting only this kind of instructions. Apart from the Reorder buffer, the store buffer does not rises the stall flag. It only has to stop pulling when becoming full. During the backward simulation, store instructions return to the decode stage from the Load/Store issue window, so the store buffer has to remove all store instructions that appear in the decode stage, as if it was returning the instructions to the decode stage.

the store buffer also keeps all committed instructions in the stack for the state restoration and uses an ID counter, which values are the same as in the Reorder buffer. Apart from instructions, the Store buffer keeps track of the `StoreBufferItems` also being restored with the instruction.

Load buffer

Similarly to the Store buffer, the Load buffer keeps all in-flight load instructions in its queue with its `LoadBufferItem` object, which has the information about the address, whether the load bypassed or not, if the Memory Access unit was accessed and when, and if the destination register is ready. The address is gained when the Load/Store address calculation is executed.

Apart from the Store buffer and its instructions, load instruction has to go through the memory access stage. This is implemented by the Memory Access unit, where the instruction is simulating the access to the memory. This can be bypassed if there is a store

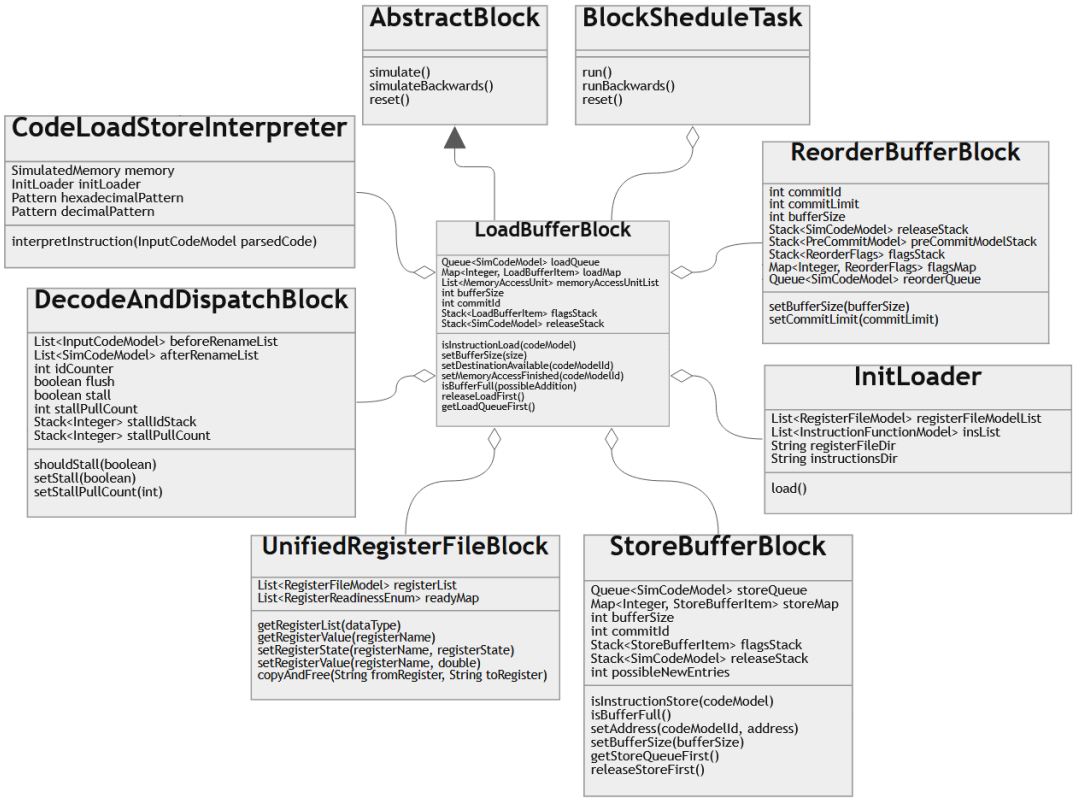

Figure 5.18: Class diagram for the Load Buffer Block

instruction having the same address, which would cause to skip this stage and take the value from the store buffer resulting in the *load bypass*. The *load forwarding* policy inside the RISC-V BOOM is optimistic. Thus, if there is no match with the store buffer addresses and the load address is ready, the instruction is placed into the Memory Access unit. The result is kept until the instruction is committed, or there is a match causing the memory ordering failure. The instruction that has been forwarded has to check each cycle until it is at front of the Reorder buffer if there is another store instruction with a matching address.

the Load buffer behaves similarly to the store buffer when pulling the instructions into the buffer, however, it only has to pull load instructions. Like the Store or Reorder buffer, the released instructions are kept in a stack with the additional `LoadBufferItem` for later state restoration.

Branch Target Buffer

The Branch target buffer serves as a buffer for different branch instruction entries and their targets. Each entry has information about the PC of the branch instruction, whether the instruction is conditional or unconditional, and the target of this instruction. The entry also keeps the order ID of an instruction and also commit ID when the instruction gets committed. These last two values are used in the backward simulation.

The Branch target buffer is accessed in three stages, instruction fetch, decode, and commit. The instruction fetch stage uses BTB to get an entry for the currently fetched instruction and possibly update its own PC counter. The update happens if there is an entry inside the BTB and the target is not set to -1. The instruction is then marked

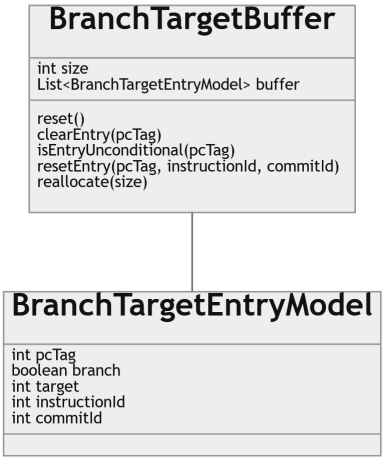

Figure 5.19: Class diagram for the Branch Target Buffer Block

according to have been taken or not, which is then used in the next stage. In the Decode and Dispatch, the prediction is checked whether have been correct or not by consulting the PHT's bit counters. If there is a mismatch between the BTB target and the prediction, the entry is updated and the instructions after the evaluated branch has to be flushed. When the instruction gets on top of the Reorder buffer and becomes ready, the third stage happens. In the commit stage, the result from the Branch function unit is compared with the existing BTB entry. If there is no mismatch, and therefore, the prediction was correct, the instruction is committed and the PHT's bit vector is updated.

Indexing into the BTB is done by using the lower part of the PC counter. The number of entries in the BTB can be configured, but it has to stay constant during the simulation. The BTB also stores old values that get rewritten for state recovery during the backward simulation.

