Microarchitectural Design and Implementation of Elements of an Out-of-Order RISC-V Processor

(Projekt mikroarchitektury i implementacja elementów procesora RISC-V Out-of-Order)

Krzysztof Obłonczek

Praca inżynierska

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

12 czerwca 2024

Abstract

Modern processors can execute instructions in a different order to what was written by the programmer or the compiler, but they still maintain the illusion of executing them in program order. They do this using hardware data structures that construct a graph of data dependencies between instructions and schedule a particular instruction for execution as soon as the hardware resources and operands required for it are available. This technique called "out-of-order" execution has been ubiquitous in processors ever since the second half of the 90s as it unlocks the potential for great performance gains.

In this work we explain inner workings of such processor and present the microarchitecture of the *Coreblocks* core – an out-of-order RISC-V core that is actively being developed at the University of Wrocław by a team comprising almost exclusively of students. Parts of the core written by the author (former member of the development team) are described in more detail, with particular emphasis on the design decisions made and problems encountered during the implementation.

Współczesne procesory potrafią wykonywać instrukcje w innej kolejności niż zapisał je programista lub kompilator, ale wciąż zachowują iluzję wykonywania ich w porządku programu. Robią to przy pomocy sprzętowych struktur danych konstruujących graf zależności obliczeń między instrukcjami, zlecając wykonanie danej instrukcji tak szybko jak tylko dostępne są zasoby sprzętowe i gotowe są operandy potrzebne do jej wykonania. Technika ta zwana „out-of-order execution" jest stosowana w procesorach praktycznie wszechobecnie od połowy lat 90. ponieważ odblokowuje potencjalnie duże zyski w wydajności.

W tej pracy przybliżamy zasadę działania takiego procesora i przedstawiamy mikroarchitekturę rdzenia *Coreblocks* – rdzenia out-of-order RISC-V aktywnie rozwijanego na Uniwersytecie Wrocławskim przez zespół złożony prawie wyłącznie ze studentów. Elementy rdzenia napisane przez autora (byłego członka zespołu deweloperskiego) zostały opisane w większych szczegółach, ze szczególnym uwzględnieniem podjętych decyzji projektowych i problemów napotkanych podczas implementacji.

In loving memory of Krystyna Tyka and Jan Panné

Contents

Chapter 1

Introduction

Traditionally we think of the processor as a black box that takes a stream of instructions and executes them sequentially. While this has been true in consumer-grade CPU implementations in the 1970s and 1980s, there is an inherent weakness in this approach. Consider this example assembly code for an abstract ISA (result is placed in the first operand of an instruction):

```
load r1 ← mem[r2]
add r3, r1
mul r4, r5
```

The `add` instruction is dependent on the result of a previous `load` because it uses `r1` register as one of its operands, while later `mul` instruction isn't because it doesn't depend on either `r1` or `r3`. To understand why this code would pose a problem to any in-order processor aspiring to be high-performance we need to look at another major component in any computer system – memory.

Speed of processors and memories have been steadily improving over the past decades, but their pace hasn't been the same. CPUs started outperforming memory ever since the 1980s and this gap has been widening ever since, roughly 50% per year.

Figure 1.1 details the evolution of CPU and memory speed starting from the year 1980 as a baseline. In 2010 processors have been three orders of magnitude faster than memory and this trend is only going to continue. Therefore the issue with our in-order CPU is as follows: every `load` instruction needs to send the request for data to memory, stall (pause) the core and wait potentially hundreds of clock cycles before the memory responds with data. No useful work is performed during this wait period, but if we could somehow deduce that a subsequent `mul` later in the instruction stream is not dependent on the result of such `load` (or any instructions that depend on it) we could execute it *out of order* and perform useful work during otherwise wasted cycles. Solving this problem lies at the heart of out-of-order paradigm in

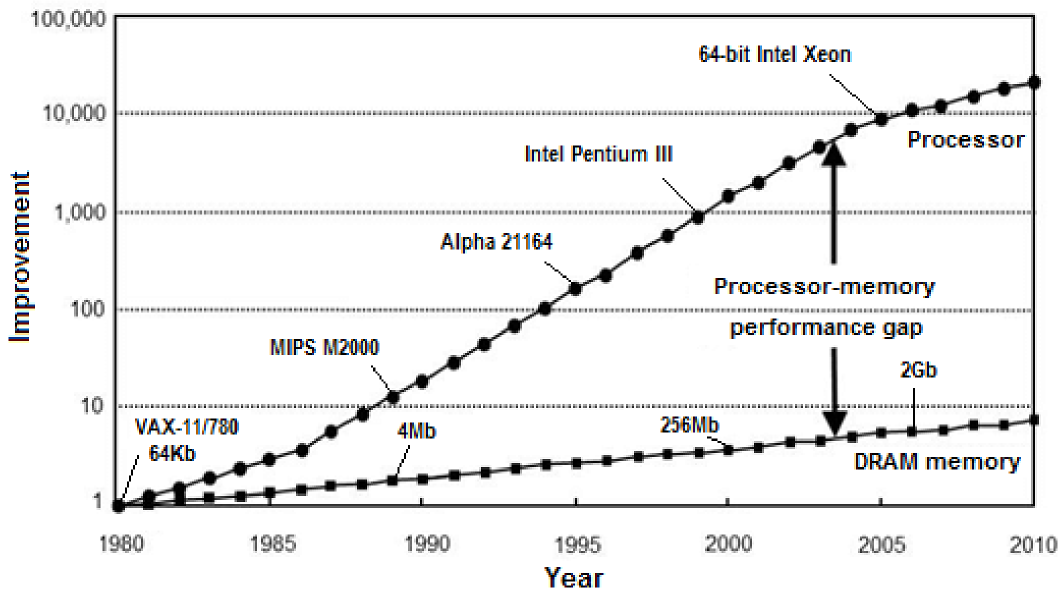

Figure 1.1: Memory and processor speed improvements [4]

processor design and, once done, enables more optimizations to be applied for even more performance gains.

In this work the classic approach for implementing out-of-order execution is outlined and a concrete implementation created at the “Kuźnia rdzeni” project at University of Wrocław is presented in detail. The project has successfully produced a working RISC-V core and the development team is working towards implementing the necessary hardware to run Linux and the Mimiker operating system also developed at University of Wrocław that was recently ported to RISC-V [2]. Parts of the implementation written by the author of this work are discussed in greater detail, with focus on the design decisions made and problems that arose during the implementation, providing insights into the development of such hardware.

Chapter 2

Out-of-order paradigm

2.1 Dataflow

A sequence of instructions that operate purely on data registers can be described as a directed graph where nodes perform the operations and results flow along the edges [3]. Example instruction stream and its corresponding dataflow graph are shown in figure 2.1

```
div r4, r2
add r1, r2
add r1, r3
mul r1, r4
```

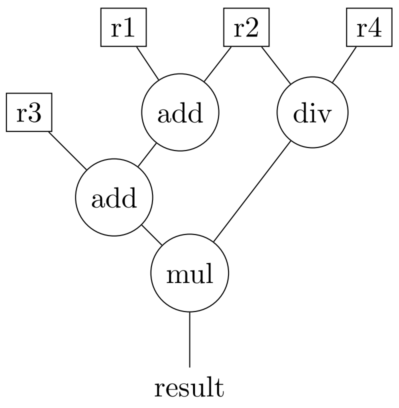

Figure 2.1: Assembly code and its corresponding dataflow graph

Conceptually, registers are supplied as inputs from the top, they travel along the edges and are used as operands to nodes labeled with specific calculation performed on them. Result is propagated downward to be used as input by further nodes (or is a final result).

Dynamically building fragments of this graph in hardware would allow us to infer e.g. that `add` instructions are independent from the (costly in terms of clock cycles) `div` instruction, so they can be executed in parallel to it. We could also infer that we need to delay the execution of the `mul` instruction until both of its operands become available. A mechanism for notifying the component holding the data about `mul` that this happened is also necessary so that it can be scheduled for execution.

2.2 Register renaming

First problem to tackle is: how to disassociate register names from actual values needed for computing the result, i.e. how to correctly associate that the `mul` instruction needs the value computed by the second `add` and not the first, even though both `adds` specify `r1` as their target?

Register renaming solves this problem by assigning a *tag* to the result of an instruction. As instruction comes in, an unused tag is assigned to it and tags of its operand registers are looked up in a *Register Alias Table (RAT)*. Architectural register IDs are replaced with those tags, old tag in RAT for the result register is replaced with the newly assigned one and the instruction proceeds further down the pipeline. This process is illustrated in figure 2.2.

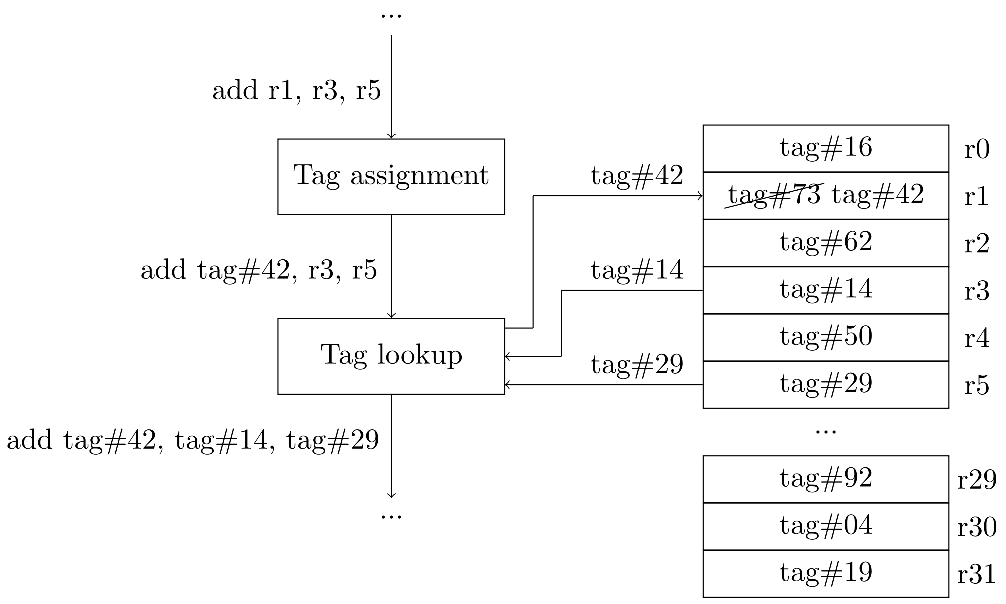

Figure 2.2: Example of register renaming

2.3 Tomasulo's algorithm

Before an instruction executes, it's placed in a *Reservation Station* that sits just before an execution unit. Reservation station is hardware block where an instruction waits for the availability of an execution unit that could execute it and the availability of its operands. Once the execution unit for a particular instruction is free and its operands ready, it might be selected for execution on that execution unit. When the computation is finished, the result and this particular instruction's tag are broadcasted on the *Common Data Bus*.

Initially, an instruction might not have all of its operands ready. Reservation stations constantly listen for broadcasts on the common data bus and compare the tag associated with the result to the tags of all operands of all waiting instructions. If any of them matches, the tag in the reservation station is replaced with the actual value of the operand. Once both of the operands have their values the instruction is ready to execute. This is illustrated in figure 2.3.

One important detail that was glossed over is – what if operand of an instruction was broadcasted *before* it entered a reservation station? The window for acquiring the value was missed so the instruction may never execute. This is usually solved by including an array of storage locations that are indexed by the tag – instruction that produced the result writes to a location associated with the tag of that result. Later, instructions that missed the broadcast of their operands can get the value that was broadcasted from the associated storage locations, provided they were already filled in. There are a few more details to take care of that will be presented in chapter 4.

Register renaming, reservation stations and common data bus together with the process described above form the basis of Tomasulo's algorithm [26].

Recall the assembly sequence from the introduction chapter:

```
load r1 ← mem[r2]
add r3, r1
mul r4, r5
```

With the hardware that was presented it's now possible to execute this instruction stream out-of-order:

1. `load` enters the core, register renaming is performed on it and enters a reservation station. Assuming `r2`'s value was already computed it starts executing right away (but it will take it a long time to fetch data from memory).
2. `add` enters the core, register renaming is performed on it and it enters a reservation station. Because `r1`'s value hasn't been computed, it can't execute yet.
3. `mul` enters the core, register renaming is performed on it and it enters a reservation station. Assuming `r4`'s and `r5`'s values have been computed, it can start executing right away.
4. `mul` finishes execution before `load` (and `add` dependent on its result).
5. `load` finally fetches data from memory and finishes execution. `add` proceeds with its execution and finishes as well.

This way we've executed `mul` before `load` has completed while maintaining correctness. This however is only one piece of the puzzle – there are some issues with this approach that we shall outline in the following section.

