POLITECNICO
DI TORINO

Master's Degree Course in Electronic Engineering

Master's Thesis

Design of the execution pipeline for LEN5, a RISC-V Out-of-Order processor

Supervisor:
Prof. Maurizio Martina

Candidate:
Michele Caon

Academic year 2018-2019

Abstract

Out-of-order execution has become mandatory to achieve good performance in modern processors. At the same time, modularity and customization of the execution pipeline are welcome features when it comes to supporting modern application-specific operations. From this perspective, RISC-V is probably the most promising open Instruction Set Architecture (ISA) at the moment. For this reason, the RISC-V Foundation can count on the support of many industry-leading companies, and the specification is evolving to support all the features that modern applications require.

The design of the back-end of the *LEN5* RISC-V processor detailed in this document aims at searching the design space for new solutions that accommodate both modularity and possibly performance. To achieve this, the execution pipeline architecture is based on Tomasulo's approach to dynamic scheduling[1], expanding it to support speculation and precise exceptions. The handshake-based instruction flow offers the possibility to easily add new custom execution units and the support for additional ISA extensions.

Chapter 1 briefly introduces RISC-V ISA while explaining the main motivations that lead to the *LEN5* core implementation. Chapter 2 reports the basic concepts of instruction scheduling on which the execution pipeline is based, as well as the reasons why the Tomasulo's algorithm was chosen over scoreboarding. Chapter 3 is the core of this document: it contains a detailed description of each basic block in the execution pipeline of *LEN5* together with schematics and diagrams. Chapter 4 briefly discuss the testing methodology and some of the obtained results, while chapter 5 draws the conclusions about this project. Finally, Section 5.3 lists and motivates some possible improvements that could be applied to the *LEN5* core in a possible future design stage. Appendix A contains additional information about the System Verilog description of the core.

[1]As presented in: John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 9780128119051.

Acknowledgements

I would like to thank Andrea Giannini for this great opportunity.

I also wish to express my gratitude to my friends and colleagues Matteo and Marco for their support besides their collaboration in this project.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction to LEN5

LEN5 is a RISC-V processor implementing in-order issue and Out of Order (OoO) execution developed at Politecnico di Torino during the master thesis projects by Marco Andorno [3], Matteo Perotti [10] and Michele Caon, the author of this document (i.e. myself). Its code is available under the open source *Solderpad Hardware Licence Version 2.0*[1] [12] on GitHub.

1.1 Motivation

Over the past 50 years, the definition and development of computer ISAs have been a prerogative of a few names among the industry-leading companies. Intel's x86 has been by far the most successful architecture from its introduction with the Intel 8086 processor in 1978. Its instruction set grew by more than one order of magnitude and continue to do so, in what is defined as an *incremental* approach to ISA extension. With any new extension, all the previous instructions must be implemented, even if outdated for modern applications or error-prone. The growing ratio increased the most over the last decade, as a consequence of the slowing down of Moore's Law: x86 has relied on Single Instruction Multiple Data (SIMD) instructions to increase Instruction Level Parallelism (ILP) and achieve the target performance improvement. With the current limits in technological innovation, the importance of architectural optimization has become an important aspect in microprocessor design, and even x86 processors are moving towards a simplification of the instruction set, where complex instructions are translated in hardware into simpler ones.

Still, the hardware to handle such a complex ISA has practically been inaccessible to anyone outside the leading companies in the microprocessor market, meaning that university students were forced to study this topic on very old and outdated

[1] based on the *Apache Licence Version 2* [5]

processor architecture. Most important, x86 is a proprietary ISA, that leaves no space for improvement in the academic world.

RISC-V was born in university (at UC Berkley in 2011) to overcome all these downsides while following a completely different approach, shaped on the requirements of modern computer applications. This modern ISA aims at becoming a candidate as a universal ISA in the next years, and the RISC-V Foundation [4] has found the support of many companies, including some of those traditionally associated to x86 architectures. Despite this, RISC-V is and is going to remain a *open* ISA. Its *modular* approach differs offers the possibility to employ it in a very wide range of applications: from very simple microcontrollers to very complex, OoO multi-core systems used in High Performance Computing (HPC) applications.

LEN5 has offered the opportunity to explore the design space of microprocessor architecture while working with a 'real world' ISA instruction set. The aim of this project is to create a starting point for the introduction of RISC-V in our academic context while trying to deal with design challenges that were only scratched during the Bachelor's and Master's degree programs. *LEN5* represents also the incarnation of the personal interests grown more and more during the past years, as well as of the intents for possible future development projects.

LEN5 follows the same modularity that is intrinsic in RISC-V, and offers many research opportunities to the students that might be interested in every single topic related to digital design in general.

In the second place, this project was also a chance to improve all the skills a digital hardware designer should have. For this reason, the System Verilog hardware description language was employed, being it the standard in most of the design and testing tools used to model and synthesize complex digital systems.

The *LEN5* design was divided into three main parts: the frontend, the execution pipeline, and the memory system, as shown in fig. 1.1. This document is about the execution pipeline. The following sections briefly introduce each of those parts.

1.2 Frontend

The frontend of *LEN5* was developed by Marco Andorno [3]. Besides implementing the fetch operations and the interface with the *instruction cache*, it provides the support for speculation by means of a *gshare* branch predictor that produces both the predicted branch outcome and the predicted target address.

1.3 Execution pipeline

The execution pipeline represents the backend of the processor, that is the part responsible for the actual execution of the instructions. An extended version of