Pattern history table and bit predictors

The Pattern history table is composed of separate bit vector counters used to predict whether the branch should be taken or not. The bit vector counters can be configured to be either zero, one, or two-bit wide with each implementing its own logic. The zero-bit counter is set during the simulation initialization and the value does not change during the simulation. The one-bit predictor changes to either *Taken* or *Not taken*. The two-bit predictor has two more states telling whether the prediction is *weakly* or *strongly* taken or not. The behavior can be seen in figure 5.20.

Same as the BTB, the PHT is accessed in the instruction fetch and at commit. In instruction fetch, PHT is used to determine whether the branch should be taken or not. At commit, the prediction is compared with the result and updated accordingly. Each bit predictor has to implement the behavior on increasing or decreasing the probability. Same as the BTB, each PHT entry has to remember previous states of its bit vector when the state restoration is needed.

Global history register

The Global history register is a bit vector. Each bit is marking according to the branch being taken or not in the program order. The register is updated in the decode stage, and

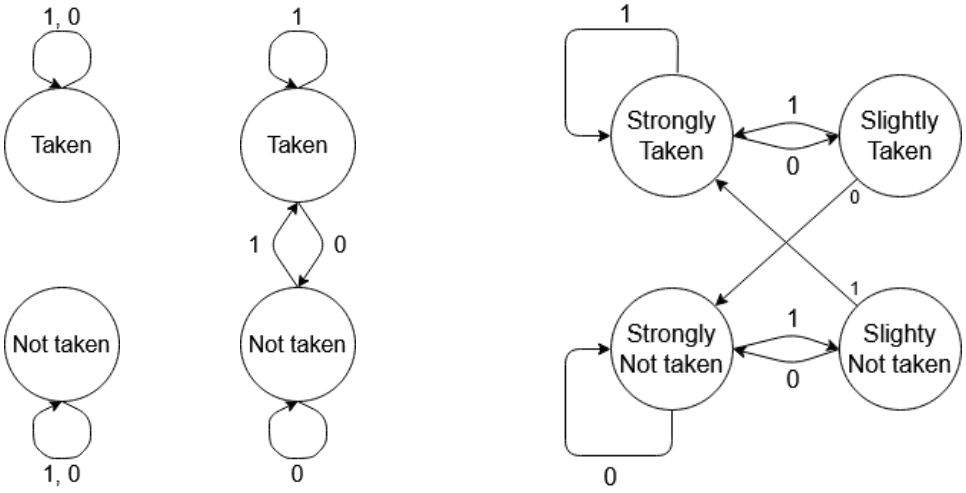

Figure 5.20: From left: Zero bit predictor, One bit predictor, and Two bit predictor

if misprediction happens, also at commit. Because of the mispredictions, the register has to remember previous bit vector states to either restore and fix, or to correct PHT indexing. When the branch instruction reaches the commit stage, the old values are released from the list of history values to the stack of history values later used in the backward simulation. An example of a bit vector can be a value 00001101. It can be seen that from the last eight branch instructions, five were not taken (marked by 0 bit), 3 were (marked by 1 bit), and the last instruction was taken.

GShare unit

The GShare unit is used in branch prediction to give an index to the certain PHT's counter based on the hashing function. The hashing employs the XOR function between the lower part of the PC counter and the Global history register bit vector. The Global history register introduces the globality of the prediction, making the prediction dependant on the branching pattern. The PC introduces the locality of the predictions. The GShare unit is only used for the hash computation, therefore, it does not have to save any previous states as long the PHT and the GHR can successfully restore themselves.

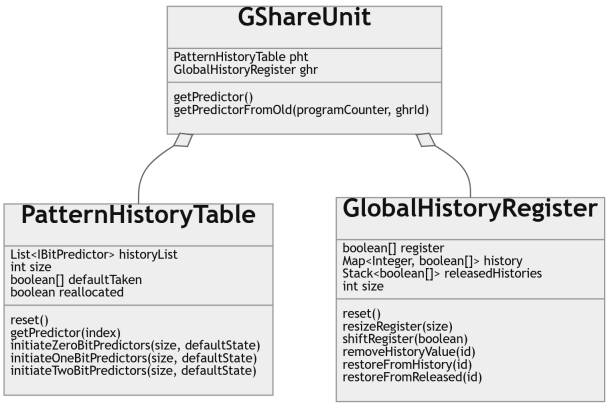

Figure 5.21: Class diagram for the Gshare Unit

5.4 UI layer

This layer provides a wrapper for all mentioned blocks in the Block layer, while also providing control features to interact with the simulator. The simulator uses JavaFX for views and controllers. Every unique block has its own view and a controller specifying the looks and behavior. The UI layer can be split into 3 main categories, the Simulation window, the Code window, and the Configuration window.

Code window

The Code window serves as a simple text editor for inserting the assembly code into the simulation. The instruction in the code has to correspond to the instructions loaded from the ISA folder. The code window provides a menu for creating a new code list, loading the existing one from a file, saving the code into the file, and compilation by calling the `CodeParser.parse()` method for parsing and loading the input code into the simulation.