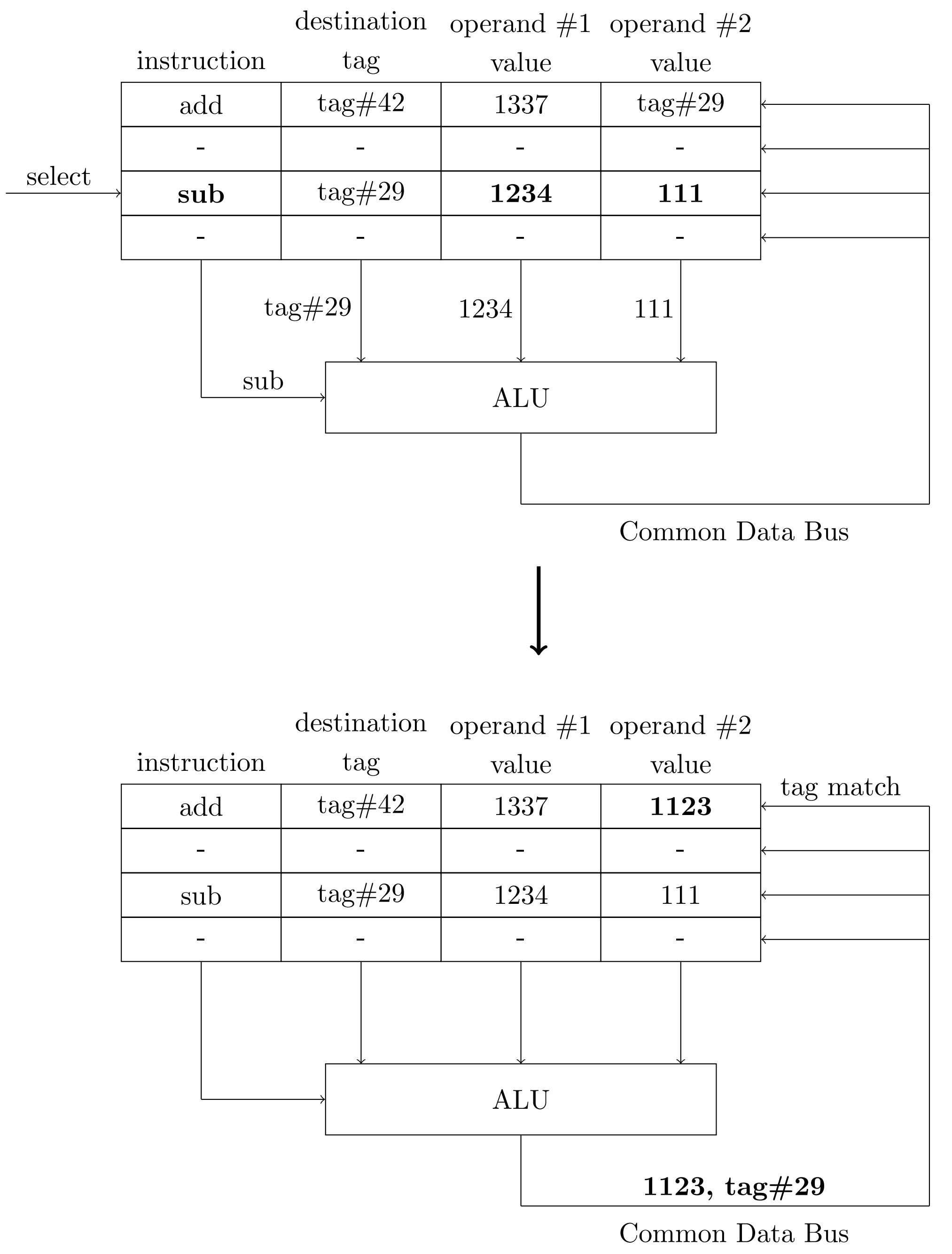

Figure 2.3: Reservation station selecting instruction for execution and updating entries based on announced tags

2.4 Tracking instruction order

So far we've been executing instructions with the assumption that they will always succeed and produce a result. In practice there are situations where this might not always be true (called exceptions). For example, an instruction that accesses memory can fail because the address it accesses is outside of the allowed range. In that case it is customary for processor implementations to stop execution at this particular instruction and jump to a function (called exception handler) dedicated to handling such situations. Once the handler returns, controls is given back to the program that failed, in hopes that the handler "fixed" the execution environment of the program and that it can proceed further as normal.

Slight problem with this is that some instructions that came in later after the offending instruction might have already executed. If we jumped to an exception handler and came back from it later to the offending instruction we might in effect execute some instructions that come after it twice. The core problem is that we don't have strict control over *when* an instruction performs changes to the core's state – once instruction has executed there is no way to revert its changes to the register alias table. But even if we were able to do that there's a second problem – we don't know which instructions' changes to revert since we don't track the order they came into the processor.

To remedy the second problem we will introduce a *Reorder Buffer* (ROB) – a data structure that tracks the order of instructions and state of their execution (whether they have finished or encountered an exception). ROB is usually implemented as a queue (and it's useful to think about it that way) that holds instructions with the ability to modify their metadata (e.g. completion status) without respecting the queue order (figure 2.4). As instructions come into the processor they're enqueued in the ROB and as they finish execution or encounter an exception their state in the ROB is updated accordingly. Once processing them has fully finished (note that this is not synonymous with finishing execution, as will be shown shortly), they're dequeued from the ROB. Knowing their order, once we solve the first problem in the next section, we will be able to revert precisely those that came after the excepting instruction.

2.5 Controlling state changes

To solve our problem we'll introduce a second RAT and a controlled way of altering state of the core by instructions. The RAT that was originally introduced is usually called frontend RAT, and the newly introduced one is called retirement RAT. Much like the frontend RAT, it stores a mapping from architectural register IDs to tags. This time however it's not used for register renaming.

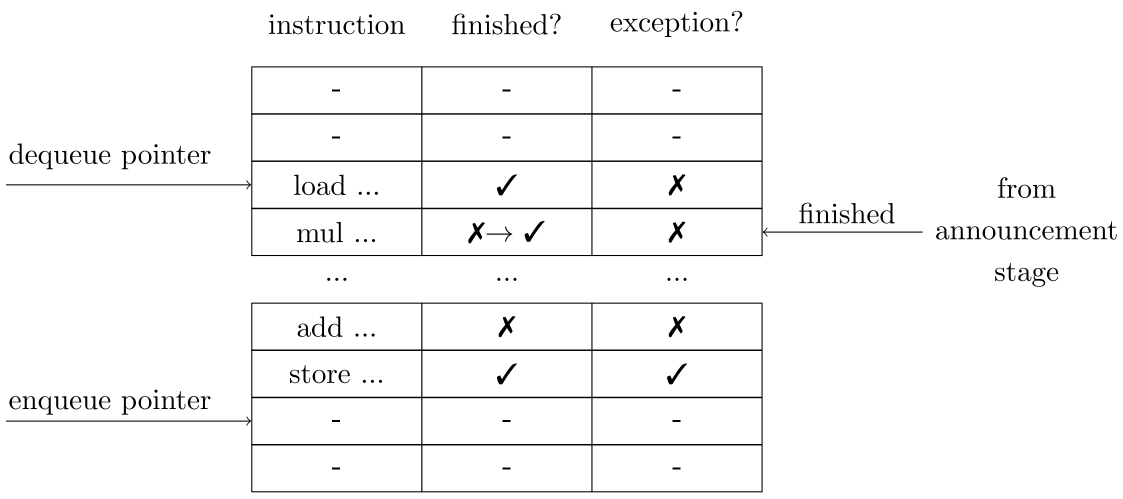

Figure 2.4: Reorder buffer

In it, a given entry for an architectural register stores a tag (that points to a storage location associated with it) for the result of the most recently *committed* instruction that had this target register. *Committed* in this context means: acknowledged that it completed and its state change applied to the core. This state change is in fact (almost) only the tag replacement itself. In the usual case when an instruction produces a result, this is written into a storage location corresponding to its assigned tag first, and once the instruction is committed this tag is written into retirement RAT (figure 2.5)

Crucial feature of committing is that it happens in-order – instructions are committed in the order they entered the core – by leveraging the ROB to keep track of their order. More precisely, to commit an instructions two conditions must be satisfied:

1. all previous instructions have been committed,
2. the instruction has finished execution.

Since no state changes are performed for instructions that haven't been committed yet, ROB and retirement RAT together precisely describe the architectural state (i.e. values of all architectural registers) of the core – meaning we can answer questions such as:

- What was the last instruction that has executed with respect to the program order?
- Given the above, what's the value of register `X` at this moment of program execution?

This solves our initial problem – now when it comes time to commit an instruction which has encountered an exception (this information is typically stored

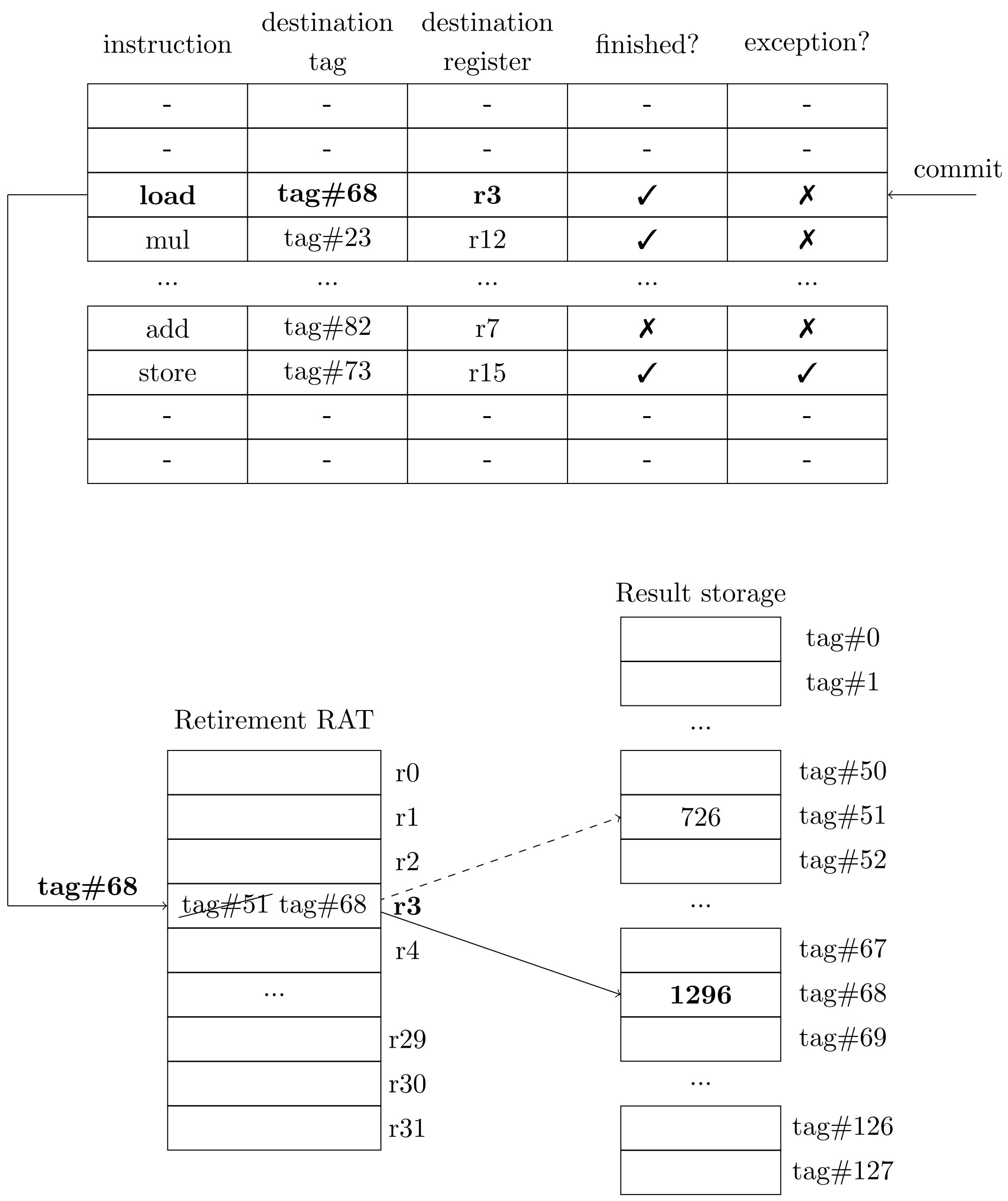

Figure 2.5: Instruction commit

in the ROB and only acted upon at commit stage) we don't have to worry about some other instructions that might've already altered the core's state because their state changes were not committed. Results of instructions following the excepting instruction can just be discarded and a jump to exception handler can proceed normally. When it returns the core jumps back to a program in a well-defined state – one in which only instructions up to the excepting instructions truly executed and committed their results.

2.6 Maximizing performance

Previous sections have outlined the motivation for out-of-order execution, classic method of implementing it and correctness problems that arise together with a method for solving them. All of this hard work would be in vain if we can't keep the core busy all the time – what's the point of being able to execute instructions out-of-order if there are too little of them in the core at any given time to benefit from it? This section describes some techniques for dealing with this problem and, in turn, maximizing the performance.

First solution that comes to mind is introducing caches – having data and instructions closer to where they're needed so they can be accessed faster.

But caches can only get us so far, so in addition another approach is scaling up the processes outlined previously. If we could perform fetch, register renaming, instruction execution and other steps on N multiple instructions at once, we could theoretically improve our performance by up to this N multiplicative factor. This comes with caveat that now complicated logic is needed to check for inter-instruction dependencies. Processors that use this approach are called *superscalar*.

There are however still situations when the core could be underutilized. Suppose that the core encounters a branch instruction. Its direction (whether it will be taken or not) and target address is unknown until this is calculated when the instruction executes, thus the core does not know where to fetch instructions from at that point. The simplest solution is to halt fetching of new instructions when such instruction is encountered and wait until the branch target address is resolved. This is inefficient in terms of resource usage – much like in the example in section 1 we are wasting cycles where otherwise useful work could be performed.

Speculation is a technique in processors for performing work ahead of time before it's known to be needed. One example of speculation – *speculative execution* – has already been presented: when instructions are executed without knowledge that some preceding instructions have succeeded – it's optimistically assumed that they will succeed (but there are mechanisms to backtrack if that assumption turns out to be wrong).

We can leverage this mechanism by speculating not only on instruction's successful execution, but also on the direction that a branch instruction will take. Based on the history of previous executions of a particular branch being taken or not we start executing instructions from a given path before the branch direction is calculated (this is also *speculative execution*, but this time it's optimistically assumed that the branch went in a certain direction). Once that's done, if the prediction turned out to be right, we've successfully performed useful work during otherwise wasted cycles. If the prediction was wrong however, we need to backtrack from the wrong path and restart execution from the correct path. This is handled similarly to exception handling described in the previous section.

Results of instructions executed after incorrect prediction are thrown out so in the end we performed work that was useless – we could've been better off if we just waited for the branch to finish without speculating about its result. In practice with a good *branch predictor* – a component that tries to accurately predict what the next outcome of a specific branch will be – the benefits of correct prediction outweigh the costs of an incorrect one. Modern branch predictors like TAGE-SC-L [24] achieve around 2.5 mispredictions per 1000 instructions because most branching patterns in programs (e.g. loops) are quite predictable.

A different example of speculation is *prefetching* – detecting memory access patterns in a program dynamically and preemptively fetching memory locations that will most likely be needed soon. For example accessing an array at consecutive indices in a loop is a common access pattern in programs that *prefetchers* exploit by fetching memory locations corresponding to indices in future iterations of the loop ahead of time, even before instructions for these memory accesses are scheduled for execution.

There are many more speculation examples – e.g. value prediction [11], pointer address prediction [14] and more approaches for maximizing performance in general but they are out of scope for this work and are better described in literature.

Chapter 3

The Coreblocks project

3.1 RISC-V

Instruction Set Architecture or *ISA* is a specification that bridges software and hardware – describing what hardware is able to do and how the programmer can interact with it. In particular the specification describes [12]:

- data types supported by the processor,
- set of opcodes and registers available for the programmer,
- virtual memory support,
- memory model,
- input/output model.

RISC-V ISA [8] is a RISC load-store architecture. One of its notable features is the conscious design decision to separate non-essential groups of instructions that perform related tasks into *extensions*. Base ISA defines a minimal reasonable set of instructions required for a processor to be usable in practice by programmers, but the set of available opcodes can be optionally extended. As an example, **M** extension could be implemented by the hardware designer to permit use of dedicated hardware multiplication and division instructions that aren't present in the base ISA.