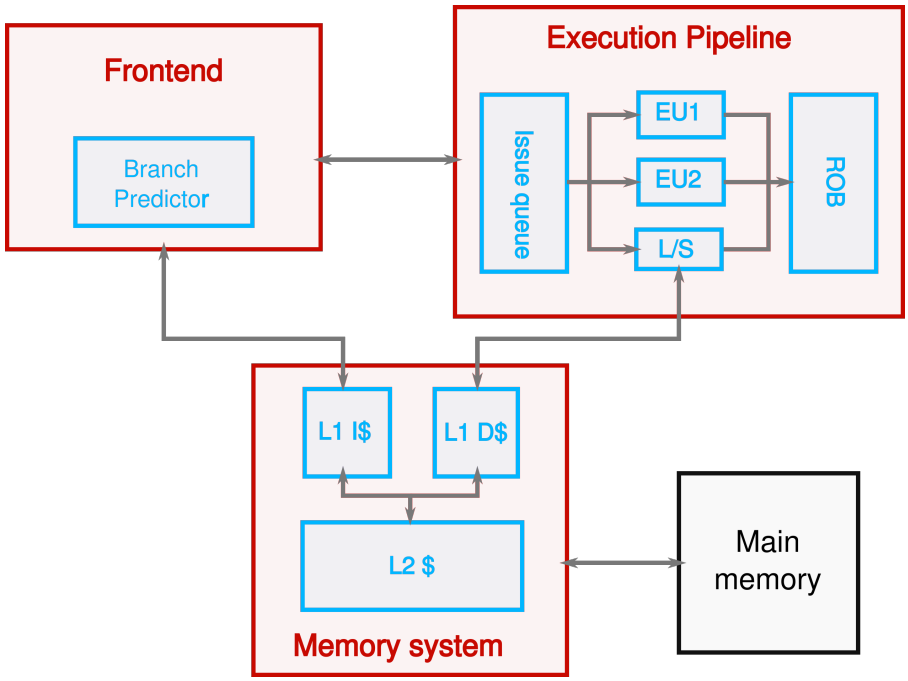


Figure 1.1: The three main parts of *LEN5*. The content of each unit is not accurate. For the detailed schematics refer to [3] (frontend), [10] (memory system) and chapter 3 of this document (execution pipeline).

Tomasulo's algorithm for dynamic scheduling was defined and implemented to support Out of Order Execution (OoOE), speculation and *precise exceptions* while adopting modularity and customization as main design principles. The next chapter will focus on the theory (chapter 2) and the implementation (chapter 3) of this part of *LEN5*.

1.4 Memory system

The memory system of *LEN5* was developed by Matteo Perotti [10]. It is responsible for the execution of *load* and *store* instructions, as well as the support for Virtual Memory (VM) with hardware address translation. The organization of the *cache system* follows a modified Harvard memory scheme, with two distinct *instruction L1* and *data* L1 caches and a unified L2 cache, all sharing the same address space.

Chapter 2

Execution pipeline architecture

Before entering the core chapter of this document, where the *LEN5* design is analyzed in-depth, some key concepts must be discussed. This chapter collects some basic notions of instruction execution in an OoO processor. The major design choices are discussed and compared to other possible solutions. In particular, the following sections focus on instruction renaming and reordering according to Tomasulo's approach to dynamic scheduling.

2.1 Basics

The performance of a microprocessor is defined by many different aspects, and each of these may have a different weight depending on the cost function that is used. Among the others, power is less than ever playing a secondary role. However, in this first version of *LEN5*, the focus has been mainly on architectural choices, while power-saving hasn't been ignored, as often pointed out in chapter 3. For what concerns processing performance, a definition that is independent on implementation choices and technological parameters should be used in principle. From the user perspective what matters is the *CPU time* required by the processor to complete a given task. The definition of performance given by Patterson and Hennessy in [7, sec. 1.6] was chosen for this purpose:

$$Performance = \frac{1}{CPU\ execution\ time} \tag{2.1}$$

Where the "*CPU execution time* [...] is the time the CPU spends computing for this task and does not include the time waiting for I/O or running other programs".

From an architect's point of view, this translates to increasing the number of instructions that the Central Processing Unit (CPU) can execute in a time unit. To

make this definition independent on technological parameters the concept of *time*, measured in *seconds*, can be replaced by the concepts of *latency*. In a synchronous digital system, the latency is the number of clock cycles (time-discrete units) elapsed from the beginning of the processing of some *information*. An additional step is to define a *task* as a sequence of instructions, which is likely the case for a microprocessor. So, the latency can be defined as the number of cycles required to complete the execution of an instruction. If instructions were executed strictly in sequential order, *increasing the performance* would simply mean *decreasing the average instruction latency*. However, modern microprocessors always execute more than one instruction at a time. This is called ILP. As an example, even a simple in-order[1] microprocessor has many pipeline stages, resulting in many instructions being processed at the same time by the different pipeline stages, while the latency remains constant in a first approximation. Taking a step back into the physical world, this goes in the direction of increasing the *throughput*, or the amount of information being processed in a time unit. The trend followed in processor design in the last decades have seen *throughput* being prioritized over *latency* most of the times [9]. In other words, in modern computer applications, the number of instructions that are completed in a time unit is usually more important than the time required to complete a single instruction or even a single task. For this reason the Instructions Per Cycle (IPC) is one of the most important metrics to evaluate processor performance nowadays. In practice, the *average IPC* is used to evaluate performance: different instructions have different latency, so the number of instructions being executed simultaneously by the processor can be different based on the code being executed in every moment.

Before briefly introducing the techniques used to exposing ILP, it is necessary to define what are the main steps an instruction goes through during its execution. The *fetch-decode-execute* loop is expanded to take into account also OoO and multiple issue processor architectures. The following terms will be used in all this document with the meaning reported below.

- **Instruction fetch:** the instruction contained in the memory location pointed by the Program Counter (PC) is read from the instruction memory by the *fetch unit* and sent to the *issue queue*.

- **Instruction issue:** the first instruction of the *issue queue* is decoded and sent to the assigned functional unit of the execution pipeline, awaiting to be executed[2]. This phase is sometimes referred to as **dispatch** in literature.

[1] In an *in-order* processors the order in which instructions are executed is the same in which they appear in the program code (i.e. they are executed *in [program] order*.)

[2] In *LEN5* the instruction is actually sent to the *ROB* and to the *RS* associated to the assigned functional unit at the same time. A detailed description of this phase can be found in section 3.6.

- **Operands fetch:** the instruction operands are read from the source registers. Only when all the required operands have been fetched, the instruction can proceed to the execution stage.
- **Beginning of execution:** the instruction is selected for execution by the functional unit. In literature, this phase is also called *execute* or *issue*.
- **Execution completion:** the instruction result is produced by the functional unit and written to a buffer. In in-order processors, the result is directly written to the *Register File (RF)* or to the memory, and this phase reduces to the *instruction commit*.
- **Write result:** In some OoOE pipelines where a Reorder Buffer is used, an additional step may be required to distinguish the writing of the result into the Reorder Buffer (ROB) from the execution completion and the commit stages. *LEN5* falls in this category, as it will be explained in section 2.4.
- **Instruction commit:** the instruction result is written to the *RF* or to the memory.

Notice that OoOE, which means that instructions **begin** their execution out-of-order, implies OoO completion. If the functional units further reorder instructions internally, the completion order may differ from the beginning of execution order.