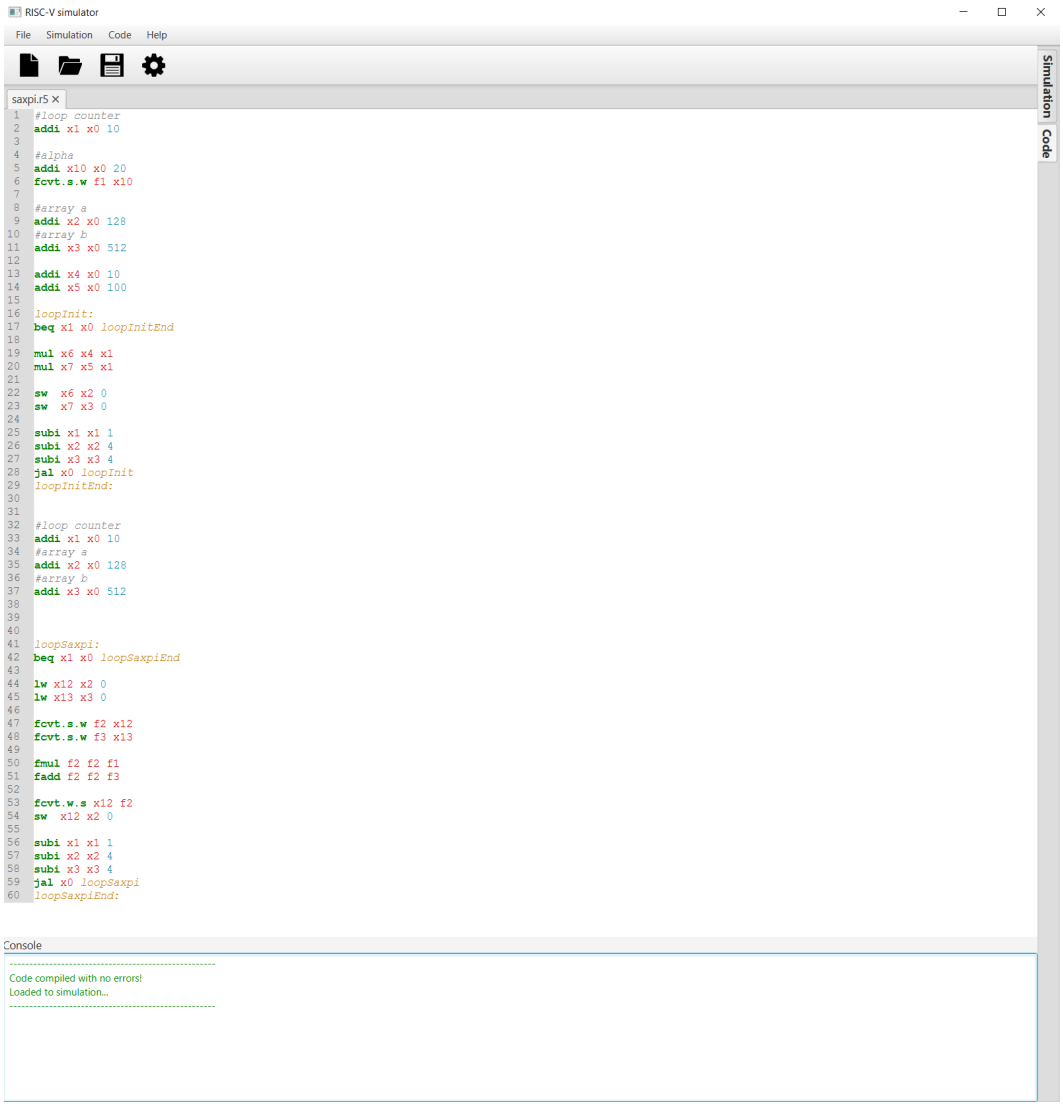

Figure 5.22: Code window

The code is inserted into the upper text code area. The text area supports highlighting of the input source code, where the highlight color is chosen based on the code type, whether it is an instruction, a register, an immediate value, or a label. When the user is writing down the source code, the code text area is dynamically creating a list of valid labels, so if there is a branch instruction pointing to an existing label, it will be highlighted. The lists of instructions and registers for highlighting are created from the loaded register files and instructions upon the app has started or when the configuration has changed.

In case of a successful compilation, the console text area will write out a message informing of a successful compilation and a dialogue will be shown to switch from the Code to the Simulation window. In case of failed compilation, the console shows the formatted output of the CodeParser's error log giving hints where the users should look for mistakes.

Simulation window

In the Simulation window, users can see how the code is being processed at different stages. The window offers a control menu to run the simulation for a specified period, or step by step. Other controls include resetting the simulation to the initial state, stepping to the previous step, and fast-forwarding to the simulation end.

The block wrappers can be seen in the Simulation view in the center of the screen. On the left side, a list of compiled instructions can be found. During the simulation, the fetch range is highlighted. The highlighted places are taken from the *fetch vector* created during the simulate method in the instruction fetch. The fetch vector's size is always the same as the *number of ways* of the Instruction fetch block. If the fetching is stopped due to loading multiple branch instructions, fetch vector is filled with the incremented values of the last position in the vector. The top window shows all architectural registers in their current state. The individual register values change during the simulation after their mappings have been released by the Rename map table. At the bottom the user is able to see statistics window, which shows information about CPI, clock ticks, number of committed and failed instruction, and branch prediction accuracy.

The main Simulation view shows all blocks wrapped behind the `TableView` primitive, where all the important values for each block are shown. There are different policies for each block when it comes to highlighting. The issue windows and the Reorder buffer highlight the instruction row in green once they are ready. The branch blocks and Rename map table move the new values on the top of the table and highlight them in yellow. All bit values are interpreted by some string value, either "YES" and "NO" in case of Reorder flags and Issue items, or "READY" and "WAIT" in the case of the Load and Store buffers. Each block is also connected with the line to the blocks which it interacts to. The function units are represented by a list of function unit blocks, where the number of units can be changed and their properties either configured or changed in the Configuration window.