RISC-V is an *open ISA*, meaning it is free of licensing limitations that often encumber other ISAs such as x86_64 or ARM, where a potential implementer of an ISA first needs to negotiate a licensing agreement between Intel or Arm Ltd. that will allow them to implement the ISA legally. In contrast, RISC-V is licensed under very permissive CC-BY-4.0 license [8, 9] which has allowed it to flourish, with many open source implementations being available under similarly permissive licenses together with freely available teaching materials and open-source tools [23].

3.2 Amaranth

Amaranth is an open-source toolchain for hardware development that consist of (mainly) the Amaranth hardware description language and the Amaranth simulator. **Amaranth HDL** is a Python-based DSL for describing digital logic, and will be referred to further as just *Amaranth* for simplicity. **Amaranth simulator** allows to simulate and test the design from within Python as well.

The language itself is embedded in Python and therefore all Amaranth programs are also valid Python programs. Amaranth however presents the users its own special syntactic idioms that use Python classes with a specialized interface that is aimed at describing hardware. Programming in Amaranth follows a metaprogramming paradigm – the programmer tells the language what Verilog statements (i.e. hardware) to generate and usually a one-to-one correspondence between Amaranth and Verilog code can be established. The largest benefit however comes from being able to leverage all of Python's programming constructs to build high-level abstractions that generate hardware – something that is very cumbersome or isn't possible at all in some cases in Verilog or SystemVerilog. Examples of such abstractions are described in section 3.4.

To better illustrate the language itself, consider sample Amaranth code that showcases its syntax and a corresponding (handwritten) Verilog code presented in figure 3.1. Notable differences in Amaranth with respect to Verilog are:

- Lack of explicit clock and reset signals – they are managed automatically so the programmer doesn't need to pass them between modules and manually initialize all signals and registers to 0.

- Lack of distinction between "register" and "wire" types – everything is declared as a `Signal` and depending on whether it's driven combinationally or synchronously it is treated either as a wire or a register (not to be mistaken with Verilog's `reg` keyword, which is used to declare both registers and signals driven combinationally):

  - to drive a signal combinationally, an assignment to it is added to a list of statements in combinational domain `m.d.comb`,
  - to drive a signal sequentially, the same is done with a different domain `m.d.sync`.

- Use of context manager syntax – the `with` keyword. Amaranth leverages this syntax to implement usual control structures present in programming languages, e.g. if/else/elseif or switch/case statements that act on values of `Signals` (or expressions involving them), meaning appropriate hardware is generated to perform this logic.

```python
m = Module()


cnt = Signal(8)
enable = Signal()
load = Signal()
load_val = Signal(8)
zero = Signal()





with m.If(load):
    m.d.sync += cnt.eq(load_val)
with m.Elif(enable):
    m.d.sync += cnt.eq(cnt + 1)


m.d.comb += zero.eq(cnt == 0)
```

Listing 1: Amaranth code

```verilog
input wire clk;
input wire rst;

output reg[7:0] cnt;
input wire enable;
input wire load;
input wire[7:0] load_val;
output reg zero;

always @(posedge clk) begin
    if (rst)
        cnt <= 0;
    else if (load)
        cnt <= load_val;
    else if (enable)
        cnt <= cnt + 1;
end

assign zero = cnt == 0;
```

Listing 2: Verilog code

Figure 3.1: Functionally identical Amaranth and Verilog code

A more comprehensive overview of language features can be found in the language's documentation [22].

3.3 Coreblocks

Coreblocks [17] is an academic project at the University of Wrocław where students partake in implementing an out-of-order RISC-V core generator, meaning that it can generate a whole range of CPU cores with different sets of RISC-V extensions and microarchitectural parameters, e.g. number of functional units or number of reorder buffer entries. It is one out of only a handful of other such projects [16, 6, 7, 1, 27] developed at universities.

Project is implemented in Amaranth hardware description language. Its use lowers the bar of entry for newcomers (as almost everyone already knows Python) and allows implementing abstractions for greater expressiveness compared to traditional HDLs like Verilog or SystemVerilog. To that end, a custom abstraction that allows for interfacing between components of the core called *Transactron* was developed, which will be described in more detail in the next section.

Directly citing the project's readme [18], it lists 3 main design goals:

- Simplicity. Coreblocks is an academic project, accessible to students. It should be suitable for teaching essentials of out-of-order architectures.
- Modularity. We want to be able to easily experiment with the core by adding, replacing and modifying modules without changing the source too much. For this goal, we designed a transaction system inspired by Bluespec.
- Fine-grained testing. Outside of the integration tests for the full core, modules are tested individually. This is to support an agile style of development.

At the time of writing Coreblocks supports generating cores with base **RV32I** ISA, **M** (multiplication and division), **C** (compressed instructions) and **B** (bit manipulation) RISC-V instruction set extensions.

Workflow used in the project is as follows:

1. A student that wants to contribute to the project first chooses a task that they are going to tackle. This doesn't necessarily mean working on the core implementation itself, as there's often work to be done on improving the documentation, continuous integration (used to run automated tests and generate documentation), the Transactron framework or software-oriented abstractions.
2. Optionally for larger features a discussion with the rest of the development team follows at a weekly meeting before implementation phase begins.
3. After initial implementation (that has to include automated tests for new features) has been completed on a separate git branch, it's submitted as a pull request on the project's GitHub page and is reviewed by members of the development team, who leave suggestions on what should be improved or ask questions about the implementation.
4. Implementation is adjusted according to suggestions. At this point more discussion might happen on weekly meetings until all comments are addressed and pull request approved by at least 2 reviewers.
5. The branch is merged into the `master` branch.

3.4 Transactron

Transactron is a library for the Amaranth language that was developed alongside Coreblocks to help unify the interfaces between different parts of the system and make hardware blocks cycle-independent of each other. It provides a set of classes that can generate complex scheduling logic automatically.

Two basic entities in the library are *methods* and *transactions*. Transactions are pieces of hardware that want to perform their computation on every clock cycle. They are atomic, in that they either execute fully or not at all. Methods are pieces of hardware that can be *called* by transactions. *Calling* a method means activating a hardware block to perform some computation. Much like methods in traditional programming languages, methods in Transactron can take arguments, call other methods, return results and modify internal state of the hardware block that they're a part of. Transaction can only execute if all the methods that it wants to call are available, i.e. they signal that they're ready and they're not being used by other transactions.

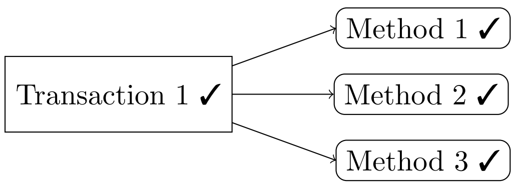

Figure 3.2: When all methods are ready the transaction can execute

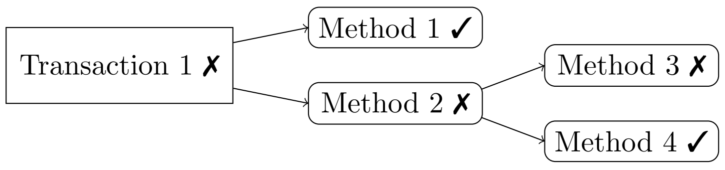

Figure 3.3: Methods can also call other methods, but any transitively not ready method prevents the transaction from executing

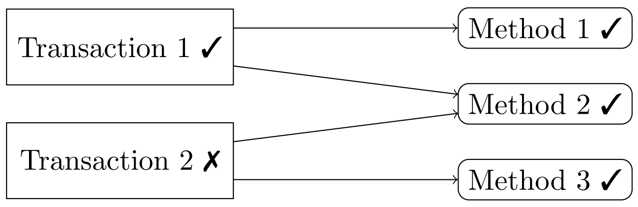

Figure 3.4: When two or more transactions want to call the same method(s) only one transaction can execute in a given clock cycle

These two building blocks allow creation of complex systems of hardware blocks that interact with each other without having to worry about structural hazards and different timing requirements of each component, as the library handles transaction scheduling and arbitrage. All this greatly reduces the burden on the programmer, who can now focus on the core problem that needs to be solved instead of the details of interacting with other components.

More details can be found in the Transactron documentation [20].

Chapter 4

Core architecture overview

This chapter serves to describe the Coreblocks core's microarchitecture as it was in January 2024. It was designed collectively by the Coreblocks development team (that the author was a part of) and evolved over the span of 2.5 years. A simplified diagram of its pipeline is given in figure 4.1. The lifecycle of an instruction roughly follows these stages:

1. Fetch
2. Decode
3. Resource allocation
4. Register renaming
5. Scheduling
6. Execution
7. Commit
8. Retirement

Subsequent sections delve into details of each stage.

4.1 Shared data structures

Before discussing the pipeline, a description of data structures shared across multiple stages of the core needs to be given.

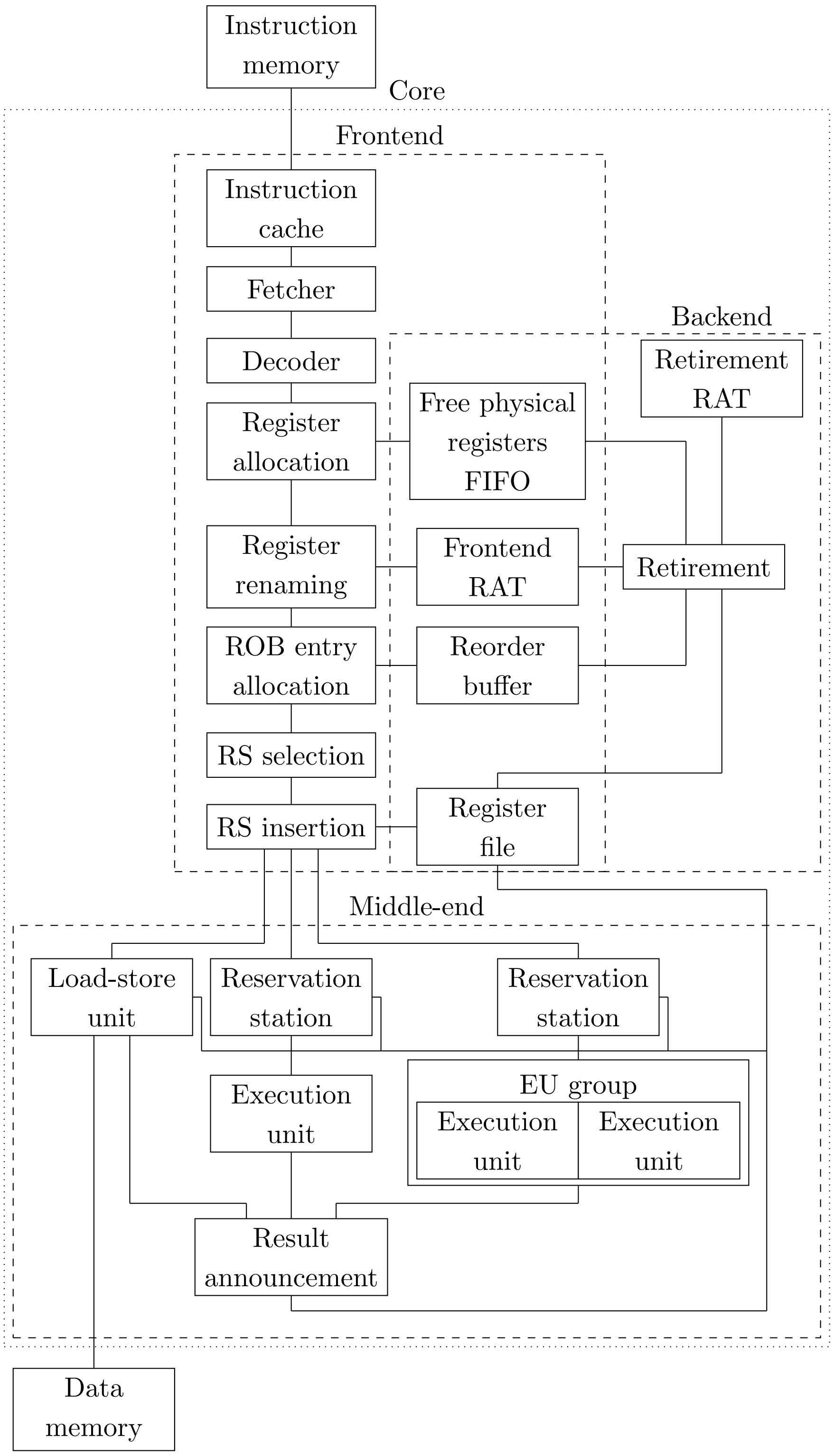

Figure 4.1: Coreblocks core microarchitecture

4.1.1 Register file

Register file in in-order microarchitectures is usually a memory with some read ports and some write ports with direct mapping between addresses and *architectural registers* (or, interchangeably, *ISA registers*). If one could look at the values contained in this memory, they would find values that the architectural registers have at a given point in time.

In out-of-order microarchitectures however the register file only holds the values of ISA registers, but there's no direct correspondence between the address and which architectural register it maps to. Registers in this kind of register file are called *microarchitectural registers* (or, interchangeably, *physical registers*), and there are many more of them than ISA registers. More precisely, microarchitectural register is a storage location for the result of an instruction. When an instruction enters the core, it gets allocated a physical register from a pool of free such registers. Later on when it finishes its computation, its result is stored in a physical register allocated to it.

By looking at the register file alone at a given point in time there is no way to determine what the values of ISA registers are, since the correspondence between them (what was encoded as the target register in the instruction itself) and microarchitectural registers (what was allocated by the core to the instruction) is stored elsewhere – in the *Register alias tables* described in section 4.1.3.

4.1.2 Free physical register IDs FIFO

To allocate a physical register for the result of an instruction, there needs to be a way to query the register file for free slots (storage locations that are currently unused). Instead of implementing this as part of the register file, a separate FIFO is maintained with IDs of free slots (further referred to as *register IDs*) in the register file.

Initially the FIFO is prefilled with register IDs in the range from 1 to a configurable size. Register ID 0 is not inserted during this initialization stage because of a microarchitectural design decision to tie it to the constant `0` and make it read-only, as described in section 4.1.1.

During normal operation register ID is taken from the FIFO when an instruction is a result-producing instruction and thus needs a storage location for its result. Register ID is inserted into the FIFO when the value that the register with that ID was holding will no longer be needed by the core. How to determine this condition will be described in section 4.5.1.

4.1.3 Register alias table

Register alias table (or RAT) is a data structure that establishes correspondence between ISA registers and physical registers. It's an array as large as there are registers (in RISC-V usually 32). At each index the ID of a physical register that holds the current value of an ISA register of this index is stored. Thus at its core it is an array of pointers to the register file. Figure 4.2 illustrates this.