2.2 Instruction level parallelism

ILP can be exploited in different ways, all of which aim at executing the largest possible number of instructions at the same time. To do this many instructions may be issued in the same cycle as in *multiple issue* processors, or a new instruction could be issued before the previous instruction has committed, as it is the case in *pipelined processors*. Other integrated circuits like Graphics Processing Units (GPUs) can implement *SIMD instructions* that are performed on a certain set of data instead of a single one. In general, all these techniques allow exploiting ILP to a level of parallelism that is given by:

$$P = \frac{number\ of\ completed\ instructions}{number\ of\ cycles\ required}$$

However, while exploiting ILP can potentially lead performance improvements in the order of P, it is not always possible to do so and when it is, it doesn't come at no cost. It is well known [15] that even using optimum techniques to expose ILP,

Here, the important concept is that this phase doesn't necessarily correspond to the beginning of the execution of the instruction.

the value of P is on average between 5 and 7. The biggest limit are *dependences* between subsequent instructions. It is very common in code that an instruction depends on the result of a previous instruction: the result of the first instruction could be used as an operand for a subsequent arithmetic operation or as the base address for a memory access. So the amount of parallelism between the instructions in a program in not arbitrary high and it is not always possible to know it a priori. This means that techniques based only on code transformations at compile-time like *loop unrolling* do certainly help but might not be sufficient. Exploiting ILP in hardware can detect if instructions can be executed in parallel dynamically. Doing so requires *dependences* to be detected at runtime to avoid hazards. There are four main types of dependences: *structural dependences*, *control dependences*, *name dependences* and *data dependences*.

Structural dependences Even if many instructions could be executed in parallel, doing so requires a sufficient number of functional units to be available. If the functional unit assigned to an instruction is still busy processing an older instruction a structural hazard occurs. In this case, the only solution is to stall the execution pipeline until the functional unit becomes free and the new instruction can be accepted.

Control dependences Control hazards are caused by branch instructions. The code between two branches is called a *basic block*. A basic block should only be executed if the current *trace* of the program includes the leading branch. A *trace* is the set of basic blocks that are executed if a certain progression of branch conditions takes place during the execution of a program. That is, all the code that follows a branch depends... at least on one branch. This is true for almost every code executed by a processor. Though software techniques to predict the correct *traces* and recover from wrong predictions exist, hardware speculation has been preferred over the last decades. In deeply pipelined microprocessors, where the amount of concurrently executed instructions is very high, recovering from wrong speculation about the branch outcome (*misprediction*) has a very high cost. For this reason, very sophisticated branch predictors have been developed to minimize the misprediction ratio [11].

Name dependences An instruction might write a register or a memory location accessed also by a previous instruction before the read operation was actually performed (*antidependence* or *WAR*[3] *hazard*). Or else two consecutive instructions might write the same destination register or memory location, so that the order in which they complete their execution changes the output (*output dependence* or

[3] *Write After Read*

WAW[4] *hazard*). Notice that OoOE intrinsically leads to WAR and WAW hazards. Name dependences don't imply an actual data transfer between the instructions. Therefore, both the types of name dependences can be resolved with *register renaming*, which consists in having more physical registers than logical ones. When a new instruction is issued, its logical source and destination registers are used to access a *renaming table* that tracks which registers are being used by other in-flight instructions. If a name dependence with a previous instruction is detected on the destination register, a free physical register is assigned to the issuing instruction and the renaming table is updated. All following reference to that logical register will be redirected to the assigned physical one so the value produced by the most recent instruction writing a logical register is correctly fetched as a source operand for the following instructions. As will be explained in section 2.3, *LEN5* uses a different solution for renaming, where each issued instruction is mapped to an entry of the *ROB*.

Data dependences These are also called *true data dependences* as opposite to name dependences. There is a data dependence every time an instruction uses as an operand a result from a previous instruction (*RAW*[5] *hazard*). In this case, the only solution is to wait for the previous instruction to have its result computed by the assigned functional unit. *Forwarding* can be used to partially or totally hide the latency introduced in this case: the result of the previous instruction can be sent to the following one bypassing one or more pipeline stages. As explained in section 2.3 *LEN5* uses a different approach that automatically forwards results as soon as they are produced by the functional units.

Besides pipeline, multiple issue and SIMD instructions, also dynamic scheduling can be used to expose and exploit ILP.

2.3 Dynamic Scheduling

Dynamic scheduling consists in reordering instructions so most of the dependences between them can be hidden by the execution of other instructions. The same can be done to overcome long latencies caused by *data cache* misses: instead of waiting for the current *load* instruction to complete, another instruction can begin its execution if it doesn't depend on the loaded value. A processor implementing this kind of scheduling has an OoOE execution pipeline, which imply OoO beginning of execution and OoO execution completion, as mentioned in section 2.2. The execution pipeline of *LEN5* implements dynamic scheduling using Tomasulo's approach to register renaming, and extends it to support speculation and precise exceptions.

[4] *Write After Write*

[5] *Read After Write*

However, this is not the only approach to dynamic scheduling: scoreboarding has been the most widely used for many decades. In order to understand the advantage of Tomasulo's algorithm with respect to scoreboarding, both approaches are briefly discussed in the following sections.

2.3.1 Scoreboarding

Scoreboarding is a book-keeping technique that allows instructions that are issued in-order to execute OoO as soon as their operands are ready (RAW hazards) and there are no true data dependences among them. Also, the required functional unit must be available to execute the new instruction. The most representative feature from its first employment in CDC 6600 mainframe computer is the *scoreboard*. It tracks the WAR and WAW dependences among in-flight instructions, allowing each instruction to proceed to the next execution stage only if all existing dependences have resolved.

- WAR hazards are solved preventing an instruction from writing its result to the destination register or memory location (i.e. committing) until all previous instructions have read the value from that register.

- WAW hazards are solved preventing an instruction that shows an output dependence with some older instruction to commit until that instruction has committed. This means that if an output dependence is detected, the commit phase is performed in-order, at least for those instructions.

RAW hazards are solved by stalling the issued instruction until its operands become available, according to the information in the *scoreboard*. Operands can be fetched from source registers only during the issue phase. This step is often referred to as **read operands** stage.

To achieve the correct flow of instructions, the scoreboard must keep three different kinds of information:

- The **instruction status:** the stage of execution a certain instruction is in at the moment.