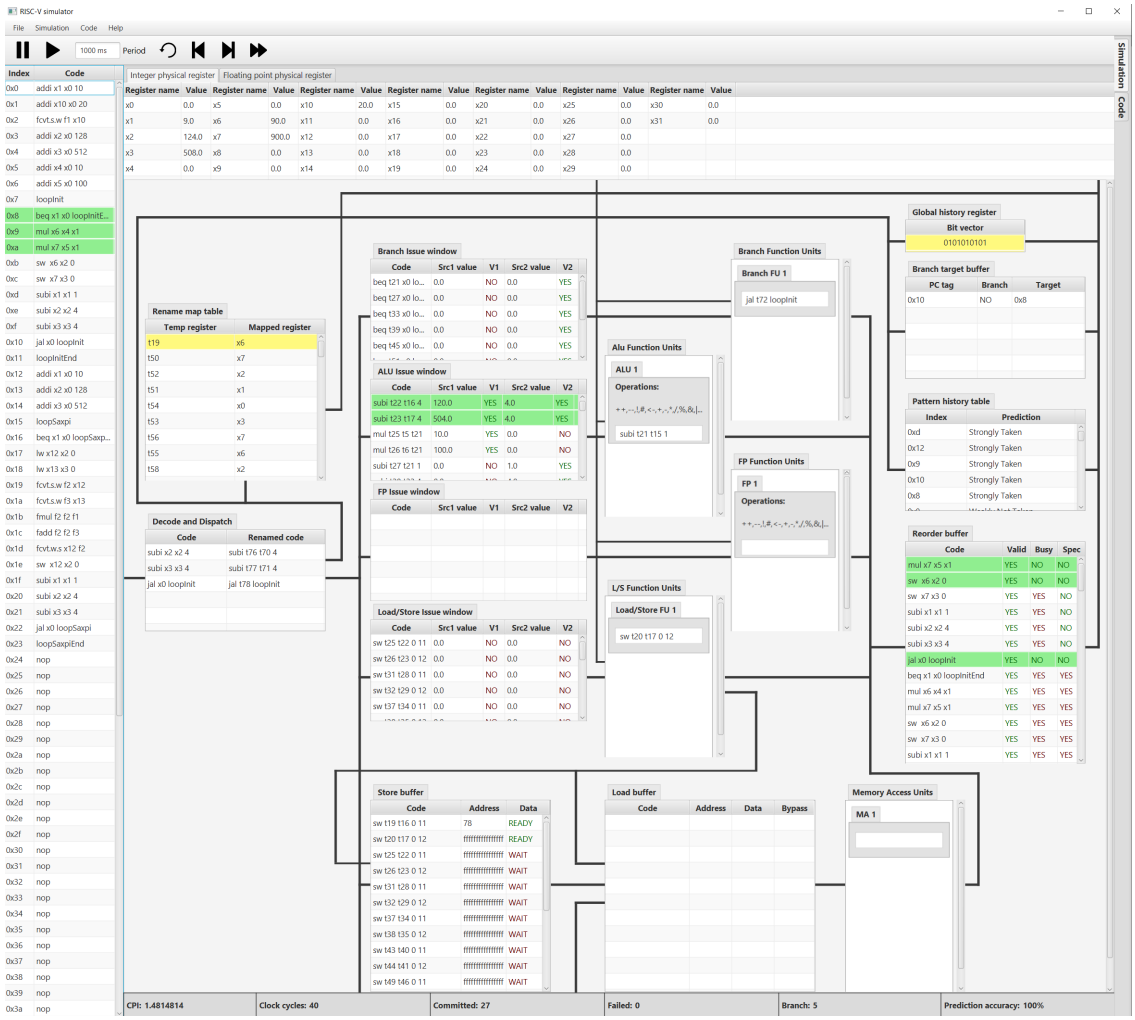

Figure 5.23: Simulation window

Configuration window

The last of the three mentioned windows is the Configuration window, where the simulation properties can be changed. The configurations are hidden under tabs named Loader, Buffers, Function unit, Branching, and Fetch and Commit.

In the Loader tab, users can specify the location of a new ISA folder or register file folder, which will be loaded on the confirmation of the dialogue. The Buffers tab allows to set size limits for the Load, Store and Reorder buffers, which would cause stalling for the lower values or free execution without stalling for the higher values. The Branching tab allows to configure the BTB, the PHT, and the GHR sizes and allows to change bit predictors and their initial state. The Fetch and Commit tab has a form for configuring the number of ways specifying how many instructions can be fetched in one clock cycle, and for configuring the commit size, specifying how many instructions can be committed in one clock cycle.

A special tab is the Function unit tab. This tab serves for creation of new function units. The tab provides a tool for "what if" scenarios, where users can add or remove certain function units. While adding the arithmetic function units for either the FX or the

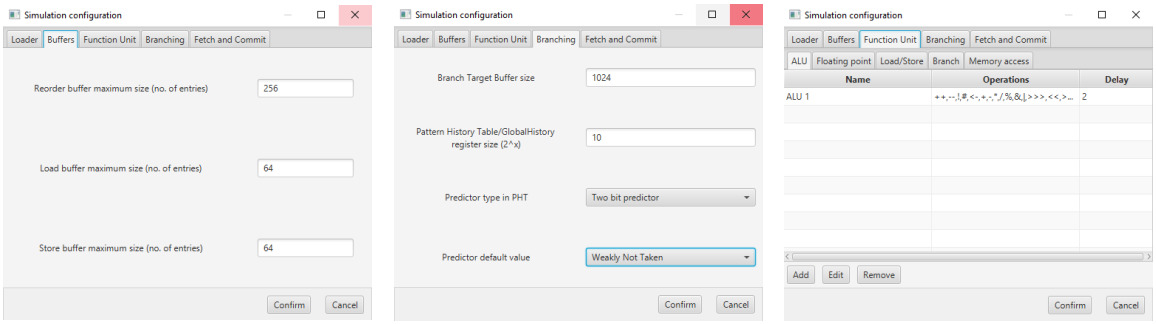

Figure 5.24: Configurations for buffers, branching and function units

FP issue window, users are also able to limit the functionality of a unit by specifying which instructions are allowed in that unit. All function units have configurable delays to make certain units faster than others. The tab also provides editing of already existing function units. When the configuration window is closed, all the function unit changes are rendered inside the function unit lists.

Chapter 6

Testing

To test this application before going public, these steps have to be taken. The loader layer has to ensure correct parsing of the JSON files ensuring that correct files are loaded into the application and forbid loading of false files with wrong JSON structure.

In the Code layer, the program has to be able to parse the source code according to the loaded instructions and verify arguments based on their type. If the argument is a register, it must be one from the loaded register files and forbid any other inputs. If the parsing is supposed to fail, it should be able to spot the mistake and log such incidents for higher layers. The interpreters must be able to interpret the source code according to the rules specified in loaded instructions. The arithmetic interpreter should be able to provide a sufficient array of instructions to implement most RISC functions and some CISC functions for experimenting. The branch interpreter needs to provide a correct offset where to move the program counter. The load and store interpreter has to provide simulated memory for loading and storing and also it should be possible to load only a certain part of the memory.

The block layer must provide expected results at the end of the simulation after a certain number of steps were completed. The main testing focus should be on the ability to simulate in both ways (forward and backward). It has to provide a simulated implementation of the Tomasulo's algorithm. There should be a noticeable increase in CPI when the instruction has a higher number of function units or function units with a small delay or by increasing the buffer size or all of the above. In contrast, there should be a decrease in CPI if there are fewer function units, function units with higher delay or really small buffers, or all of the above. The simulator should be able to simulate load forwarding and bypassing. Testing should also focus on the branch prediction and its impact on the order program execution.

In the UI layer, testing should focus on the correctness of displayed values according to the simulation time. The code input should highlight only loaded instructions. The changes in the configuration should be distinguishable either before or during the simulation.

6.1 Business logic testing

All the bottom layers were tested using the JUnit framework for creating Unit tests with Mockito[1] for mocking the needed classes from other layers. The Loader layer was tested on several JSON inputs with different config values, missing lines, or corrupted files. In the code layer, I tested if the parser can parse a specified source code based on the mocked loader values, and whether it can spot mistakes at specified places.

[1] https://github.com/mockito/mockito

Since the arithmetic interpreter is the most feature-heavy out of all of the available interpreters, there are multiple unit test files, testing the different operators and their combinations, e.g., multiply and add, using operators with different precedence, and indexing certain bits. In the load and store, the testing was focused on the memory manipulation and loading of different byte sizes, where for example, an integer value was stored and after that loaded as a short or byte, or a float was stored and when pulled by an integer load function, the computation resulted in an invalid value. The branch interpret was tested by mocking a code snippet and presenting a branch instruction with conditions of different outcomes.

The block layer tests verified the functionality by constructing programs that cause a RAW hazard, load forwarding and bypassing, branching, and so on. The validation was done by checking the values inside the blocks at each step of the simulation. The backward simulation was done similarly by simulating its end and then stepping it back to the initial state, while checking the simulation state.