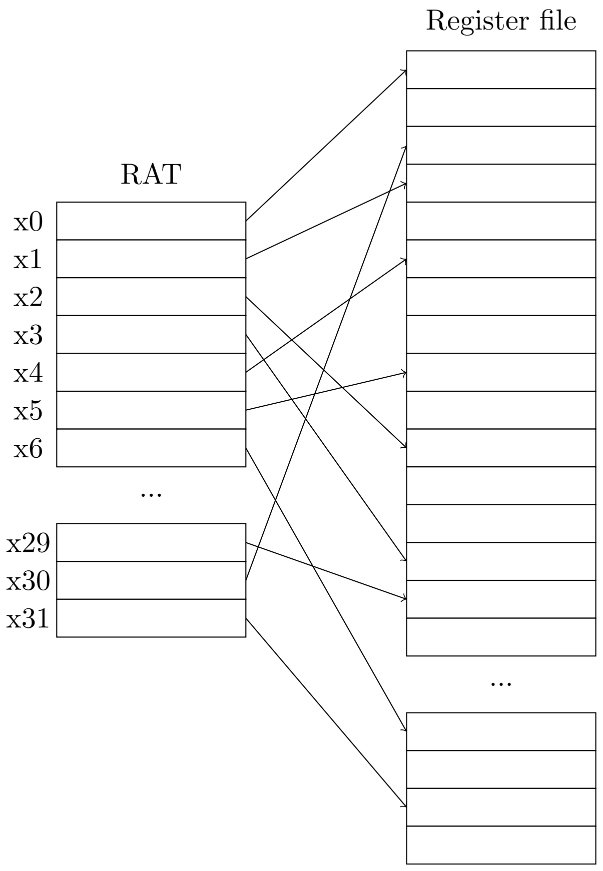

Figure 4.2: Register alias table points at registers in the register file

Frontend RAT is central to register renaming which will be covered in section 4.3.2. During normal operation it holds a *speculative* view of the state of ISA registers, i.e. some entries point to registers in the register file that hold/will hold the results of instruction executed speculatively.

Retirement RAT is used at retirement stage and holds an actual view of the state of ISA registers, i.e. if we were to halt the core at a specific point in time and ask “what is the current state of all architectural registers?”, this is where we would look. As we will see in section 5.5 retirement RAT is required for correct handling of traps, exceptions and interrupts.

4.1.4 Reorder buffer

Reorder buffer (or ROB) is a queue with a few extra operation defined on its contents that holds information about all instructions currently present in the core and tracks their completion status.

Each entry in the ROB contains the following fields:

- **logical register** – index of destination ISA register that was encoded in the instruction,
- **physical register** – index of destination microarchitectural register that was allocated by the core for the result of an instruction,
- **done** – boolean flag indicating whether the instruction has finished and wrote its result to the physical register assigned to it,
- **exception** – boolean flag indicating whether the instruction's execution has caused an exception. Type of the exception is stored elsewhere in a global register.

An entry for the instruction is allocated after it has reached past the physical register allocation stage in the core, since both logical and physical register indices are known at this point. The instruction is assigned a *ROB ID* during this, and this ROB ID from that point on travels through the pipeline with the instruction up to the commit stage.

While ROB doesn't necessarily have to work like a FIFO queue, it's easier to implement it that way. Some additional functionality is also required:

- marking the instruction as done based on the supplied ROB ID (note that this requires random access as opposed to FIFO access),
- retiring the instruction, i.e. deallocating its entry (or popping from the queue), but only when it's marked as done.

Reorder buffer is the central entity in the core – instructions are inserted in-order, can execute (i.e. can be marked as done) out-of-order, and are retired in-order, thus maintaining the illusion that the program is executed sequentially. Each of these stages will be described in the following chapters.

4.2 Control and status registers

Control and status registers (CSRs in short) are registers that, in general, control the core's overall behavior and provide information about its state. Examples include:

- performance counters,
- registers for configuring behavior and reading status of interrupts,
- registers for configuring physical memory protection.

For a full list refer to the RISC-V Privileged Specification [9].

CSRs are accessed by a set of read/write/read-modify-write instructions dedicated for them. These are handled by a separate execution unit described in section 4.4.2 that implements correct semantics of these operations.

The base implementation of a CSR is a generic register with two sets of read and write methods – one for the execution unit, and one for the hardware that uses them, with the former taking priority. Some writes can be ignored by specifying a bit mask of read-only bits of that register. Examples of instances of such registers that are currently used are:

- `mcause` – machine exception cause – stores the ID of an event that caused an exception,
- `mtvec` – machine trap-vector base-address register – stores the address of an interrupt handler or an interrupt vector table,
- `mepc` – machine exception program counter – stores the return address from an exception/interrupt.

This generic implementation can be specialized to provide CSR-specific functionality. An example of this is a counter CSR that increments itself every cycle that can be used e.g. for benchmarking how many cycles executing a given piece of code took.

4.3 Frontend

Frontend is part of the core responsible for fetching the instructions, allocating dynamic resources that the instruction needs (physical register for the result, ROB entry), renaming registers and scheduling the instruction for execution. We will look at each of those in detail.

4.3.1 Instruction fetch and decode

Entity responsible for fetching instruction is the *Fetcher*. It contains the program counter and a speculative program counter and issues requests for successive instructions. All requests go through an intermediary instruction cache. It's connected to the outside world – the instruction memory – with a Wishbone bus. The cache is optional and can be substituted by a dummy hardware block that just forwards the requests directly to memory and doesn't have any actual storage.

Wishbone [21] is an open source computer bus standard that specifies a set of signals and their behavior (together called an interface) to define a communication protocol. It's commonly used in non-commercial projects due to its simplicity. This is the primary reason why was it chosen for the core in the first place. Hardware blocks that implement this specification (i.e. have all required external signals that behave according to the specification) can be connected together, provided that one side of the connection is a master and the other a slave. In this case the instruction memory has a Wishbone slave interface, while cache has a Wishbone master interface.

After the fetcher receives the instruction it forwards it to the decoder, which unpacks the instruction into a format that is easier to handle later on in the pipeline. Its fields are:

- `exec_fn` – groups fields responsible for selecting the opcode:
  - `op_type` – custom operation group code used for routing the instruction to an execution unit that can handle it,
  - `funct3` – `funct3` field from RISC-V ISA,
  - `funct7` – `funct7` field from RISC-V ISA,
- `regs_l` – groups logical (architectural) register indices:
  - `rl_dst` – destination ISA register,
  - `rl_s1` – first source (operand) ISA register,
  - `rl_s2` – second source (operand) ISA register,
- `imm` – immediate if the instruction encoding contains one, else constant `0`,
- `csr` – index of the CSR register if the instruction explicitly accesses one,
- `pc` – program counter associated with this instruction.

Depending on the configuration, the core can be generated with support for the **C** (compressed instructions) extension. This substitutes the fetcher with one that can handle instructions aligned to 2 bytes (as opposed to 4 bytes without the **C** extension) and inserts a *decompression* block that translates instructions from 2-byte compressed format into standard 4-byte instructions. There's some additional functionality embedded into the fetcher related to handling branches but this will be covered in section 4.4.3.

The pipeline for this stage is presented in figure 4.3

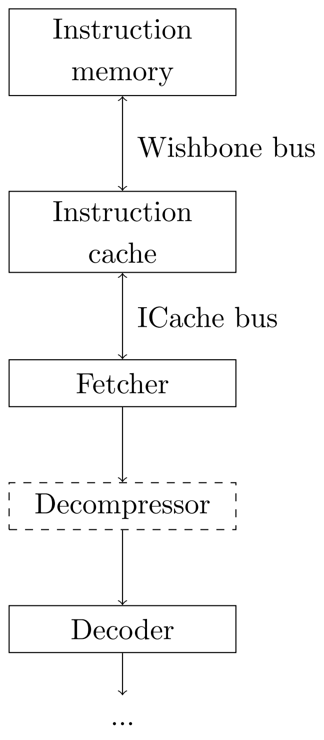

Figure 4.3: Instruction fetch and decode stage

4.3.2 Scheduler

Scheduler is a part of the core that is responsible for allocating resources, performing register renaming and scheduling the instructions for execution. This pipeline starts off with physical register allocation. Register allocation stage receives a decoded instruction from the decoder and looks at the destination register. If it's not `x0`, then it pulls a free value from the free physical registers FIFO and appends it to the list of fields described in section 4.3.1 under the name `rp_dst`.

Next up is register renaming. This is performed by supplying the frontend RAT with the following data:

- `rl_s1` – first source (operand) ISA register,
- `rl_s2` – second source (operand) ISA register,
- `rl_dst` – destination ISA register,
- `rp_dst` – physical destination register.

For each source register, the frontend RAT performs a lookup in its memory and returns the physical registers associated with them. The source ISA register IDs `rl_s1` and `rl_s2` are no longer needed further down the pipeline and are instead replaced by fields `rp_s1` and `rp_s2` respectively. These contain respective physical register IDs received from the frontend RAT. For the destination register, it replaces the contents in the frontend RAT at index `rl_dst` with the value of `rp_dst`. This effectively tells future instructions that have one of its operands be the same logical register `rl_dst` that they should look for its value in the physical register `rp_dst`. One observation to note is that the *tag* described in section 2.2 is the value of `rp_dst` itself.

After that a ROB entry allocation is performed. This simply appends `rl_dst` and `rp_dst` at the end of the ROB queue and a `rob_id` of the allocated entry is returned and passed down the pipeline.

Further down the pipeline is reservation station selection stage. It performs two tasks:

- Looks at the `op_type` of the instruction, execution unit groups available in the core and what `op_type`s they support and whether or not they have free slots available. Based on that it decides which reservation station (responsible for a particular execution unit group) is going to receive it.
- Sends a request to the selected reservation station to allocate a free slot for this instruction. In response to this request a slot ID is returned.

As a result two more fields are pushed further down the pipeline after this stage: `rs_selected` with the selected reservation station's ID and `rs_entry_id` with the allocated slot ID within this reservation station.

Last is reservation station insertion. Apart from routing the instruction into the reservation station selected in the previous stage, it also performs a lookup of `rp_s1` and `rp_s2` in the register file. As the values of these registers (i.e. results of previous instructions) might not have been computed yet, the register file additionally returns whether or not the value of a register is valid (has been computed and stored in the register file) at the time of the query. If the value of `rp_s1` or `rp_s2` isn't known at that point in time, they are stored in the reservation station itself as tags – they will be compared with tags announced on the common data bus (a process described in more detail in section 4.4.1), otherwise constant `0` is stored.

The reason for splitting reservation station selection and insertion was mostly motivated by wanting to avoid performing too many computations (look up free slots in relevant reservation stations, compute the slot to take, route the data there) in a single clock cycle.

Figure 4.4 illustrates stages described above.

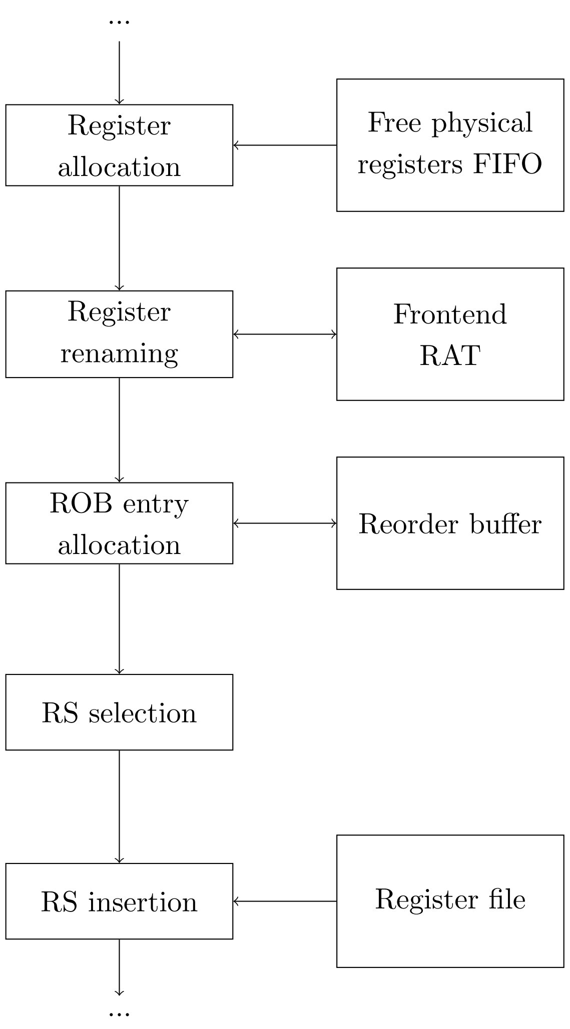

Figure 4.4: Resource allocation and register renaming in the scheduler

4.4 Middle-end

Middle-end is part of the core that can execute instructions out-of-order. It consists of reservation stations and various execution units. It's the job of previous stages to keep these execution units occupied for maximum utilization of computational resources of the core.

4.4.1 Reservation stations and instruction scheduling

As described in section 2.3, reservation station (or RS) is a data structure where an instruction waits until all of its operands are ready and the functional unit that is capable of executing it is ready. In this subsection we will look at it in more detail.

Each entry in a reservation station has the following fields (their meaning is the same as previously):

- `rp_s1` – first source (operand) ISA register,
- `rp_s2` – second source (operand) ISA register,
- `rp_dst` – physical destination register,
- `rob_id` – ROB entry ID allocated for this, instruction
- `exec_fn` – operation to ,
- `s1_val` – value of the first operand (might be initially unknown),
- `s2_val` – value of the second operand (might be initially unknown),
- `imm` – immediate if the instruction encoding contains one, else constant 0,
- `pc` – program counter associated with this instruction.

Reservation station listens on the common data bus for broadcasts of `(tag, value)` tuples and on each broadcast compares `tag` to both `rp_s1` and `rp_s2`. If e.g. `tag == rp_s1`, then `s1_val` is assigned the value of `value`, and `rp_s1` is zeroed to signify that this operand's value is known. Same happens for `rp_s2` and `s2_val`. When both `rp_s1` and `rp_s2` are zeroed, this condition indicates that the instruction is ready to be executed (refer to figure 2.3 for a simplified visualization this process).