- The **functional unit status:** the instruction to be performed in each functional unit, the destination register where the result must be stored and whether the operands are ready or not.

- The **register status:** the functional unit that will write each destination register.

The data is organized in three different tables that are checked and updated according to the progression of each instruction in the execution pipeline. An

example[6] of the content of the three tables is provided in tables 2.1a to 2.1c.

The fact that all the tables must be read and updated in every step of the execution and for every instruction makes scoreboarding an expensive solution. Besides this, if a WAR hazard occurs, the instruction is prevented to write the register file or the memory until all previous instructions have accessed the contained information. A solution to this problem consists in using a Reorder Buffer where to save instruction results before they are copied to the destination register or memory location. However, this makes a scoreboarding implementation even more complex. The situation gets worse if renaming is used to solve WAW hazards too since additional tables must be accessed during the issue stage. Another disadvantage is that every new functional unit that is added to the execution pipeline requires a new entry in the functional unit status table, as shown in table 2.1b.

In summary, scoreboarding has the advantage of permitting dynamic scheduling by stalling the pipeline if some dependence is detected among instructions. However, it has some disadvantages that are not negligible in most applications:

- All the necessary information is kept in a single structure (the *scoreboard*) containing different tables that must be accessed and updated in every cycle by every in-flight instruction, thus requiring a very complex and potentially slow control logic.
- In order to avoid stalling the pipeline when WAR and WAW hazards are detected, the scoreboard must be complicated to include register renaming and ROB management.
- The number of entries in a functional unit status table is directly proportional to the number of functional units in the execution pipeline, regardless of their type.
- Speculation is hard to introduce since a mispredicted instruction would affect the *scoreboard*, making it hard to recover from such a situation while still maintaining the results from instructions that are older than the mispredicted one.

Because of all the downsides that come with scoreboarding, another approach was chosen to design the execution pipeline of *LEN5*: Tomasulo's algorithm, which is introduced in section 2.3.2.

2.3.2 Tomasulo's Algorithm for Dynamic Scheduling

Tomasulo's approach to dynamic scheduling was originally introduced in 1967 to expose ILP in the floating-point execution pipeline of the IBM System/360, Model

[6] In this and following examples, RISC-V assembly code will be used as it appears in the reference card in [8].

Instruction	Execution stage
`ld x1, 0(x3)`	Commit
`ld x2, 8(x3)`	Begin of execution
`add x3, x1, x2`	Operands fetch
`addi x2, x3, -1`	Operands fetch

(a) Instruction status table content.

f.u.	Op.	rd	rs1	rs2	src f.u. 1	src f.u. 2	rs1 ready	rs2 ready
load-store	load	x2	x3	imm.	-	-	yes	yes
ALU 1	add	x3	x1	x2	-	load-store	yes	no
ALU 2	add	x2	x3	imm.	ALU 1	-	no	yes
MULT	-	-	-	-	-	-	-	-
branch	-	-	-	-	-	-	-	-

(b) Functional unit status table content.

Register	Source f.u.
x1	-
x2	load-store
x3	ALU 1
...	...
x31	-

(c) Scoreboarding example: register status table content.

Table 2.1: Scoreboarding example.

91 [13], where multiple functional units were available. Another main motivation was overcoming the need for implementation-specific compile-time optimization to achieve high performance on different chips. As discussed in section 2.1, modern high-performance microprocessors try to maximize the number of instructions that can be executed in parallel. To do this the first requirement is having a sufficient number of execution units, meaning many Arithmetic and Logic Units (ALUs), Floating-Point Units (FPUs), and even multiple load-store units. Tomasulo's algorithm meets all the requirements of this kind of systems while enabling the possibility to use pipelined and even variable latency functional units without any changes to the original structure. Notice that the original algorithm was introduced before the presence of caches in microprocessors. The fact that long and variable latencies were natively supported was one of the main reasons for it to be largely adopted also in modern processors.

Despite all those features, the control logic remains quite simple, because hazard detection and register renaming are handled automatically. This section shows the main differences and advantages of this approach compared to scoreboarding, while the next one will briefly introduce how this algorithm is implemented in *LEN5*.

First, let us describe the fundamental components of a dynamically scheduled execution pipeline implementing Tomasulo's algorithm.

Reservation stations The main difference between Tomasulo's approach and scoreboarding is that the former employs several buffers associated with every functional unit. These buffers are called *Reservation Stations* and hold issued instructions during the operand fetch and the execution phases. As soon as a functional unit becomes available, an instruction in the Reservation Station (RS) whose operands are available is selected to begin its execution. When the functional unit produces the result (execution completion) it is saved in the same RS entry of the instruction. From here, the result is sent to the Common Data Bus. An exception in this behavior is the load-store queue: here instructions are executed in program order. This means that the load-store queue is a unified buffer that holds both *load* and *store* instructions and it's compiled as a First In First Out (FIFO) buffer: instructions are pushed in program order during issue and executed in the same order. Only the head entry of the load-store queue, or the oldest one, can perform the memory access if the operands to compute the memory access are ready. The execution of a *store* corresponds to its commit, since no result to save in a destination register is produced. A load that has performed the memory access can proceed writing the leaded value on the Common Data Bus (CDB) as any other instruction. The information contained in a RS is shown in table 2.2.

RS	**valid**	**rs1_value**	**rs2_value**	**rs1_srcRS**	**rs2_srcRS**	**res_ready**
0	yes	15	-	0	RS[2]	no
1	no	-	-	-	-	no
2	yes	7	3	0	0	yes
...	...	...	...	...	...	...
m-1	yes	-	-	RS[2]	RS[0]	no

Table 2.2: Reservation station data structure. In this example, the third entry of the reservation station has already completed its execution since it had both the operands available.

Common data bus The Common Data Bus is a bus shared by all the RSs. When the result of an instruction has been produced by a functional unit and saved in the associated RS, it is copied on the CDB as soon as this becomes available. The result is broadcast to every RS and to the RF. If an in-flight instruction needs the result of the instruction that's writing on the CDB, it copies it from the CDB to its RS entry. To do so each in-flight instruction must know which entry in which RS will produce the result that it needs. Also, the result is written to the RF (commit) only if the RS entry that wrote it on the CDB is the last that should write the corresponding destination register to avoid WAW hazards. Both this information

are available in the *register status* data structure. Notice that this implements *register renaming* automatically and for every instruction. Each source register is renamed after the RS that will produce that value whenever it is not available in the RF yet. This is one of the main features of Tomasulo's algorithm.