6.2 UI and Application testing

The UI testing was done by hand by checking the simulation table views during the simulation and its correct highlighting when instruction was either ready or changed. The block layer was also tested by pressing the control buttons in different configurations, where a few bugs were discovered and promptly fixed. The Code window was tested by trying different inputs and seeing how they get colored. Moreover, the behavior of new, open, save, and compile options was verified.

The Application was tested on different platforms, which covered Windows 10 64-bit, Manjaro Linux with I3 display manager, and Ubuntu 20.04 and 16.04 with gnome display manager. Based on these testing, I've created a package for all these systems with instructions on how to run the application. This was tested on a couple of volunteers, which I provided with a package and a survey, for additional suggestions. I was able to get 5 people to go through the survey and based on their answers I've been able to do some UI style fixing and I've able to verify that they were able to run the application with provided instructions.

Chapter 7

Conclusion

The goal of this thesis was to create an interactive simulator where users can change certain configurations and see the behavior on their source codes. This goal was met.

In the second chapter, i summarized required information about pipelining, data hazards, and different parts of the suprescalar processor. The third chapter includes my review on current existing simulators, where i talked about its features and possible upgrades. The forth chapter chapter includes my proposed system, in which i created mockup and planned my approach to the problem. Then in the fifth chapter, i described the my implementation, layer by layer, as was proposed. This concludes items 1 through 6 in the thesis specification.

The current state of the simulation can show most of the things that compose the core of the AVS course, being RAW dependency, Tomasulo's algorithm for out-of-order execution, load bypassing and forwarding, and also all the different units inside the superscalar processor. Apart from other simulators, I've devised a system of modifying the count of function units into the system, which can illustrate speedup or slowdown depending on the configuration. Another thing different from all of the reviewed simulators is the visualization and deterministic approach to processing branch units, which can be also used to explain branching processes. Due to its modular approach, other ISA's can be programmed into the simulator, as long as it follows syntax restrictions. Thanks to the layer layout, the application is fairly easy to extend and implement or modify certain parts.

A limitation of this simulator is the lack to configure the number of speculative registers. Another limitation is the issue window size. Both of these can be easily added by adding a checked on register/space exhaustion and link it to existing stalling mechanism. Another limitation is the size of the simulator, where the blocks are stretched over the Y axis of the window, which results in lack of ability to fit everything in one window and needs to be scrolled to. I've placed similar blocks to each other, which allows to see behaviour of certain parts of the simulator in one place.

Although this is build mainly on RISC-V ISA, I had to study a lot of other instruction sets and it's architectures its runs on before settling to this one. Also I've learned a lot about different approaches to the branching and to the load forwarding problem.

The future work should focus on other forms of a configuration of the processor, such as the mentioned speculative register file size, or implementing a slow mode, which would show sending the instructions as in Cisco's Packet Tracer [8]. Another feature would be a memory view to inspect the current state of all memory cells, where one cell would contain a specified number of bytes.

Bibliography

[1] Branch prediction using Selective Branch Inversion. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. 1999. DOI: 10.1109/pact.1999.807405. ISSN 1089795X.

[2] Albuquerque, N., Prakash, K., Mehra, A. and Gaur, N. Design and implementation of low power reservation station of a 32-bit DLX-RISC processor. In: *Proceedings - 2016 International Conference on Information Science, ICIS 2016*. 2017. DOI: 10.1109/INFOSCI.2016.7845330.

[3] Andrew Waterman, K. A. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA [online]*. CS Division, EECS Department, University of California, Berkeley: SiFive Inc., december 2019 [cit. 2021-05-07]. Available at: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[4] Baer, J.-L. *Microprocessor architecture : from simple pipelines to chip multiprocessors*. New York: Cambridge University Press, 2010. ISBN 978-0-521-76992-1.

[5] Castellanos, A. *Jupiter, RISC-V Assembler and Runtime Simulator* [online]. 2019 [cit. 2020-12-03]. Available at: https://github.com/andrescv/Jupiter.

[6] Co, M., Weikle, D. A. and Skadron, K. Evaluating Trace Cache Energy Efficiency. *ACM Transactions on Architecture and Code Optimization*. 2006, vol. 3, no. 4. DOI: 10.1145/1187976.1187980. ISSN 15443973.

[7] Faraz, A., Haque Zeya, F. U. and Kaleem, M. A Survey of Paradigms for Building and Designing Parallel Computing Machines. *Computer Science & Engineering: An International Journal*. 2015, vol. 5, no. 1. DOI: 10.5121/cseij.2015.5101. ISSN 22313583.

[8] Frezzo, D., Wang, M., Chen, M., Anderson, B., Hou, J. et al. *Cisco Packet Tracer*. Cisco Networking Academy, 2010.

[9] García Ordaz, J. R., Ramírez Salinas, M. A., Villa Vargas, L. A., Lozano, H. M. and Macías, C. P. A reorder buffer design for high performance processors. *Computacion y Sistemas*. 2012, vol. 16, no. 1. DOI: 10.13053/cys-16-1-1369. ISSN 14055546.

[10] Hayashi, T. and Kanasugi, A. A design of EPIC type processor based on MIPS architecture. *Artificial Life and Robotics*. 2020, vol. 25, no. 1. DOI: 10.1007/s10015-019-00554-w. ISSN 16147456.

[11] Higginson, P. *RISC Simulator by Peter Higginson* [online]. 2016 [cit. 2020-12-03]. Available at: https://www.gwegogledd.cymru/wp-content/uploads/2018/04/RISC-Simulator-Design.pdf.

[12] Kiat, W. P., Mok, K. M., Lee, W. K., Goh, H. G. and Andonovic, I. A comprehensive analysis on data hazard for RISC32 5-Stage pipeline processor. In: *Proceedings - 31st IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2017*. 2017. DOI: 10.1109/WAINA.2017.20.

[13] Kinsy, M. *A Browser-based RISC-V Simulator* [online]. 2019 [cit. 2020-12-03]. Available at: https://ascslab.org/research/briscv/simulator/files/tutorial.pdf.

[14] Kočí, K. *Graphicall CPU Simulator with Cache Visualization*. May 2018. Diploma thesis. Faculty of Electrical Engineering ČVUT in Prague.

[15] Miklós, D. S. Árpad. *VSIM - A Superscalar CPU Simulator*. H-1034 Budapest, Nagyszombat u. 19.: John von Neumann Faculty of Informatics, Budapest Polytechnic, 2001.

[16] Petersen, M. B. *Ripes Introduction* [online]. 2020 [cit. 2020-12-03]. Available at: https://github.com/mortbopet/Ripes/wiki/Ripes-Introduction.