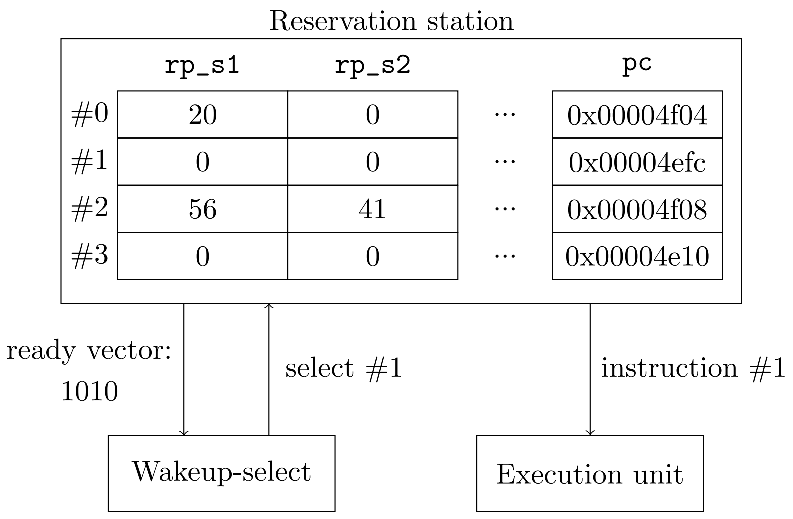

Figure 4.5: Wakeup-select choosing an instruction to be executed in a given clock cycle

A wakeup-select hardware block is responsible for arbitrating which of the ready instruction should be selected ("woken up") to be passed down to the execution unit in a given cycle. Reservation station exposes an interface that allows wakeup-select to get a *readiness vector*. It's a bit vector where for each entry in a reservation station, the i-th bit indicates that the i-th entry is ready. The index of the last non-zero bit is used as the index of the entry that will be scheduled for execution. It's a simple algorithm but is sufficient in the current (non-superscalar) implementation of the core. The selected index is supplied to the reservation station, which returns all of the fields listed above (except for `rp_s1` and `rp_s2` as they're needed only for listening on the common data bus for value broadcast) and they are passed down to the execution unit, at which point the entry at that index is freed and can be reused (figure 4.5).

4.4.2 Execution units

Each execution unit supports a particular subset of instructions grouped by similarity. Currently the core can be configured to include:

- Standard ALU – supports arithmetic and bitwise operations from the base **I** (integer) extension as well as a subset of instructions from the **Zba** (address generation) and **Zbb** (bit manipulation) extensions, with the exception of shift instructions by a variable amount.
- Shift unit – covers all shift instructions by a variable amount from the base **I** extension and **Zbb** extension if it's enabled. Since some shift instructions are required by the **I** extension this unit always has to be included whenever standard ALU is included. Separating this functionality into a separate unit makes the overall implementation more modular.
- Jump-branch unit – covers all jump and branch instructions as well as `auipc` from the base **I** extension, with support for jump/branch targets aligned to 2 bytes if **C** (compressed instructions) extension is enabled.
- Load-store unit – covers all load and store instructions from the base **I** extension. Due to uniqueness of loads and stores in out-of-order cores its operating principles are also unique and will be discussed in detail in section 4.4.4.
- Multiplication unit – can execute all integer multiplication instructions from the **M** (integer multiplication and division) extension. Different multiplicator implementations are available:
  - shift-based – resource-cheap multi-cycle multiplier that implements Russian Peasants multiplication,
  - DSP multi-cycle – uses a single DSP block to generate a multi-cycle recursive implementation,
  - DSP single-cycle – uses as many DSP blocks as required to generate a single-cycle recursive implementation.
- Division unit – supports all integer division instruction from the **M** extension.
- Exception unit – special execution unit that handles instructions that are supposed to cause exception.These include instruction that are part of the ISA

like `ecall` or `ebreak`, or custom instructions internal to the microarchitecture and can be artificially inserted into the pipeline, e.g. `illegal_instruction`. This handling usually only means reporting a corresponding exception of the same name to the *exception cause register*.

- CSR unit – responsible for handling all instructions for reading and modifying CSRs (control and status registers). All instantiated CSRs scattered across the core are gathered and handled by this unit. Due to semantics of reads and writes on some of the CSRs, special care is taken that these are not executed in a speculative context.
- Privileged instruction unit – responsible for handling opcodes from the Privileged ISA [9]. Currently only handles `mret` (return from machine-mode interrupt) which is required for interrupt handling.
- Zbc unit – handles opcodes from the **Zbc** extension (carry-less multiplication – multiplication in the polynomial ring over $GF(2)$).
- Zbs unit – handles opcodes from the **Zbs** extension (single-bit instructions that set, clear, invert or extract a single bit in a register).

Each execution unit also contains its own dedicated decoder that converts the `(op_type, funct3, funct7)` tuple coming from the reservation station into an internal one-hot bit vector format for a more streamlined implementation of the unit itself.

4.4.3 Jump-branch unit

Handling branches is somewhat inefficient in the current core implementation, but it allowed for processing branches without having to worry about implementing handling branch speculation mispredictions (which is one of the central parts of an out-of-order core but is also a huge task in itself) at the early stage of project development. The idea is simple – if we detect that an instruction is a branch at fetch stage (or any jump instruction for that matter), we stall fetching further instructions until an information from the jump-branch unit where from to start fetching next is received. This information arrives at the fetcher soon after the jump or branch instruction that triggered the stall is executed as at that point a target program counter is computed. After arrival the fetcher is unstalled and proceeds like normal until another branch or jump is encountered.

4.4.4 Load-store unit

Load-store unit (or LSU) is perhaps the most complicated functional unit in the core and will remain so in the future. Current implementation aims to be as simple as possible while still supporting all instructions required by the **I** extension.

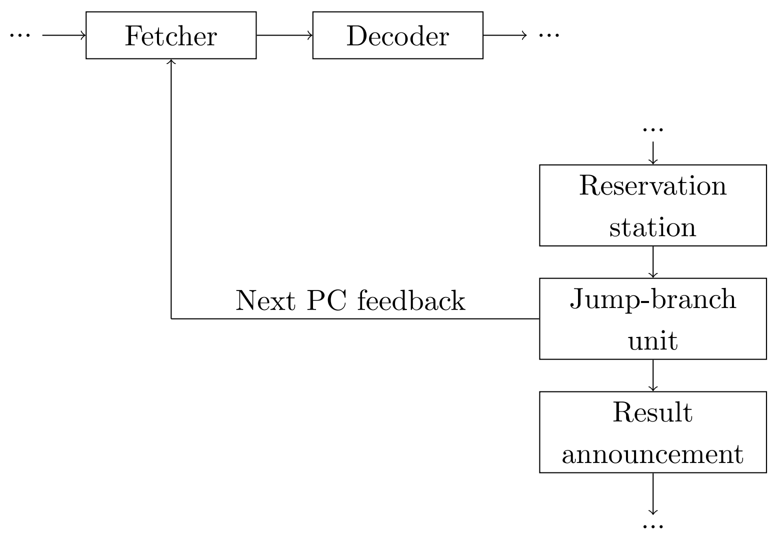

Figure 4.6: Jump-branch unit feeds a computed next PC to the fetcher

First thing to note is that it doesn't follow the usual scheme of having a corresponding reservation station. The LSU exposes an interface partly that of a reservation station to receive instructions from the pipeline, and partly that of an execution unit to submit the results for announcement on the common data bus. This is a deliberate choice dictated by the fact that the LSU must know the order of instructions as they appear in dynamic execution of the program. A reservation station can schedule instructions to be executed in different order and information about that order is lost when the instruction is inserted into it. This knowledge is needed for maintaining a correct order of operations on memory. What constitutes a "correct" order is specified by the RISC-V memory consistency model. RISC-V uses a model called **RVWMO** (or RISC-V Weak Memory Ordering) [10], with an optional **Ztso** (total store ordering) extension.

Current LSU implementation executes instructions sequentially, as there can be only a maximum of 1 instruction present it the LSU at any given time. Instruction's lifecycle inside it is illustrated in figure 4.7.

As in a standard reservation station, the instruction first waits for all of its operands. Once they're ready, what happens depends on whether the instruction is a load or a store.

Store needs a special signal from retirement called *precommit* to be asserted for an instruction with this particular ROB ID before it can start executing. It controls when side effects (e.g. modifying memory contents) of instructions happen. For the correct handling of interrupts they have to be precisely controlled, as side effect can't happen for an instruction that is considered to have happened after an interrupt. This signal is asserted by retirement when an instruction is ready to be retired, i.e. it is at the dequeue pointer of the ROB queue. Once that has happened, a store request is sent over Wishbone to data memory and this either causes an exception

– either `STORE_ACCESS_MISALIGNED` when the address was not correctly aligned or `STORE_ACCESS_FAULT` when the bus returned an error for whatever reason (e.g. no such address exists in the physical address space). Either way a result (successful store or an exception) is returned.

Loads on the other hand can be executed speculatively provided they don't target an MMIO region, so they don't need to wait for the precommit signal. Recognizing this condition is the responsibility of a separate *physical memory attributes checker* hardware block that allows defining such memory ranges. After a request for data to memory is sent, it can similarly cause one out of two exceptions – `LOAD_ACCESS_MISALIGNED` or `LOAD_ACCESS_FAULT` – and regardless of this either read data or exception is returned.

The **I** base extension mandates that the core must also support the `fence` instruction. Because of the implicit serialization of loads and stores nothing has to be done to implement its proper semantics – it is effectively treated as a no-op.

4.4.5 Result gathering and announcement

After an execution unit has performed its computation the result (along with some metadata, e.g. ROB ID, physical register index) is placed in its dedicated FIFO queue. The results from all queues are then serialized and passed to a hardware block that:

1. marks the instruction in the reorder buffer as being done or having produced an exception,
2. writes the result to the register file,
3. sends an announcement (tuple `(rob id, result)`) on the common data bus to all the reservation stations.

New instructions that need this result can thus find it in the register file, while slightly older instructions that have already reached reservation stations will receive the announcement and can discover the value of their operand that way.

This is illustrated in figure 4.8.

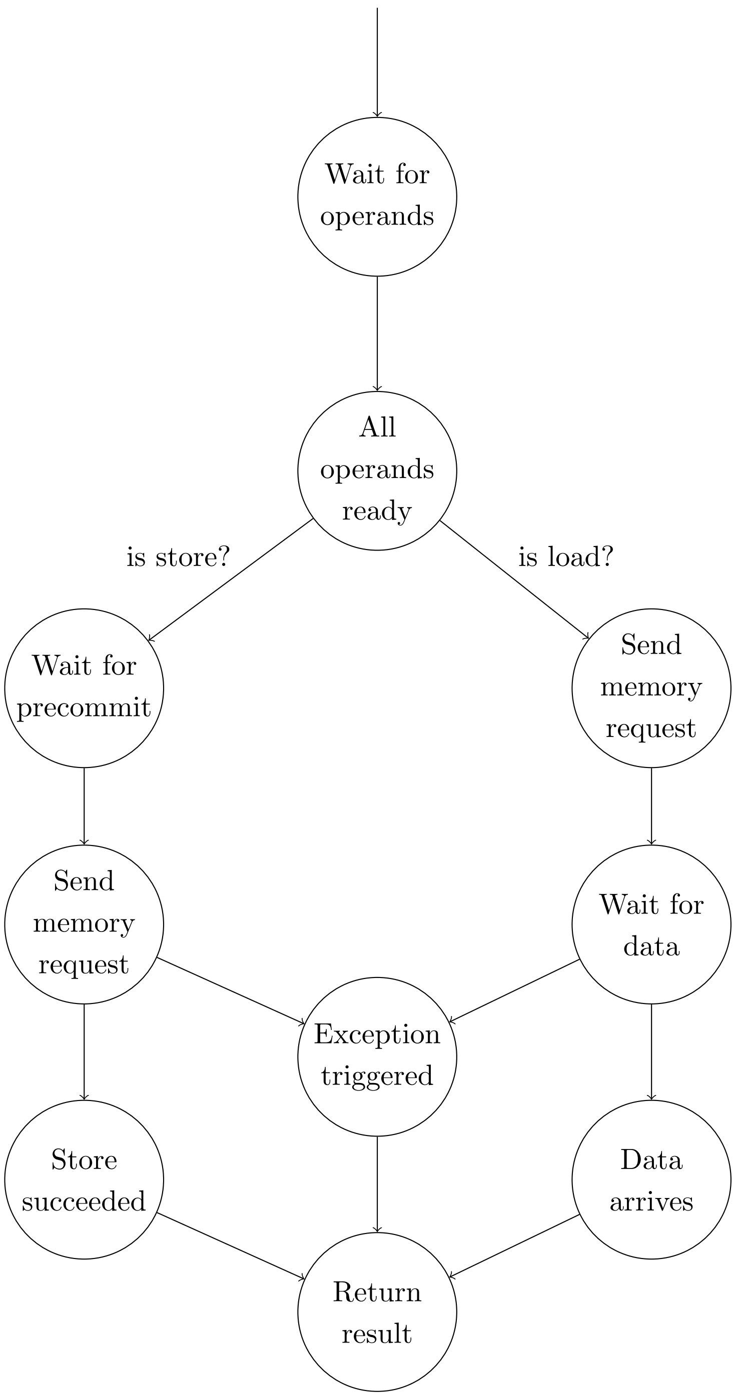

Figure 4.7: Load or store instruction's lifecycle inside the LSU

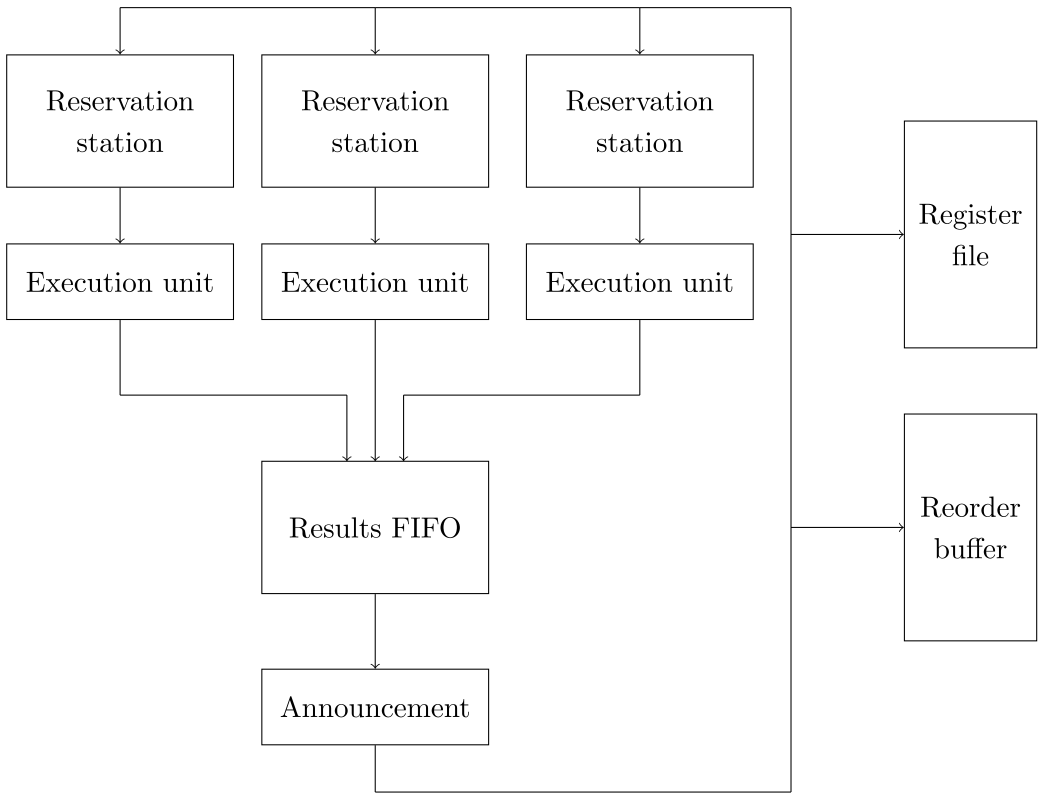

Figure 4.8: Announcement stage

4.5 Backend

Backend is part of the core responsible for *committing* (making results of the instructions architecturally visible) and *retiring* (freeing dynamically allocated resources) an instruction.