Register status When an instruction is issued, it accesses the register status data structure to know if its source operands are available in the RF or which RS entry will write them. Therefore, the register status structure has one entry for each register in the RF. Also, the reservation station entry assigned to the issuing instruction is recorded next to the destination register of the instruction to know once again which in-flight instruction will write that register. The register status is accessed again when an instruction writes its result on the CDB to know if the result should be stored to the RF, as described in the previous paragraph. The register status data structure is shown in table 2.3.

RF	**RF.srcRS**
`x1`	RS[3]
`x2`	0
`x3`	RS[5]
...	...
`x31`	RS[i]

Table 2.3: Register status data structure. In this example, register `x1` will be written by the RS entry 3, while register `x2` is already available in the RF.

These are the only components of the simplest execution pipeline implementing Tomasulo's algorithm. In this version, speculation and precise exceptions are not supported, and control dependences are avoided stalling the issue stage. In fact, after this phase, it's practically impossible to reconstruct the program order to know which instructions are valid and which are not. The next section explains how this limit is overcome in *LEN5*. Before that, the pseudo-code of the algorithm for arithmetic instructions is reported in algorithms 2.1 to 2.5, split into the main execution steps and the main loop. Notice that for simplicity in this example there is no difference between the *begin of execution* and the *execution completion* phases, as if the functional unit was combinational, thus producing the result in the same cycle in which the operands become available.

2.4 LEN5 implementation of Tomasulo's Algorithm

As mentioned in section 2.2, speculation is almost mandatory in modern processors to achieve good performance. The frontend of *LEN5* uses a *gshare* branch predictor

Algorithm 2.1 Tomasulo's algorithm: **issue** stage

```
procedure ISSUE(instr)
    q ← assigned RS entry
    while (!(q is empty)) do
        wait                                          ▷ wait for q to become free
    end while
    RS[q].valid ← yes                                 ▷ the fetched instr. is inserted in q
    regstat[rd].srcRS ← q                             ▷ the dest. register rd will be written by q
    return q
end procedure
```

Algorithm 2.2 Tomasulo's algorithm: **fetch operands** stage

```
procedure OPERANDFETCH(q)
    if (regstat[rs1].srcRS ≠ 0) then
        RS[q].rs1_srcRS ← regstat[rs1].srcRS          ▷ save the index of that RS entry
    else                                              ▷ rs1 already available in the RF
        RS[q].rs1_value ← RF[rs1]                     ▷ fetch the value of rs1
        RS[q].rs1_srcRS ← 0
    end if
    if (regstat[rs2].srcRS ≠ 0) then
        RS[q].rs2_srcRS ← regstat[rs2].srcRS          ▷ save the index of that RS entry
    else                                              ▷ rs2 already available in the RF
        RS[q].rs2_value ← RF[rs2]                     ▷ fetch the value of rs2
        RS[q].rs2_srcRS ← 0
    end if
end procedure
```

Algorithm 2.3 Tomasulo's algorithm: **execute** stage

```
procedure EXECUTE(q)
    if (RS[q].rs1_srcRS = 0 and RS[q].rs2_srcRS = 0) then      ▷ both operands
available
        result ← RS[q].rs1_value <op.> RS[q].rs2_value         ▷ compute result
        RS[q].res_ready ← yes
        RS[q].res ← result
    end if
end procedure
```

Algorithm 2.4 Tomasulo's algorithm: **commit** stage

```
procedure COMMIT(q)
    for (∀x) do                                          ▷ for each register x
        if (regstat[x].srcRS = q) then
            RF[x] ← RS[q].res                            ▷ copy the result to the RF
            regstat[x].srcRS ← 0                         ▷ mark the register as available
        end if
    end for
    for (∀y) do                                          ▷ for each RS y
        if (RS[y].rs1_srcRS = q) then
            RS[y].rs1_value ← RS[q].res                  ▷ copy the result to y
            RS[y].rs1_srcRS ← 0                          ▷ mark operand as ready
        end if
        if (RS[y].rs2_srcRS = q) then
            RS[y].rs2_value ← RS[q].res                  ▷ copy the result to y
            RS[y].rs2_srcRS ← 0                          ▷ mark operand as ready
        end if
    end for
    RS[q].valid ← no                                     ▷ free q since the instr. has committed
end procedure
```

Algorithm 2.5 Tomasulo's algorithm: main loop

```
while (1) do                                             ▷ in every cycle
    if (∃ new_instr) then                                ▷ fetched instr. available
        r ← ISSUE(new_instr)
        OPERANDFETCH(r)
    end if
    for (∀i) do                                          ▷ for all in-flight instructions
        if (RS[i].valid and !RS[i].res_ready) then
            EXECUTE(i)                                   ▷ compute result
        else if (RS[i].res_ready and CDB available) then
            COMMIT(i)                                    ▷ write result to RF and RSs
        end if
    end for
end while
```

that speculates both on the branch outcome and on the branch target address[7]. Using speculation implies providing a way to recover from misprediction: in this case, speculative instructions must be flushed before they modify the *processor state*. In other words, speculative instructions cannot write their results to the RF or the memory until they are confirmed to be non-speculative. The same is true when an exception is raised: only exceptions coming from instruction that would execute if speculation was not allowed must be processed.

Besides speculation, there are instructions that modify the processor Control Status Registers (CSRs) instead of writing data to the RF or memory. If these instructions were allowed to be executed OoO, some instructions would execute before the CSRs are actually updated, while others could execute with the updated value even if they preceded the CSR instruction if program order was maintained.

Both these issues can be overcome forcing the *commit* phase to be carried out *in program order*. However, since no track of the original order of instructions is maintained in the execution pipeline with the original version of Tomasulo's algorithm, some modifications and some additional components are required.

Notice that following an *in-order commit* scheme has another great advantage: interrupts and exceptions can be processed *in-order* during commit, and so in program order. After the exception or interrupt has been processed, the next instruction can be replayed without losing data or coherence with the sequential execution of the program. This kind of exception handling mechanism follows a *precise exception* model. In opposite, the original algorithm follows an *imprecise exception* model. When instructions are allowed to complete *and* commit out of order, interrupts and exceptions can be processed only during the execution completion phase. Therefore, since no information about the original instruction order is maintained, there is no way to know if all instructions preceding the exception or interrupt have completed or if the raised exception is, in fact, the first that would have occurred if the code was executed in-order.