[17] Shrivastav, S., Kumar, S., Gupta, S. and Bhushan, B. Qualitative Analysis of 32 Bit MIPS Pipelined Processor [online]. *International Journal of Engineering Research and*. 2020-05-01 [cit. 2021-05-07], V9, no. 05, p. 558–561. DOI: 10.17577/IJERTV9IS050484. ISSN 2278-0181. Available at: https://www.ijert.org/qualitative-analysis-of-32-bit-mips-pipelined-processor.

[18] Stefan Metzlaff, N. K.-L. *DLXMIPS processor simulator* [online]. University of Augsburg, 2013 [cit. 2020-12-03]. Available at: https://github.com/smetzlaff/openDLX.

[19] Václav Dvořák, V. D. *Architektura procesorů* [online]. 2006 [cit. 2020-01-04]. Available at: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FACH-IT%2Ftexts%2F2008%2FACH-cast1.pdf&cid=13577.

[20] Wang, C., Li, X., Zhang, J., Zhou, X. and Nie, X. MP-Tomasulo. *ACM Transactions on Architecture and Code Optimization*. 2013, vol. 10, no. 2. DOI: 10.1145/2459316.2459320. ISSN 1544-3566.

[21] Zhao, J., Korpan, B., Gonzalez, A. and Asanovic, K. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine [online]. *Fourth Workshop on Computer Architecture Research with RISC-V*. May 2020 [cit. 2021-05-07]. Available at: https://github.com/riscv-boom/riscv-boom.

Appendix A

Contents the included storage media

The DVD disc contains following folders:

- `./Misc/Mockups` - The mock-ups of the early version of the app
- `./Source` - The folder contains the source codes for the app itself. It also contains doxygen file for generating source code documentation
- `./Thesis` - The root folder for the LaTeXfiles used to generate this report
- `./Application`[1] - The application itself with the README file containing instructions on how to run the application

[1]The current version is also available on https://nextcloud.fit.vutbr.cz/s/eWSxejPno3pxnQW

Appendix B

User manual

This chapter explains how to install, use, and configure the RISC-V simulator.

Installing dependencies

Linux

You need to have JDK 15 or higher installed on your system. If you don't have JDK 15 installed, follow the steps below:

Ubuntu

You will need at least `openjdk-15-jre` and `openjdk-15-jdk` packages. To install then use the following commands:

```
sudo apt-get install openjdk-15-jre
sudo apt-get install openjdk-15-jdk
```

Usually, after installing these packages they should be set as default. To check the set version, use:

```
java --version
```

If not, use `update-alternatives` to change the default version of java.

```
sudo update-alternatives --config java
```

Arch Linux/Manjaro

To install required packages, use the following command:

```
sudo pacman -S jre15-openjdk-headless jre15-openjdk
sudo pacman -S jdk15-openjdk openjdk15-doc openjdk15-src
```

After that, you need to set the default JDK using the following command:

```
sudo archlinux-java set java-15-openjdk
```

To check which version is active you can use `archlinux-java status` or `java --version`.

All versions

After installing the required packages, you need to run the `run.sh` script. Most of the time, you will need to set up permission to be able to run this script. Use `chmod +x run.sh` to make it runnable. After that, you'll be able to run the application.

Windows

Download and install the Java SE Development Kit 16.0.1 from the official Oracle site: https://www.oracle.com/java/technologies/javase-jdk16-downloads.html and after that, run the 'run.bat' file.

Application window

The simulator consists of two main windows that are linked to the control tabs on the right side of the screen, being the Simulation and Code windows. The Simulation window can be seen at the start of the application. It is used for controlling and the visualization of the simulation. The Code window serves as an input for user-made source code, following the ISA loaded by the application. There is also the main menu on the top of the screen where users can configure the simulator properties, load existing examples, and find out which instructions got loaded.

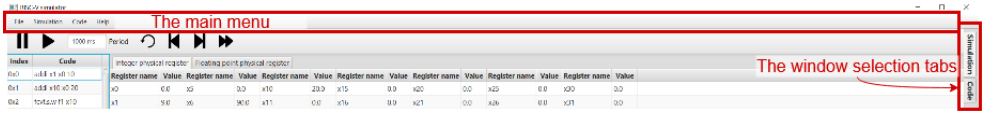

Figure B.1: The main application controls

Code window

The Code window is used for entering source code and loading it into the simulation logic. For entering the source code, please use the Code text area in the middle. The code window menu is located at the top of the window, providing controls for file manipulation and compiling the source code. At the bottom, there is a console output, for the result of the compilation, which on success, loads the code into the simulation.

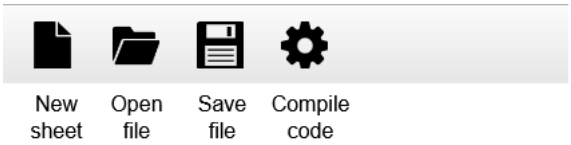

Figure B.2: The Code control buttons

The window button functions are from the left as follows:

- New - Opens a new sheet for the source code
- Open - Opens a file dialogue to selected the file to be loaded into the code window. If the selected sheet is empty, it will load into the selected one. Otherwise, a new sheet with file contents will be opened.

- Save as/Save - The selected sheet will be saved. If not saved before, a file dialogue opens, expecting the user to specify the path and filename. If the sheet has been saved before, changes will be saved into that file.
- Compile - Takes the selected sheet and tries to load it into the simulation. If an error arises, the output is written into the Console window.

Figure B.3: The Code window

The Code window sheets support basic highlighting, where the style is as follows:

- Green - Represents the instructions loaded into the simulator.
- Red - Represents the registers loaded into the simulator.
- Blue - Represents numerical values.

- Orange - Represents labels. List of acceptable labels if formed dynamically based on the source code
- Grey - Represents comments

The list of allowed instructions and their syntax can be found inside the *Help->Instruction list* in the main menu.

Simulation window

In the simulation window, users can simulate their code after a successful compilation. If the user is inside the Code window during the compilation, a dialogue will ask him, if they want to move to this window. Inside the window, the user can find:

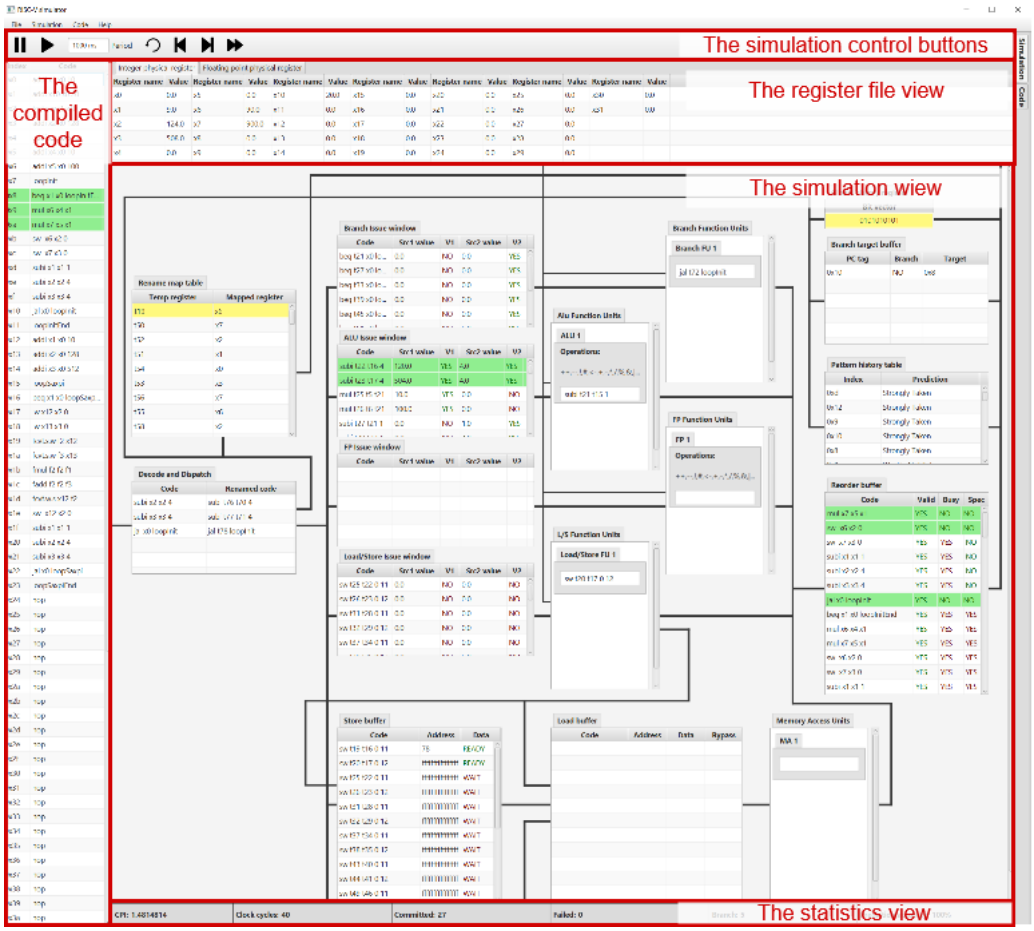

Figure B.4: The Simulation window

- Control menu - Placed on the very top of the window, with controls to the simulation
- Register file window - Situated on the top of the screen, displaying all loaded registers.
- Code window - Situated on the right, showing the compiled code.
- Simulation view - Showing all the blocks of the simulation

- Statistics - At the bottom, showing the gathered statistics from the run

The control menu button functions are from the left as follows:

- Pause - Stops the periodic execution.
- Play - Steps the simulation at fixed intervals. The period can be changed in the Period text field located also in the menu
- Reset - Resets the simulation to the initial state
- Previous step - Simulator will take one step back in the simulation. If the simulation is in the initial state, nothing happens.
- Next step - Simulator will take one step forward in the simulation.
- Fast-Forward - Simulator will simulate the whole input source code and shows the result. (WARNING: may take some time)

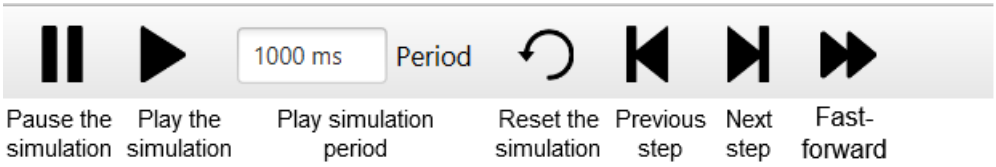

Figure B.5: The Simulation control buttons

As soon as the simulation reaches the end, the user is notified using the dialogue window.

Basic configuration

The configuration inside the simulator can be accessed by locating the *Simulation->Configuration* in the main menu. A dialogue will pop out showing multiple tabs featuring different configuration options.

- Loader - Configuration of the locations of the register file and ISA folders
- Buffers - Configuration for buffer sizes
- Function Unit - Configuration of function units for different issue windows
- Branching - Configuration of the branch units
- Fetch and Commit - Configuration of the amount of instruction fetched/committed in one cycle

The function unit configuration tab offers additional configuration options. The users can create their own function unit or edit the existing ones here. The function unit configuration tab contains additional tabs for each issue window plus the load buffer. Each sub-tab has a table, showing all the existing function units and at the bottom options for modifying the table. The options are:

- Add - A pop-up window will show up, with a form for creating a new function unit. The field delay is required.
- Edit - A pop-up window will show up, with a filled form from the selected function unit.
- Remove - Removes all selected field in the table

Each of the pop-up windows has a Confirm button and a Cancel button. The Confirm button either adds new correctly configured unit into the list or edits selected function unit with new values. The Cancel button closes the window without committing the changes. The ALU and Floating-Point function units have one extra field in the creation pop-up window, where operators can be specified. The list of allowed binary operators is as follows:

- Add (+)
- Subtract (-)
- Multiply (*)
- Divide (/)
- Modulo (+)
- AND (&)
- OR (|)
- Arithmetic shift right (»>)
- Logical shift right (»)
- Logical shift left («)
- Less than or equal (<=)
- Greater than or equal (>=)
- Equal (==)
- Less than (<)
- Greater than (>)

List of allowed unary operators is as follows:

- Increment (++)
- Decrement (–)
- NOT (!)
- Squared (#)
- Assign (<-)

The comparator operators are used only for the ALU function unit, while they return 1 if the condition is true or 0 if it is false. The assign operator is mainly used for the convert functions between integer and floating-point values.

When the user wants to confirm his changes, there is a confirm button inside the configuration window, which will reflect the changes and reset the simulation. If the user wants to exit without confirming the changes, there is a cancel button at the bottom of the window.

Advanced configurations

Apart from the configurations done inside the simulation, users can extend, modify, or change completely the register files and instructions used inside the application. The default configuration files can be found in `./riscisa` and `./registers` folder. These folders are free to be configured or copied to create different configurations. The configuration files are written in JSON with a fixed structure of the object names, which every file in a certain category must follow.