4.5.1 Retirement

Contrary to its name, the retirement hardware block handles both committing and retiring an instruction. Committing means applying a state-changing action in either:

- the outside world (e.g. writing to memory),
- the retirement RAT, which indirectly holds the architectural state of the core.

Retiring means freeing resources dynamically allocated in:

- register file,
- reorder buffer.

These two are performed in tandem via the following algorithm:

1. Look at the dequeue pointer in the ROB. If it points to an instruction that has been marked as done, mark it as retired (invalid) and receive indices of both physical and logical destination registers.
2. Perform a lookup and replace in the retirement RAT – under the index of logical register read the index of *old* physical register and write the index of *new* physical register.
3. Mark the old physical register in the register file as free (invalid).
4. Push the old physical register index to the free physical registers FIFO.

This is all performed in a single cycle as illustrated in figure 4.9.

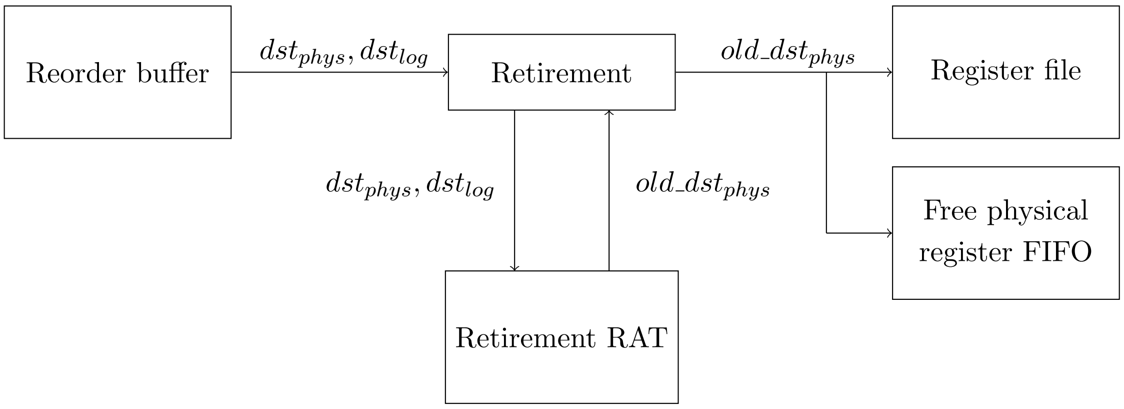

Figure 4.9: Retirement stage

4.5.2 Exception handling

Some processes in the core can fail. Those defined in the ISA include misaligned memory accesses, memory accesses with insufficient privileges, page faults and processing illegal instructions. These situations cause an *exception* – "exceptional condition" that needs to be manually handled by the programmer. A special exception handling routine is invoked when this happens, e.g. in case of a page fault perhaps the page wasn't loaded into memory and has to be fetched from disk. Once the handler finishes control is passed back to the previously executing program. Other situations that cause an exception, but this time on purpose, are execution of breakpoint and environment call instructions. These might be used for debugging or to implement system and hypervisor calls. Current implementation also adds two other internal exception sources – arrival of asynchronous interrupt and branch misprediction. These are treated as exceptions to facilitate reuse of existing mechanisms for handling them, thus avoiding additional complexity in hardware.

In the most common case the exception handling mechanism works as follows:

1. An instruction, while being executed, encounters one of the exceptional conditions outlined above (for instructions that trigger exceptions on purpose their execution is the exceptional condition).

2. `exception` bit in the ROB for the entry corresponding to that instruction is asserted and the exception cause is stored in a register dedicated to this purpose. Only one global exception cause is needed (as opposed to one per entry in the ROB) since only soonest-to-be-retired instruction will trigger exception handling – all subsequent instructions will be flushed, thus we don't need to store information about causes of their exceptions.

3. Once the instruction is about to be retired, the retirement stage looks at the `exception` bit in the ROB and since it's asserted the instruction is not committed and relevant CSRs (`mepc`, `mcause`) are set to appropriate values.

4. A flushing state is entered, where the core is waiting for all instructions to go through the core and reach retirement stage. The number of instructions still currently in the core is tracked by a counter incremented when instruction is pushed from the fetcher to a further stage. The counter is decremented when an instruction is retired or flushed. Once it reaches 0 we are guaranteed to have emptied the ROB.

5. During this process physical register ID of the flushed instruction is recycled and Frontend RAT state is rewound. This is accomplished by rewriting architectural-physical register mappings in the Frontend RAT with old mappings from the Retirement RAT (since it holds a state that Frontend RAT held before all uncommitted instructions were renamed). Since information about flushed instructions is available, in particular their architectural destination registers, this is done only for corresponding entries in the Frontend RAT, since only these can possibly differ from their counterparts in the Retirement RAT. As this is performed in parallel with flushing no additional delay is incurred.

There are a few cases where this process slightly differs.

1. In case of a failed instruction fetch or detecting an illegal instruction during decoding, a special microarchitectural opcode with information about the exception cause is injected into the instruction stream. During its handling in a dedicated execution unit it always triggers an exception (asserts `exception` bit in the ROB).

2. In case of mispredicted branches the offending branch triggers a special microarchitectural exception for branch mispredictions. During retirement it **is** committed and CSRs related to exception handling **are not** modified.

3. In case of an asynchronous interrupt (that is – when global interrupt bit is asserted), a special microarchitectural exception is triggered on branch, jump, CSR-related or interrupt return instructions. This specific set of instructions types is dictated by the need to know address of an instruction to return to from an interrupt handler (branches and jumps have it readily available since they calculate it explicitly) and by the RISC-V specification (as it dictates that CSR and interrupt instructions must be considered as interrupt entry points). During retirement the instruction that triggered this interrupt **is** committed and CSRs related to exception handling **are** modified appropriately.

Chapter 5

Implementation

Due to the highly collaborative nature of the project and continuously-changing codebase, author's contributions are shown in their original form at the time when they were contributed instead of their form at the time of writing, unless noted otherwise. This serves to clearly separate the code originally written by the author of this work from the code authored by other contributors and make the description more focused. All sections are going to have links to particular commits that are discussed in a section.

The initial core microarchitecture was designed and agreed upon collaboratively. Some parts have been written early in the development stage of the project so they're going to differ from the description of the current state of the core given in chapter 4. Major differences are going to be explicitly mentioned. Due to the agile style of development of the project all changes required mandatory tests and went through a code review process with at least 2 approvers, unless noted otherwise.

5.1 Scheduler

Commits for this contribution:

- `https://github.com/kuznia-rdzeni/coreblocks/commit/71fad2a5f026ef24aac4f9c5d72148fbff9f3e9f`
- `https://github.com/kuznia-rdzeni/coreblocks/commit/6b9a3f8f69c9f7f89fe4d4a56573a085c8397292`

This part of the core was first implemented in the very early stages of development but it retained the same structure as the scheduler described in section 4.3.2. As part of this change the register file and frontend RAT were also implemented.

Very early into the development a question was posed how should interfaces between pipeline stages be structured in terms of what Transactron entities should they use. Options available at the time were:

- Transaction in one hardware block that calls a method in another hardware block as in figure 5.1, or vice versa.
- Methods in both hardware blocks connected by another hardware block with a transaction, as in figure 5.2.

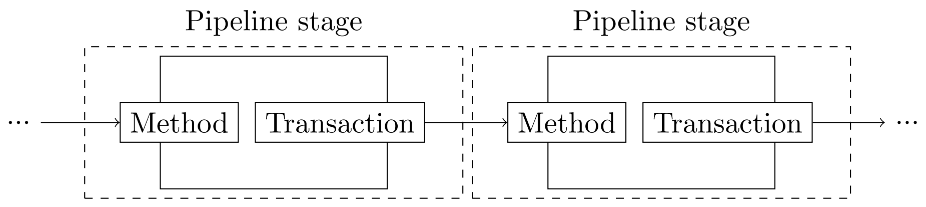

Figure 5.1: Transaction in one hardware block calls a method in another hardware block

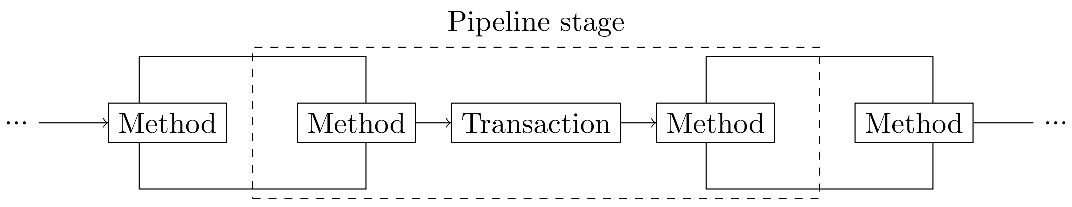

Figure 5.2: Transaction ties two methods together and exchanges data between them

Ultimately we agreed to use the second scheme since passing data between two methods called in the same transaction is trivial, whereas passing data between a method and a transaction in the same hardware block requires extra programming work.

It's useful to think about transactions in that scheme as stages in a pipeline performing some computation, surrounded by buffers on both input and output sides. In the implementation these are 2-element FIFOs queues. Note that using size of 1 would not work as that forces a 1-cycle delay between a write to the FIFO and a read from it, as both read and write methods can't be called simultaneously – either read can be called when the FIFO contains an element, or write can be called when it is empty.

Transactions connected with FIFOs in a more general view form a consumer-producer structure (producer transactions write to FIFOs which are then read by consumer transactions), so they can also be thought of as an analogue of software threads communicating through queues.

Figure 5.3 illustrates the scheduler implementation. Small boxes represent transactions, while larger ones represent data structures. Flow of data is represented by arrows between transactions and methods, the latter being part of some data structure.

Figure 5.4 presents an example pipeline stage. `get_instr` method is called to read data from the preceding FIFO, another method `get_free_reg` is conditionally called to allocate a physical register id (unless destination register is `x0`) and processed data is written to the following FIFO by calling `push_instr` with argument to write.

```
free_reg = Signal(self.gen_params.phys_regs_bits)
data_out = Record(self.output_layout)

with Transaction().body(m):
    instr = self.get_instr(m)
    with m.If(instr.rl_dst != 0):
        reg_id = self.get_free_reg(m)
        m.d.comb += free_reg.eq(reg_id)

    m.d.comb += data_out.rl_s1.eq(instr.rl_s1)
    m.d.comb += data_out.rl_s2.eq(instr.rl_s2)
    m.d.comb += data_out.rl_dst.eq(instr.rl_dst)
    m.d.comb += data_out.rp_dst.eq(free_reg)
    m.d.comb += data_out.opcode.eq(instr.opcode)
    self.push_instr(m, data_out)
```

Figure 5.4: Implementation of the register allocation stage

All scheduler stages are implemented in a very similar manner and work as described in section 4.3.2, with the exception of the reservation station selection stage, which doesn't choose between multiple reservation stations (since there's only one at this point in the development) and only reserves a slot for the instruction. As part of scheduler implementation register file and frontend RAT were implemented. Other data structures (ROB, reservation station, FIFO) were implemented by other contributors.

Register file is implemented as an array of configurable amount of individually-addressable registers that contain their value and a `valid` bit. The latter is required for distinguishing whether an instruction that has a particular register as its destination has already written a value there. There are two read ports and one write port implemented as Transactron methods, where write and read to the same register in the same cycle reads the new value. There's also a `free` method that sets `valid` bit to `0` for use when the physical register is freed. The 0th register is always kept

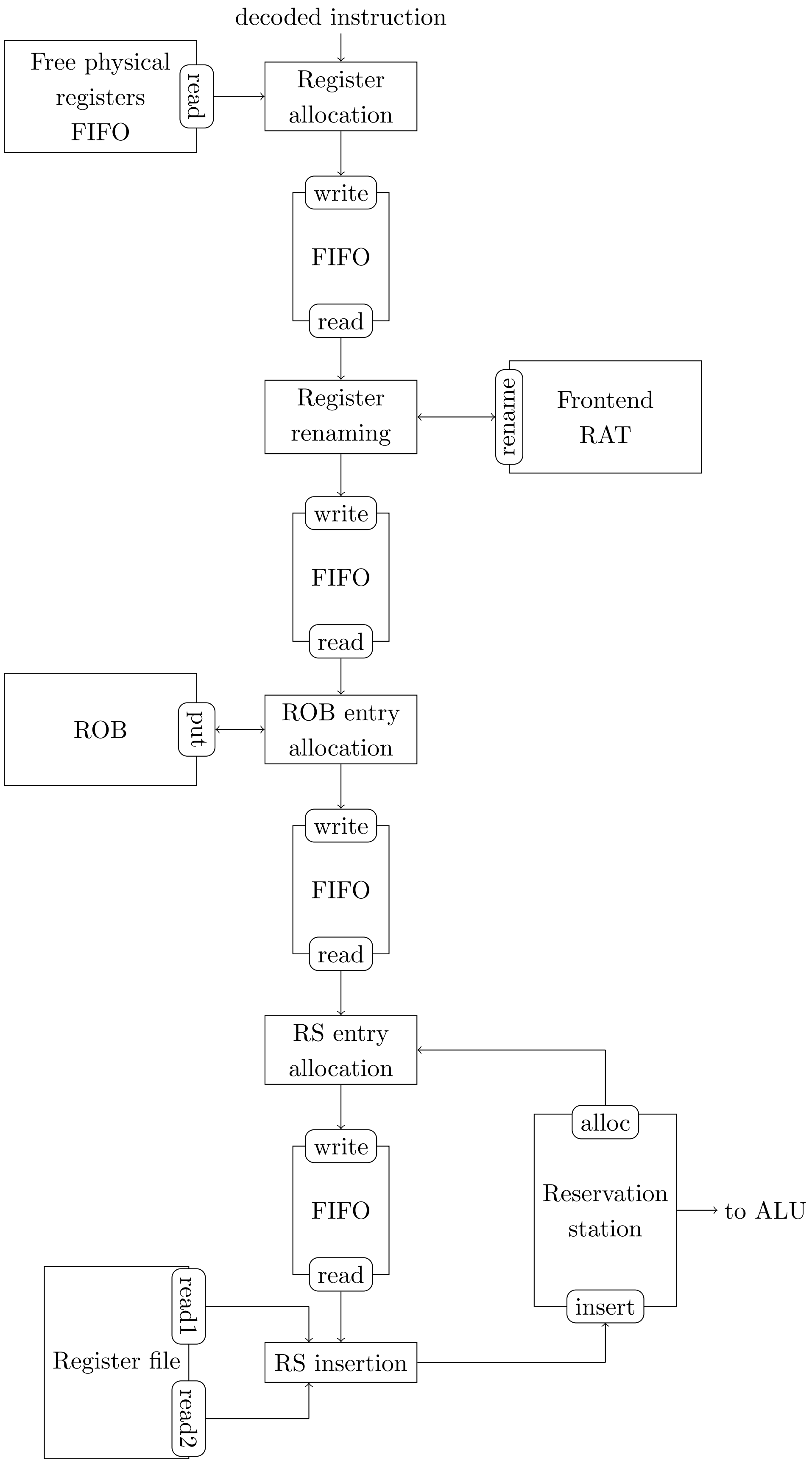

Figure 5.3: The scheduler pipeline in detail

valid with a value of 0 to simplify reads and writes from/to architectural register `x0` which is defined in the ISA as always having the value `0` and writes having no effect.