Usually, when an exception or interrupt is raised, the execution pipeline is flushed and the associated *service routine* is loaded. When an imprecise exception model is implemented, all the instructions are flushed unconditionally, regardless of whether they preceded or followed the offending instruction in program order. As a consequence, exception handling, which is usually performed by the Operating System (OS) becomes more difficult. While RISC-V specification doesn't require a precise exception model, this was implemented in *LEN5*, since it comes at almost no cost with the extended version of Tomasulo's algorithm.

Now that all the motivations for a new, more powerful execution algorithm have been explained, the changes that must be applied to the original algorithm and the original architecture will be discussed.

[7]More on this in the separate document [3].

First of all, let us introduce the new key component whose name has already been mentioned a few times in the previous sections: the *ROB*.

2.4.1 Reorder Buffer

The ROB is what allows instruction to commit in order, meeting the most important requirement of speculation, CSR instructions and precise exceptions. This data structure is somewhat similar to the load-store queue mentioned in section 2.3.2, since it is compiled in order (as in a FIFO) and contains information about all the in-flight instructions. In some implementations, the store buffer can effectively be subsumed by the ROB, as suggested in [6]. In *LEN5*, both the load and the store queue are implemented as separate queues for reasons that are explained in section 3.7.

The changes that the presence of the ROB introduces in the execution algorithm are discussed in the next paragraphs. As a consequence of them, also the content of each of the data structures introduced in section 2.3.2 must be slightly modified and extended with additional fields. This and all the other aspects of the actual execution pipeline implementation of *LEN5* are discussed in detail in the next chapter so they are omitted here. The pieces of the extended algorithm shown here should be quite easy to understand anyway.

Issue Whenever an instruction is issued, it is allocated both in the assigned RS *and* in the ROB. Since the issue phase follows the program order, so do the instructions in the ROB. Moreover, the index of the ROB entry assigned to the instruction is saved in a dedicated field in the RS entry instead of the destination register. This actually implements register renaming using ROB entries instead of reservation stations as it was done in the original algorithm. Each ROB entry acts as a physical register that can be used alongside the RF to fetch operands during the issue stage.

To compare this new portion of the algorithm to the one reported in algorithm 2.1, the pseudo-code for arithmetic instructions is reported in algorithm 2.6.

Algorithm 2.6 Extended Tomasulo's algorithm: **issue** stage

```
procedure ISSUE(instr)
    q ← assigned RS entry
    t ← ROB tail entry
    while (!(q is empty) or !(t is empty)) do
        wait                                  ▷ wait for empty RS and ROB entries
    end while
    RS[q].valid ← yes                         ▷ the fetched instr. is inserted in q
    RS[q].res_ready ← no
    RS[q].dstROB ← t
    ROB[t].valid ← yes                        ▷ the fetched instr. is pushed in t
    ROB[t].instr ← instr
    ROB[t].res_ready ← no
    regstat[rd].busy ← yes
    regstat[rd].srcROB ← t                    ▷ the result will be buffered in t
    return q
end procedure
```

Operands fetch During issue, the operands can also be fetched from the ROB besides the RF. The information about where they can be found is stored in the register status data structure. If an operand is not available neither in the RF nor in the ROB, it can be fetched from the CDB as soon as it is produced by a functional unit. This remains almost unchanged except that this time the ROB index is used to check if the CDB is carrying the required value instead of the RS index. Notice that doing so avoids data hazards, since an instruction will fetch an operand from the *last* older instruction that produces it, regardless of the execution order.

The changes with respect to the original version reported in algorithm 2.2 are highlighted in algorithm 2.7.

Algorithm 2.7 Extended Tomasulo's algorithm: **fetch operands** stage

```
procedure operandFetch(q)
    if (regstat[rs1].busy) then
        a ← regstat[rs1].srcROB                    ▷ ROB entry producing rs1
        if (!ROB[a].res_ready) then                ▷ result not yet available in h
            RS[q].rs1_srcROB ← h                   ▷ save the index of that ROB entry
            RS[q].rs1_ready ← no
        else                                       ▷ rs1 already available in h
            RS[q].rs1_value ← ROB[a].res           ▷ fetch the value of rs1 from the ROB
            RS[q].rs1_ready ← yes
        end if
    else                                           ▷ rs1 already available in the RF
        RS[q].rs1_value ← RF[rs1]                  ▷ fetch rs1 from the RF
        RS[q].rs1_ready ← yes
    end if
    if (regstat[rs2].busy) then
        b ← regstat[rs2].srcROB                    ▷ ROB entry producing rs2
        if (!ROB[b].res_ready) then                ▷ result not yet available in h
            RS[q].rs2_srcROB ← h                   ▷ save the index of that ROB entry
            RS[q].rs2_ready ← no
        else                                       ▷ rs2 already available in h
            RS[q].rs2_value ← ROB[b].res           ▷ fetch the value of rs2 from the ROB
            RS[q].rs2_ready ← yes
        end if
    else                                           ▷ rs2 already available in the RF
        RS[q].rs2_value ← RF[rs2]                  ▷ fetch rs2 from the RF
        RS[q].rs2_ready ← yes
    end if
end procedure
```

Execution This phase remains essentially unaffected by the introduction of the ROB, and so does the interface between the RSs and the functional units. The result from a functional unit is still buffered in the RS while it waits for the CDB to become available. Therefore, the pseudo-code of this step is quite similar to the one in algorithm 2.3, as shown in algorithm 2.8.

Algorithm 2.8 Extended Tomasulo's algorithm: **execute** stage

```
procedure execute(q)
    if (RS[q].rs1_ready and RS[q].rs2_ready) then   ▷ both operands available
        result ← RS[q].rs1_value <op.> RS[q].rs2_value   ▷ compute result
        RS[q].res_ready ← yes
        RS[q].res ← result
    end if
end procedure
```

Write result When the CDB is available, the result is broadcast to all the RS but not to the RF or the memory. Instead, the result is copied to the ROB entry where the producing instruction was allocated during issue. Also, exception that might arise during the instruction execution are buffered in the ROB instead of being processed immediately. This has a very important consequence: the execution completion does not correspond to the instruction commit. The produces result can still be used by subsequent instructions, even speculative ones, that may read it from the CDB, but the state of the processor is not modified yet. For this reason, a new step shall be introduced into the execution algorithm to take into account the difference between writing the result into the ROB and writing it into the RF or memory.

The pseudo-code of this new phase in the execution is shown in algorithm 2.9.