Register file

The JSON of the register file structure can be seen in listing B.1. Each register file object must have the `name`, `dataType`, and `registerList`. The `name` specifies the display name that is used in the Register file window. The `dataType` is used for comparing with the instruction argument data types during the compilation. The data type can be either integer (`kInt`), long integer (`kLong`), float (`kFloat`), or double (`kDouble`).

```
{
  "name": "Integer physical register",
  "dataType": "kInt",
  "registerList": [
    {"name": "x0", "isConstant": true,"value": 0},
    {"name": "x1", "isConstant": false,"value": 0},
    {"name": "x2", "isConstant": false,"value": 0},
  ]
}
```

Listing B.1: Register file example

The register list must be composed of one or more register objects. The Register object can be seen in listing B.2. It has 3 objects, a `name`, an `isConstant` flag, and a `value`. The `name` is used to address the specified register in the source code, and the user can also see it in the Register file window. The flag `isConstant` is telling the simulator, whether the register value is read-only (flag set to true) or read/write (flag set to true). In the `value`, the user can specify the initial value for a certain register.

```
{
    "name": "x1",
    "isConstant": false,
    "value": 0
}
```

Listing B.2: Single register example

Instruction

The instruction is also a JSON object. The structure can be seen in listing B.3.

```
{
  "name": "add",
  "instructionType": "kArithmetic",
  "inputDataType": "kInt",
  "outputDataType": "kInt",
  "instructionSyntax": "add rd rs1 rs2",
  "interpretableAs": "rd=rs1+rs2;"
}
```

Listing B.3: Instruction example

The name is used to address certain instructions when writing the source code. The `instructionType` is to specify the type of the instruction. The simulator uses three types, being arithmetic (`kArithmetic`), load

The `inputDataType` is specifying the data type of input arguments. It is used to validate the data type of an argument if it is the register or used to cast the argument if it is an immediate value. The `outputDataType` is used to specify the data type of the output register argument.

The `instructionSyntax` is telling the parser the syntax of the instruction with the types of each argument. The arguments are either destination register (`rd`), source register (`rsX`, where `X` is a natural number), or an immediate value (`immX`, where `X` is a natural number). Lastly, the interpretableAs is used to tell the interpreter how arguments should be processed.

The `interpretableAs` field has different syntax depending on the type of instruction. Examples can be found in the `./riscisa` folder. The syntax uses the abbreviations used by the `instructionSyntax` to link arguments together. The syntax can be divided into the 3 categories.

Arithmetic expressions that are similar to the Java expression ($rd = rs1 - rs2 + +$, Increments the value inside the rs2). The arithmetic expressions allows brackets ($rd = (rs1 - rs2) * rs3$), and also indexing separate bits ($rd[5:0] = rs1[31:25]$), where the range needs to be specified in the "downto" fashion. See the FX and FP instructions in *Help->Instruction list* for more examples.

Branch instruction syntax has two versions: unconditional jump (by specifying `jump` in the `interpretableAs` field), or conditional jump, where the syntax is as follows: "`(unsigned|signed):`*compareExpression*". The first part of the conditional jump syntax tells the interpreter, whether the expression should be evaluated as signed or unsigned. The second part is the condition, where compare operators and either the register, the immediate value, or the numerical constants are allowed. An example of such a condition would be $rs1 == 0$. See the branch instructions in the *Help->Instruction list* for more examples.

The load/store instruction syntax is different depending, whether it is a store instruction or a load instruction. The load instruction syntax is "`load` *dataType*`:(signed|unsigned)` *what where offset*", where `load` literal specifies the load instruction, dataType specifies how many bytes are loaded from the memory (allowed values: `byte, half, word, doubleword` for the integer values, `float, double` for floating-point values). The signed or unsigned literals tells the interpreter, whether the loaded value should be signed or unsigned. The

what, *where* and *offset* can be either the register, the immediate value, or the numerical constant, where *what* is the value to be saved, *where* points to the certain place in the memory, and *offset* specifies the offset from the *where* value. The store instruction has similar syntax being "`store` *dataType what where offset*", where the `store` literal specifies the store instruction and other arguments has the same rules as in the load syntax.

Use cases

I want to run a simulation

1. Run the application
2. Move to the Code window using the tabs in the top right corner
3. Either write a new code or load the existing one
4. Press the Compile button (cog icon in the menu)
5. Confirm the pop-up dialogue
6. Use the buttons in the simulation window menu to control the simulation

I want to open the existing code

Expecting you are already running the application.

1. Move to the Code window using the tabs in top right corner
2. Click the open icon (the folder icon in the menu) or press CTRL+S combination
3. Select the file, which you want to load
4. Confirm the dialogue

I want to save my new code

Expecting you are already running the application.

1. Move to the Code window using the tabs in the top right corner
2. Click the save button (the floppy disk icon in the menu) or press CTRL+S combination
3. Enter the name of the file
4. Specify the path, where to store your source code
5. Confirm the dialogue

I want to save my existing code

Expecting you are already running the application.

1. Move to the Code window using the tabs in top right corner
2. Click the save button (the floppy disk icon in the menu) or press CTRL+S combination

I want to simulate at certain period

Expecting you are already running the application and code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. In the simulation menu, specify your period in the Period text field (supported units: s, ms)
3. Press the play button (the play icon in the simulation menu)
4. If simulation needs to be stopped, press either the play button or stop button (the pause icon)

I want to see each step of the simulation

Expecting you are already running the application and code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. Use the Previous and the Next step buttons to carefully observe the state in each step (the fourth and the fifth buttons in the Simulation window)

I want to see the result state of my source code

Expecting you are already running the application and the code has been compiled.

1. Move to the Simulation window using the tabs in the top right corner
2. Use the Fast-forward button to see the end state of the simulation (the sixth button in the Simulation window)

I want to edit the buffer sizes

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Buffers tab
4. Specify the desired sizes
5. Click the confirm button at the bottom of the configuration window

I want to edit the branch configurations

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Branching tab

4. Set the desired BTB size
5. Change the GHR and the PHT sizes
6. Change the predictor type in the selector
7. Change the initial state of all predictors
8. Click the confirm button at the bottom of the configuration window

I want to edit the commit or the fetch size

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Fetch and Commit tab
4. Change the fetch size and the number of ways text field
5. Change the commit size in the text field
6. Click the confirm button at the bottom of the configuration window

I want to add new function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs to which you want to add the new function unit
5. Press the Add button in the Function unit sub-tab
6. Set the Function unit name
7. If configuring ALU or FP units, write down allowed instruction split by the coma
8. Specify the delay of the unit
9. Create the unit by clicking Confirm button
10. To confirm your changes into the simulation, press Confirm button in the configuration dialog

I want to edit existing function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs in which you want to edit the function unit
5. Select the function unit from the table that you want to edit
6. Press the Edit button in the Function unit sub-tab
7. Edit the values in the forms
8. Press the Confirm button
9. When done editing, pres the confirm button in the configuration window

I want to delete function unit

Expecting you are already running the application.

1. Navigate to the very top menu
2. Go to *Simulation->Configuration*
3. In the configuration select the Function unit tab
4. Select the function unit list from the sub-tabs in which you want to delete the function unit
5. Select the function unit from the table that you want to delete
6. Press the Delete button in the Function unit sub-tab
7. When done deleting the function units, pres the confirm button in the configuration window