Frontend RAT is implemented as an array of 32 individually-addressable registers that map architectural registers to physical register ids. It implements only one method `rename` that updates the mapping for the supplied destination register and returns mappings for supplied source register(s) as described in section 2.3.

For testing fuzzing was employed – generating random inputs that are "correct enough" and don't break any assumptions that need to hold for a piece of data to get correctly processed. These are then fed into a simulated component and its outputs are cross-checked with the expected results calculated in software.

The scheduler pipeline was tested as a whole as opposed to testing individual stages. Methods of components that weren't internally used in the scheduler were controlled by the testbench. For example, free physical registers FIFO had to be manually prefilled with all register IDs at the beginning of the testbench and replenished with recycled IDs as instructions left the simulated pipeline using its `write` method. Reservation station was "mocked" entirely – methods controlled by the testbench were created and passed to the scheduler that behaved as if they were part of a reservation station's implementation. In this case testing didn't reveal any bugs since most of them were caught in code review.

5.2 Assembling and debugging the core

Commits for this contribution:

- `https://github.com/kuznia-rdzeni/coreblocks/commit/2db2b4bb641840689b2b3a0f4e553e9369099f60` – co-authored with Michał Opanowicz (24%) and Piotr Węgrzyn (12%)
- `https://github.com/kuznia-rdzeni/coreblocks/commit/ad87fe91ee41ebea395ece7d525878d76ffb694f`
- `https://github.com/Kristopher38/riscv-python-model/commit/b5d07371fed9666cc3895da675081ef459596f75`

After all the required components were implemented and tested, they had to be connected together to form a functioning processor core. Shape of most interfaces was agreed upon beforehand, but some were not finalized until this stage. In particular, scheduler's pipeline didn't contain proper fields for the execution unit further down to be able to execute instructions. Fields `opcode` and `exec_fn` that had enough information about the instruction were thus added to the pipeline.

After this prerequisite work was completed, all components were integrated together. Overview of the core at this point in the development in presented in figure 5.5

Testing came next. Initial tests were contributed by Michał Opanowicz. These have revealed an oversight in the implementation regarding handling of immediates which was promptly fixed by Piotr Węgrzyn.

Initial tests only tested one manually crafted scenario – a few registers were populated with data and then two `adds` and one `lui` instruction were performed. A more robust testing method was needed.

As with scheduler tests in section 5.1, fuzzing was also employed here. An open-source RISC-V emulator written purely in Python was used as a reference implementation [29]. Both the emulated core and our core in simulation were initialized to start from a known state and were fed a random instruction stream consisting only of arithmetic instructions, since only those were supported by our core at the time. After all instructions finished execution architectural register state was compared for any mismatch between the emulated and the simulated core.

This testing method was successful in finding some bugs in the implementation:

- `valid` bit in the register file wasn't cleared when a register was freed, making instructions that reused a previously used physical register read incorrect data as their operand. This was fixed by calling appropriate method of the register file in retirement.
- If an instruction had its physical destination register set to `0` (which implies architectural destination register `x0`), after instruction finished execution its result was announced on the common data bus by the result announcement stage. This was not only redundant since writes to register `x0` should be ignored but also caused spurious value updates in the reservation station tag with value `0` is often present there since it indicates that the operand's value has already been acquired (see section 4.4.1 for more details). With the right timing this caused some instructions to operate on incorrect data. It was fixed by conditionally sending the result announcement if the destination register was not `0`.

There were also minor bugs found in the implementation of the emulated core – some shift operations were not compliant with the RISC-V specification. Those were promptly fixed and a pull request was submitted upstream.

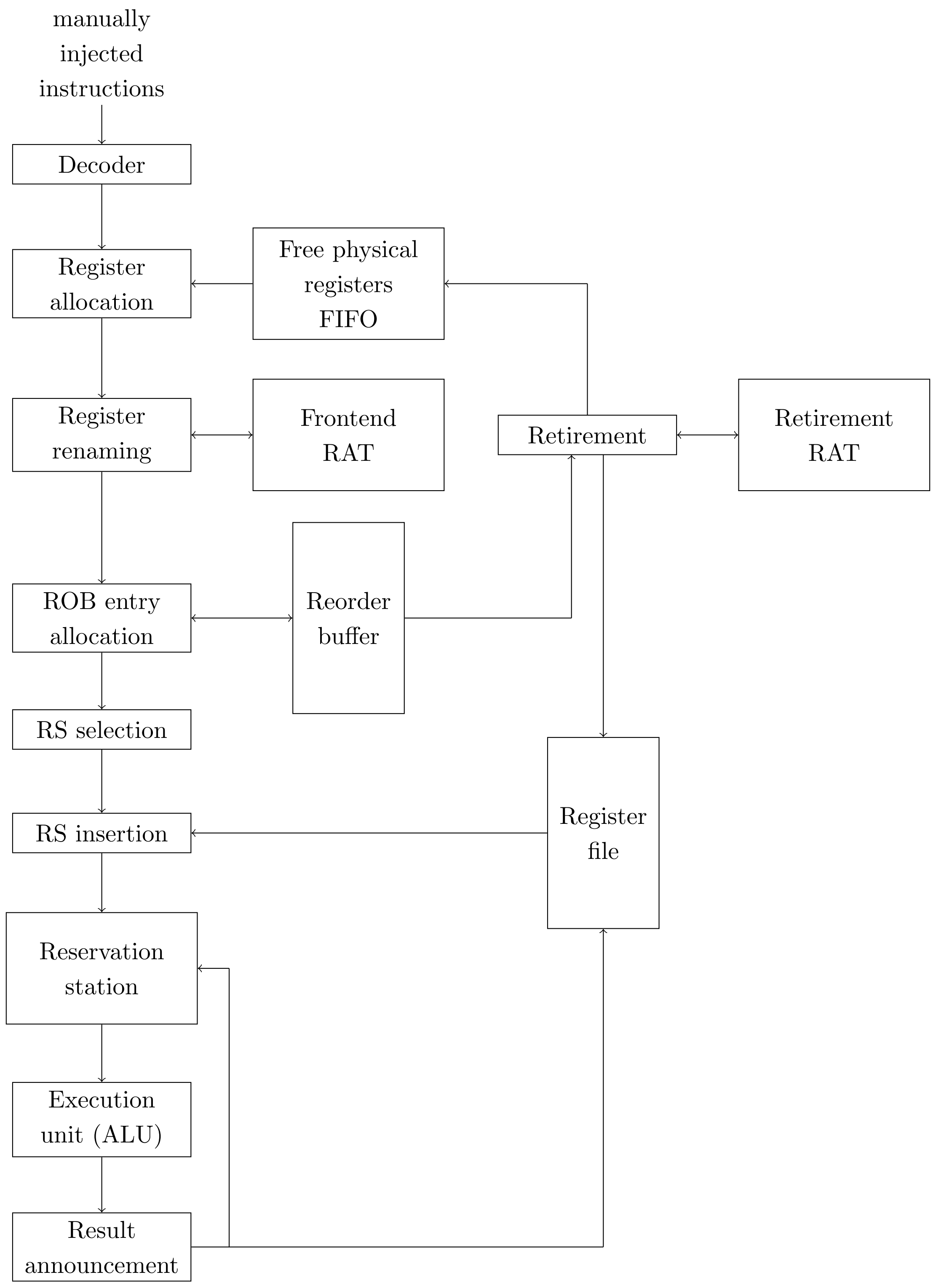

Figure 5.5: Initial core schematic after first integration

5.3 Instruction memory

Commits for this contribution:

- `https://github.com/kuznia-rdzeni/coreblocks/commit/f280080f54b24e6d6f4098b526eaec9ff40dc3f7`
- `https://github.com/kuznia-rdzeni/coreblocks/commit/7d6e6192358bb06ee7e272458c29b10732007573`

So far the core had to be manually injected with instructions, so even though we've had an implementation of the fetch stage, it was unused. At the beginning of the project it was decided that we're going to use Wishbone bus for interfacing with external peripherals. Thus the fetch stage had a Wishbone master interface, but we lacked an implementation of memory with a compatible Wishbone slave interface.

Initial implementation allowed adjusting data and address width, memory depth, initial contents, had one read and one write port and performed a standard, non-burst data transfers. Theoretical maximum performance was one transfer every 2 clock cycles since the Wishbone bus protocol dictates that an acknowledgement signal for each transfer in this mode is sent by the receiving side and a new Wishbone transaction is started for every transfer. In reality the implementation of Wishbone master used by the fetch stage only allowed one transfer every 4 clock cycles.

Fetch unit was then integrated into the core and memory initialized with randomly generated instructions was connected to the core in tests. This revealed a bug – since fetch assumed a byte-addressable memory while the implementation was a word-addressable memory, it fetched every fourth instruction. A simple fix was made to divide the addresses in requests sent by the fetch unit by 4. Since there was no support for the **C** extension yet (which would allow requests to instructions on a 2-byte boundary) this was correct.

5.4 Branch support

Commits for this contribution:

- `https://github.com/kuznia-rdzeni/coreblocks/commit/128312add363ad5fc393bd3ec2bd31010041a082`
- `https://github.com/kuznia-rdzeni/coreblocks/commit/91e54a59771966f729cb01c9c42d337aadf25039`

At this point the core was essentially a very elaborate calculator – it could not perform jumps nor alter its program counter in any way. Every program was a linear sequence of arithmetic instructions that operated on data registers only. To achieve full support of **RV32I**, handling of branch and other instructions that operate on the program counter had to be added.

First, a hardware block that is responsible for executing these instructions was implemented. This process involved closely reading the RISC-V specification [8] chapter 2.5 that details the semantics of each branch and jump opcodes, as well as parts of chapter 2.4 which contains the description of `auipc` instruction, and translating this specification into code. This *functional* unit (usually called *execution* unit in literature) was somewhat special in a sense that it produced two results – one was the result to be written into an ordinary data register, and one was the result to be written into the program counter to perform a jump.

Randomized tests were written to ensure the implementation was correct. Testing however has the following drawback: if identical wrong assumption is made in both the implementation and tests, they effectively cancel each other out. Much later on in the core's development a bug caused by the way instructions were decoded has caused instructions with large positive offsets to jump to an incorrect address due to immediate getting truncated. Further details are available in [28]. This was not caught in testing precisely because of incorrect assumptions in both places.

Next, the unit into had to be integrated into the core. Some minor modifications had to be carried out in various parts of the code:

1. Program counter field was added to the frontend's pipeline to facilitate calculating jump addresses in the functional unit.
2. Detecting jump and branch opcodes was introduced in the fetcher. If such opcode is detected fetching is stalled. This is performed before instruction decoding (this simplifies the whole process) by looking at specific bits in the instruction that unequivocally distinguishes them from other instructions.
3. Fetching is resumed when a signal from the functional unit with value of the PC where fetching should continue is sent from the functional unit.

Refer to figure 4.6 for a block diagram.

Last bit of the work was writing integration tests. To make this more streamlined, core tests were extended with the ability to compile and load RISC-V assembly sources. This was achieved by compiling them with RISC-V GNU assembler `as` and extracting the compiled machine code from the resulting file with `objcopy` and initializing instruction memory with it. The core was then simulated for a set number of cycles and architectural register state was compared with an expected state. A program to compute Fibonacci numbers was written in RISC-V assembly to test that branches and jumps do indeed work after integration. More comprehensive tests were introduced later by other contributors.

5.5 Interrupt handling

Commits for this contribution:

- `https://github.com/Kristopher38/coreblocks/commit/d8854c3271e4b6fe02b187e3746e388aed09d290`
- `https://github.com/Kristopher38/coreblocks/commit/ff22067db2e9728bdda4d72ac2cbb5da7a7c4288`

5.5.1 Introduction

A CPU that doesn't expose any means to affect its control flow from the outside (short of manipulating instructions in instruction memory itself) isn't very convenient to program for. Thus such mechanism in commonly implemented in CPUs and is usually known under the umbrella term *interrupts*. Microcontrollers commonly use interrupts for reacting to external and internal stimuli, such as button push or timer overflow. Operating systems rely on timer interrupts being available to implement preemptive multitasking.

Depending on the ISA *interrupts* might mean software or hardware interrupts, with different terminology for distinguishing types of interrupts. For the purposes of this section, the term "interrupt" will refer to *hardware interrupts* specifically – that is interrupts triggered by a change in one of the processor lines, internal or external (as opposed to being triggered by the execution of program).

One of the long-term goals of the project is to run Linux and Mimiker [19] operating systems on the Coreblocks CPU. A strong prerequisite for this is supporting interrupts. This contribution introduces necessary changes to divert the flow of execution to an interrupt handler and return back from it, while keeping all internal data structures in a consistent state. Ultimately it wasn't fully merged into `master` because an objectively better solution was discovered and implemented but it went through the full review cycle, spawned many crucial discussions about the approach to handling not only interrupts but exceptions (i.e. software interrupts) as well and tests implemented as part of this change were reused as tests for a later implementation.

5.5.2 Initial approach

There are multiple ways microarchitecture designer can implement interrupts in an out-of-order engine, each with their own benefits and drawbacks. This subsection contains a brief overview of approaches taken by other out-of-order cores where documentation on it could be found and details the approach used in this contribution.