Algorithm 2.9 Extended Tomasulo's algorithm: **write result** stage

```
procedure writeResult(q)
    a ← RS[q].dstROB
    result ← RS[q].res
    RS[q].valid ← no                          ▷ remove the instruction from the RS
    ROB[a].res_ready ← yes                    ▷ copy the result to the ROB
    ROB[a].res ← result
    for (∀y) do                               ▷ for each RS y
        if (RS[y].rs1_srcROB = h) then
            RS[y].rs1_ready ← yes
            RS[y].rs1_value ← result          ▷ copy the result to y
        end if
        if (RS[y].rs2_srcRS = H) then
            RS[y].rs2_ready ← yes
            RS[y].rs2_value ← result          ▷ copy the result to y
        end if
    end for
end procedure
```

Commit In each cycle, only the instruction that reaches the head of the ROB can commit, meaning that its result can be copied into the RF or into the memory. This actually implements the in-order commit requirement. Also, exceptions are only processed at this time, implementing the precise exception model. If an exception was raised or the instruction has been detected to be mispredicted, the entire execution pipeline including the ROB can be flushed. By construction, only instructions that followed the offending one are affected, because all the previous ones have already completed and committed. In other words, the state of the processor is modified in program order, and only by instructions that are confirmed not to be speculative. Notice that because of this modifying the CSRs leads to a hazard. In fact, instructions that followed the CSR instruction can still be executed

on out-of-date CSRs content if they are executed before the CSR instructions commits. On the other hand, the CSRs cannot be updated during execution, because some older instructions might not have executed yet. *LEN5* avoids this by stalling the execution pipeline when an instruction that modifies the content of the CSRs is detected during issue, so that all subsequent instructions are prevented to execute until the state of the processor has successfully been updated[8].

Once again, the extended algorithm version of the commit phase reported in algorithm 2.4 is shown in algorithm 2.10.

Algorithm 2.10 Extended Tomasulo's algorithm: **commit** stage

```
procedure COMMIT(h)
    dest ← ROB[h].instr.rd                    ▷ read rd from the committing instr.
    RF[dest] ← ROB[h].res                     ▷ copy the result to the RF
    ROB[h].valid ← no                         ▷ pop the committing instr.
    if regstat[dest].srcROB = h then
        regstat[dest].busy ← no               ▷ mark rd register as available
    end if
end procedure
```

Finally, the main loop of the extended version of the algorithm is reported in algorithm 2.11.

Algorithm 2.11 Extended Tomasulo's algorithm: main loop

```
while (1) do                                  ▷ in every cycle
    if (∃ new_instr) then                     ▷ fetched instr. available
        r ← ISSUE(new_instr)
        OPERANDFETCH(r)
    end if
    for (∀i) do                               ▷ for all in-flight instructions in RS
        if (RS[i].valid and !RS[i].res_ready) then
            EXECUTE(i)                        ▷ compute result
        else if (RS[i].res_ready and CDB available) then
            WRITERESULT(i)                    ▷ write result to the ROB
        end if
    end for
    h ← ROB head entry
    if ROB[h].res_ready then
        COMMIT(h)
    end if
end while
```

[8] more on this in section 3.3

Chapter 3

LEN5 Execution Pipeline Design

In this chapter, the design of the execution pipeline of the *LEN5* processor is discussed. At first, the top-level architecture is introduced and some design choices that are common to all the building blocks are briefly explained. Then, an in-depth analysis of each basic block in the execution pipeline is provided. When referring to one of these blocks, its name will be printed in *italic* to distinguish it from the general component not taken in the context of the design of *LEN5*. So, the *LEN5* implementation of the ROB will be called *ROB*.

As mentioned in the previous chapters, *LEN5* is a single-issue processor, meaning that one instruction per cycle can be issued at most. Notice that the extended version of Tomasulo's algorithm supports multiple issue without requiring heavy changes to the execution pipeline. The dependences between all the concurrently issuing instructions must be detected and addressed in parallel. Moreover, more than one instruction of the same type might be issued in the same cycle. This means that more than one RS must be allocated in each cycle. The same is true for the ROB, whose entries must be allocated at the same time while respecting the program order. Moreover, to achieve the maximum theoretical IPC, which corresponds to the number of instructions issued in each cycle, multiple results must be written to the ROB at the same time. This probably requires more than one CDB. The possibility of implementing multiple issue was considered at the beginning of this project. However, it was then wisely decided not to overly complicate what is already a quite sophisticated architecture. So this possibility is left for possible future developments of *LEN5*, as suggested in section 5.3.

3.1 Top level

As pointed out in sections 2.3 and 2.4, one of the main features of Tomasulo's approach to dynamic instruction scheduling is the implementation of register renaming and operands fetch by means of Reservation Stations. These, together with the Reorder Buffer, the Common Data Bus and the Register Status data structure form the backbone of the execution pipeline of *LEN5*. While these components should be already known from chapter 2, some others are actually required for the execution pipeline to work properly. A brief description of each one of these is given in the following paragraphs. Notice that every sequential component in the execution pipeline has an asynchronous reset built-in, even if this is not pointed out every single time. Besides, the size of each data structure is parametric, and some possible values are discussed in chapter 4.

Issue queue A FIFO queue that holds fetched instructions before they are issued.

Issue logic A combinational control block that implements all the steps of the *issue* and *operands fetch* stages described in algorithms 2.6 and 2.7 for all the instructions (not only the arithmetic ones). One of its key components is the main instruction decoder (*issue decoder*), that directs instructions to the correct *RS* and generates some of the control signals for the main control logic of the execution pipeline.

Integer and floating-point register files Both contain 32, 64-bit wide registers that implement the architectural registers if the RISC-V RV64I base integer ISA and the F/D single-precision and double-precision floating-point extensions.

Load-store unit The functional unit dedicated to *load* and *store* instructions. It contains two separate queues: the *load buffer* and the *store buffer*. The interface with the *data cache* and the *Data Translation Lookaside Buffer (DTLB)* are implemented by this unit, as well as the store-to-load *forwarding* mechanism.

Branch unit The functional unit dedicated to branch instructions. It is fed by a special RS compiled in FIFO order, and communicates directly to the front-end to update the branch predictor data structures.

Control Status Register The CSRs holding the performance counters and all the data that define the processor status and privilege mode of execution.

Commit logic A second control block that manages the instruction *commit* described by algorithm 2.10 for all the instructions. It also generates the control signals that are required to handle exceptions and flush the execution pipeline.