- **RiscyOO** and **BOOM** – a special instruction representing an interrupt is inserted into the ROB in the frontend, which will be processed at commit stage when ROB dequeue pointer reaches it [15, 5].
- **Intel P6** – global interrupt signal is reported to the retirement stage and handled appropriately there, together with other events such as exceptions or mispredicted branches [25].
- **Coreblocks** (latest version) – interrupt is marked as an exception at jump or branch instructions and will be processed at the commit stage when ROB dequeue pointer reaches it.

In all cases subsequent entries in the ROB are flushed and Frontend RAT is restored to a non-speculative state as if instructions after the interrupt didn't go through renaming (how this is done is up to a particular implementation). This ensures that the core is in a well-defined state once it enters the interrupt handler – i.e. no uncommitted instructions that are said to have happened after the interrupt have affected the architectural state.

Initial proposal for interrupt handling used a different approach – every ROB entry would contain an `interrupt` bit that would be set by an interrupt handling unit and acted upon at retirement stage. This idea seemed reasonable at first because we thought that would help us unify interrupt and exception handling in the future (as exceptions would use a similar mechanism – setting a flag in the ROB on excepting instruction), but it was quickly abandoned since it didn't provide any benefits over having a single global interrupt flag.

After a global interrupt flag was raised, all ROB entries would be flushed, registers allocated for the corresponding instructions freed and Fronted RAT restored to the state of the Retirement RAT. Pending data in everything that keeps a temporary state – fetcher, functional units, reservation stations, FIFOs connecting various components would also be flushed.

It quickly turned out that this approach is insufficient in itself, for several reasons that we shall now examine.

5.5.3 Register leakage

One of the issues spotted even before the implementation began was possibility of "leaking" physical register IDs during a core flush. There are a few cycles of delay between a register ID being taken out of free physical registers FIFO and being stored in the ROB (as seen in figure 4.4), and as that value travels through the pipeline it's only present in FIFOs between the stages. There was thus a risk that a core flush would cause that register ID to be lost, unable to be used until a full core reset. This was remedied by combining ROB entry allocation with physical register allocation such that it happened atomically. This meant that a particular register ID was always either in the free physical registers FIFO or ROB.

5.5.4 Precommit stage

Instructions can be generally classified as either:

- Side-effect free – they only affect integer or floating point registers, e.g. `add`.
- Side-effectful – they perform some action on the state outside of the core, e.g. memory store, or they read/write CSR registers (as those can control the core's behavior in a significant way, e.g. turning support for compressed instruction on and off).

Up until this point all instructions started executing as soon as they entered their dedicated functional unit. Introduction of interrupts has created the possibility of speculative execution – since there could be a delay between raising and servicing an interrupt, all instructions that chronologically came after it was raised could've already executed and are now only awaiting to be committed and retired, but in reality they will be flushed. This is fine for side-effect free instructions as their impact on the overall state of the system is tightly controlled and localized – such flushing is able to reverse their changes to the core's state. On the other hand side-effectful instructions perform some work that changes the state of the system in such a way that the effects of it can't usually be undone.

This problem was recognized early in the implementation phase and sparked a discussion among the development team. Clearly there was a need to have a way to delay performing the side effects of an instruction (if it does any). Proposed solution was to implement a `precommit` method in each functional unit that would be called every cycle once an instruction is about to be retired (i.e. dequeue pointer in the ROB has reached it) until it's not marked as done. Such method would signal to the functional unit that the instruction tagged with a specific physical register ID can now perform its side effects as at that point we know with certainty that the instruction is not on a speculative execution path. This was implemented by Marek Materzok [13].

5.5.5 Implementation bugs

It's expected from the CPU architecture to provide a way to return from an interrupt back to the interrupted instruction stream. This return address is usually stored either on a general-purpose or a specialized stack, or in a dedicated register. On RISC-V it's the latter – it's a CSR named `mepc`. Under the implemented mechanism, natural place to get the interrupt return address from is address of the instruction that was about to be retired (while also discarding it and all subsequent instructions, to be re-executed after servicing the interrupt) – thus a PC field was added to all ROB entries to store it. More importantly, with the introduction of a `precommit` method described previously side effects were not performed for the soonest-to-be-retired instruction, address of which was meant to be stored into `mepc`. But even with this mechanism two bugs were discovered.

At first setting the PC by jump/branch unit wasn't considered as side-effectful. This has caused a particularly interesting bug that could occur while the interrupt was being prepared to get serviced. One of the steps in this process is setting the PC to the address of the ISR. With the right timing a jump instruction from the interrupted instruction stream could've already set the PC to a target address and overwrite the address pointing to the ISR. This was fixed by setting PC only when `precommit` was called for a particular instruction.

Since `precommit` might've already been called at least once for a soonest-to-be-retired instruction when an interrupt arrived, address of that instruction couldn't be used as a valid value for `mepc` since it might've already started performing its side effects. Thus tracking whether this has happened or not was implemented, and if that was the case the core waited until retirement of the next instruction. This introduced a subtle problem that could lead to an interrupt never being serviced. Starting the process of handling an interrupt was only possible when two conditions were satisfied: ROB was not empty and, due to slightly incorrect implementation, the core has retired an instruction in the previous cycle – the latter was to ensure that `precommit` hasn't been called for a subsequent instruction and that it hasn't started performing side effects yet. This meant however that there needed to be at least two allocated entries (instructions) in the ROB for interrupt handling to proceed. While this is usually the case, there is at least one case where this would never happen – in an infinite loop consisting only of one jump instruction that jumps to itself. Because a jump stalls the fetch stage, it's the only instruction (with a corresponding ROB entry) in the core at the point of its retirement. Once the target address of a jump is resolved fetching is resumed but in this case the next instruction is also a jump that will also be the only instruction in the core until its retirement. Thus begins a cycle where there's no chance to ever have two instructions (with two corresponding entries in the ROB) simultaneously in the core and so the interrupt is never handled.

5.5.6 Implementation details

The central part of interrupt handling is the *interrupt coordinator*. This hardware block receives an interrupt via an external `interrupt` method and contains a state machine to orchestrate the interrupt handling process that executes the following steps:

1. Wait for the `interrupt` method to get called. This is assumed to be done by a yet-to-be-implemented interrupt controller.

2. Wait for ROB to be nonempty (to always have a source of PC value to come back to from an ISR) and signal retirement to stall once an instruction that hasn't had precommit method called on it is about to be retired and wait for an acknowledge signal from retirement.

3. Save interrupt return address of soonest-to-be-retired instruction into `mepc` CSR, clear all state in the pipeline (functional units, pipeline FIFOs, reservation stations) and restore Frontend RAT from Retirement RAT. This is done by copying all values at once from R-RAT to F-RAT, but could very-well be done sequentially at the cost of larger interrupt processing time. This is also where the fetcher is stalled – it needs to be done precisely after retirement is stalled in the previous step to avoid a jump or a branch instruction resuming the fetcher after it was stalled.

4. Flush entries from ROB, making sure physical register IDs that those entries contained are inserted back into free physical registers FIFO.

5. Jump to the ISR (resuming the fetcher in the process), address of which is contained in the `mtvec` CSR and resume retirement.

6. Wait for interrupt return instruction `mret` (return from machine mode interrupt). Handling this kind of instruction is delegated to a specialized functional unit that calls an `iret` method in the interrupt coordinator that triggers transition to the next step.

7. Jump back to the interrupted instruction stream by redirecting the PC to value stored previously in `mepc`.

The implementation disallowed nested interrupts (that are allowed by RISC-V specification), but this was meant to be a temporary measure to simplify testing until a proper mechanism involving an `mstatus` (machine status register) CSR was devised.

One drawback of this approach is the necessity to implement `clear` method (for clearing the internal state) in all relevant parts of the core – FIFOs, reservation stations, functional units, etc. These necessarily *conflict* with other methods (that read or manipulate the internal state) as `clear` has to take priority so it's marked as such (e.g. when both `write` and `clear` methods need to be called in some component in the same clock cycle, Transactron's scheduler will always select `clear` to be called in that clock cycle). This in turn complicates the generated Transactron scheduling logic. Moreover, due to a centralized approach to clearing (interrupt coordinator calling `clear` in all parts of the core) this turns the circuit into a graph with lots of components connected to a single node (the interrupt coordinator). Both make the Transactron scheduling logic have worse timing parameters in the synthesized core.

Chapter 6

Summary and future work

Implementing an out-of-order core is an example of practice being harder than the theory, as many traps lurk in the shadows for an aspiring microarchitectural designer. Examples given in this work include striking bugs such as accidentally overwriting operands of instructions, not freeing dynamically allocated resources (which ultimately leads to the core halting) or more subtle ones, like accidentally "leaking" allocated resources or spontaneously jumping to an incorrect place in memory in rare circumstances when servicing an interrupt. Mandatory test writing was thus paramount to the success of the project, as these gave the development team confidence in their implementation and very quickly highlighted bugs during the development.

The core is also a great example of learning through implementation. The the development team learned many nontrivial details and properties of such system and where potential bugs could be, much more than could be hoped for by just reading the literature available on the topic.

Ultimate goal of the project is to run Linux and Mimiker operating systems on the *Coreblocks* core. There is much to be done for it to be possible – currently the core lacks an interrupt controller, support for atomics (**A** extension), virtual memory and privilege levels – all required for running any modern OS.

Another avenue is improving the core's performance. Obvious point for improvement is not stalling on branches but instead using a branch predictor and executing instruction speculatively. The load-store unit can also be improved to support multiple loads/stores at once and reorder them. Another larger task in this category is making the core superscalar by fetching and retiring multiple instructions at once. Data structures would also need to be updated to handle multiple operations on them at the same time.

Finally, while not a priority, floating point extensions are yet to be implemented. These are respectively the **F** extension for 32-bit floats and **D** extension for 64-bit floats.

In this work we've shown how an out-of-order core works and described the implementation of and author's contributions to *Coreblocks* – an out-of-order RISC-V being developed at the University of Wrocław. We hope that this work will be a good introduction to the *Coreblocks* project and serve future students in their efforts to further their knowledge in modern processor design.

Bibliography

[1] Marton Bognar, Job Noorman, and Frank Piessens. Proteus: An extensible RISC-V core for hardware extensions. In *RISC-V Summit Europe '23*, June 2023.

[2] Michał Błaszczyk. Port of Mimiker operating system for RISC-V architecture. Bachelor's thesis, University of Wrocław, Faculty of Mathematics and Computer Science, Institute of Computer Science, February 2022. Available at: `https://ii.uni.wroc.pl/media/uploads/2022/11/18/baszczyk-micha-praca.pdf` Accessed: 2024-06-11.

[3] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. Technical report, Massachusetts Institute of Technology, 1974.

[4] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. A survey of different approaches for overcoming the processor-memory bottleneck. *International Journal of Computer Science & Information Technology (IJCSIT)*, 9(2), April 2017.

[5] MIT CSAIL's Computation Structures Group. RiscyOO design document. `https://github.com/sizhuo-zhang/RiscyOO_design_doc`. Accessed: 2024-06-11.

[6] MIT CSAIL's Computation Structures Group. RiscyOO: RISC-V out-of-order processors. `https://github.com/csail-csg/riscy-OOO`. Accessed: 2024-06-11.

[7] Institute of Computing Technology, Chinese Academy of Sciences and Peng Cheng Laboratory. XiangShan - open-source high-performance RISC-V processor project. `https://github.com/OpenXiangShan/XiangShan`. Accessed: 2024-06-11.

[8] RISC-V International. The RISC-V Instruction Set Manual volume I (Unprivileged Architecture). `https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf`. Accessed: 2024-06-11.

[9] RISC-V International. The RISC-V Instruction Set Manual volume II (Privileged Architecture). `https://github.com/riscv/riscv-isa-manual/releases/download/20240411/priv-isa-asciidoc.pdf`. Accessed: 2024-06-11.

[10] RISC-V International. RVWMO memory consistency model. `https://five-embeddev.com/riscv-isa-manual/latest/rvwmo.html`. Accessed: 2024-06-11.

[11] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237, 1996.

[12] Arm Ltd. Arm glossary - Instruction Set Architecture (ISA). `https://www.arm.com/glossary/isa`. Accessed: 2024-06-11.

[13] Marek Materzok. Pull request #370 - implementation of precommit method. `https://github.com/kuznia-rdzeni/coreblocks/pull/370`. Accessed: 2024-06-11.

[14] O. Mutlu, Hyesoon Kim, and Y.N. Patt. Address-value delta (AVD) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12 pp.–244, 2005.

[15] The Regents of the University of California. Berkeley Out-of-Order Machine documentation. `https://docs.boom-core.org/en/latest/`. Accessed: 2024-06-11.

[16] The Regents of the University of California. RISC-V BOOM - the Berkeley Out-of-Order RISC-V processor. `https://boom-core.org`. Accessed: 2024-06-11.

[17] University of Wrocław. Coreblocks GitHub repository. `https://github.com/kuznia-rdzeni/coreblocks/`. Accessed: 2024-06-11.

[18] University of Wrocław. Coreblocks project README. `https://github.com/kuznia-rdzeni/coreblocks/blob/6db5cf098633462f109b8b5fb5406f69c0de908b/README.md`. Accessed: 2024-06-11.

[19] University of Wrocław. The Mimiker project. `https://mimiker.ii.uni.wroc.pl`. Accessed: 2024-06-11.

[20] University of Wrocław. Transactron library documentation. `https://kuznia-rdzeni.github.io/coreblocks/transactions.html`. Accessed: 2024-06-11.

[21] OpenCores Organization. *Wishbone B4 WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores.* Available at: `https://cdn.opencores.org/downloads/wbspec_b4.pdf` Accessed: 2024-06-11.

[22] Amaranth project contributors. Amaranth language documentation. `https://amaranth-lang.org/docs/amaranth/latest/`. Accessed: 2024-06-11.

[23] Surya Raj. Awesome RISC-V resources. `https://github.com/suryakantamangaraj/AwesomeRISC-VResources`. Accessed: 2024-06-11.

[24] André Seznec. TAGE-SC-L branch predictors. In *Proceedings of the 4th Championship on Branch Prediction*, June 2014. Available at: `https://jilp.org/cbp2014/paper/AndreSeznec.pdf` Accessed: 2024-06-11.

[25] John Paul Shen and Mikko Herman Lipasti. *Modern Processor Design*. Waveland Press, 2005.

[26] Robert Marco Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), January 1967.

[27] Gabriele Tripi. ApogeoRV RISC-V - high-performance and highly customizable CPU core. `https://github.com/GabbedT/ApogeoRV`. Accessed: 2024-06-11.

[28] Jakub Urbańczyk. Pull request #361 - fix for immediate truncation bug in jump-branch functional unit. `https://github.com/kuznia-rdzeni/coreblocks/pull/361`. Accessed: 2024-06-11.

[29] Stefan Wallentowitz. RISC-V python software model. `https://github.com/wallento/riscv-python-model`. Accessed: 2024-06-11.