Main control logic The top-level control logic, that handles and directs those operations that affect all the execution pipeline, like *stall* and *flush*.

The communication between all these components is performed using an AXI-like *valid-ready* handshake protocol that is defined in section 3.1.1. The top-level architecture of the *LEN5* execution pipeline is reported in Figure 3.1.

Some features are common to most of the blocks in this chapter. Two of these are discussed here once and for all: the handshake protocol and the control model of the data structures like the RSs or the ROB. After, the blocks that are common to many units in the pipeline are detailed.

3.1.1 Handshake

To achieve both modularity and support for variable latency functional units, a handshake protocol is required. In *LEN5*, the same handshake protocol is used for the communication among the internal components of the execution pipeline and for the interfaces with the functional units, the front-end, and the memory system. Different units are granted access to shared ones by arbiters that act only on the handshake signals. Moreover, the employed handshake protocol must support the maximum throughput of one transaction per cycle.

To meet all those requirements, the *valid-ready* handshake protocol from the Advanced eXtensible Interface (AXI) communication interface was chosen. AXI compliant busses are one of the most successful when it comes to on-chip communication. However, *LEN5* internal links are **not** AXI-compliant. Only the handshake process from the AXI specifications [2] is actually implemented in each communication channel. The format of the transmitted payload is completely custom and specific to each bus. Also, some additional control and status signals may be transferred in the payload together with the data, without using a second dedicated channel with its own handshake signals. Still, the transaction actually takes place only if the handshake process has been successful.

The AXI-like handshake protocol is based on two main signals for each channel:

- **valid:** signals the receiver that the payload available on the bus is valid from the same cycle in which the *valid* is raised.
- **ready:** signals the sender that the payload can be accepted.

The payload is considered transferred only when *both* the *valid* and the *ready* signals are asserted. These two signals are subject to the following constraints:

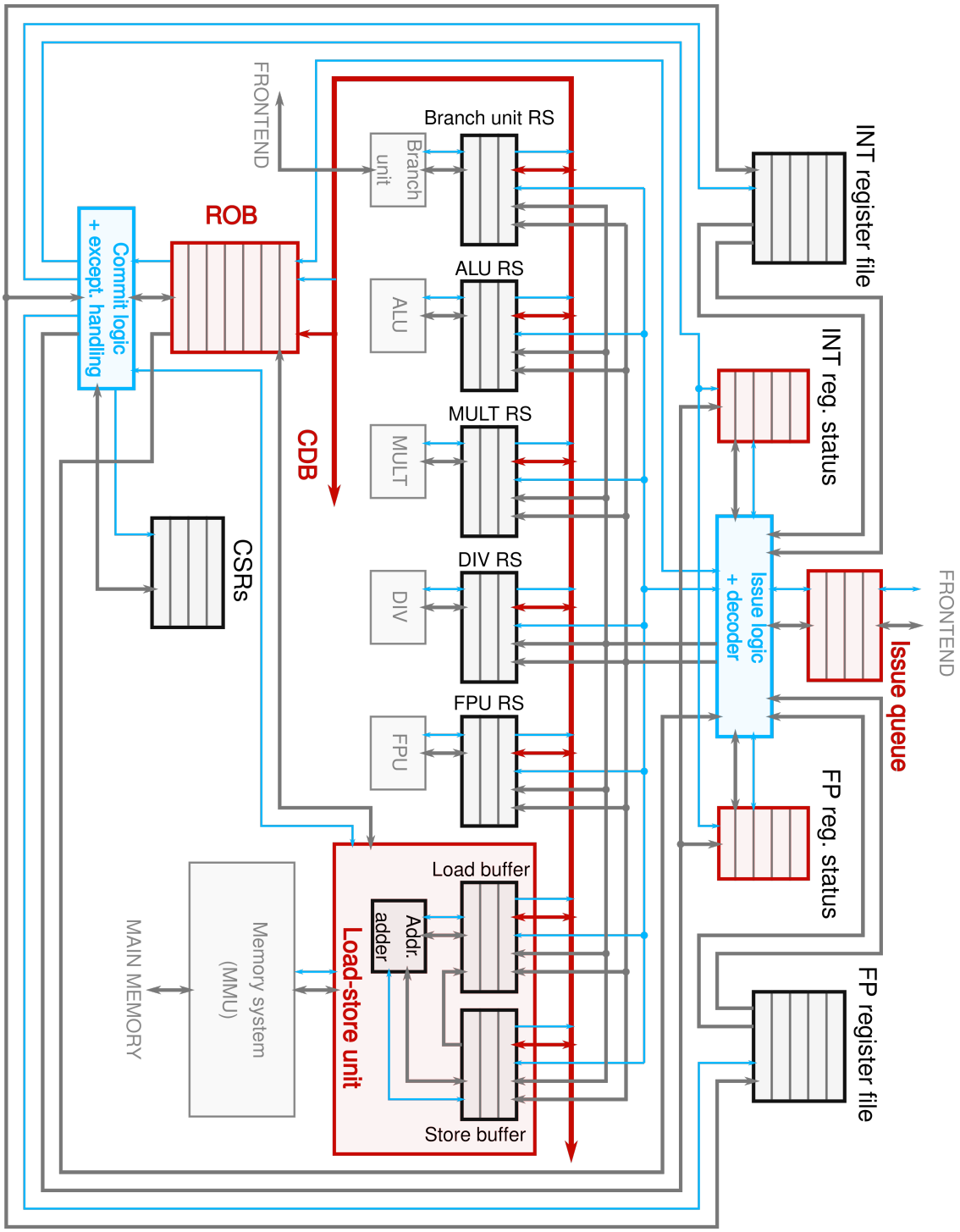


Figure 3.1: Top-level architecture schematic of the execution pipeline of *LEN5*. Handshake and control signals are colored in blue.

1. The *valid* signal must remain asserted until the handshake completes and the request is caught by the receiver, that is when the receiver asserts its *ready* signal.

2. In a synchronous system, the sender must maintain the payload stable until the first active clock edge since the moment the receiver raised it *ready* signal. The receiver will sample the data at the end of the clock cycle in which it raised the *ready* signal. From the beginning of the next clock cycle, the sender can change the payload on the bus.

3. The sender cannot wait for the receiver to raise its *ready* before asserting the *valid*. This means that a request can be done in every cycle, regardless of whether the receiver can accept it or not.

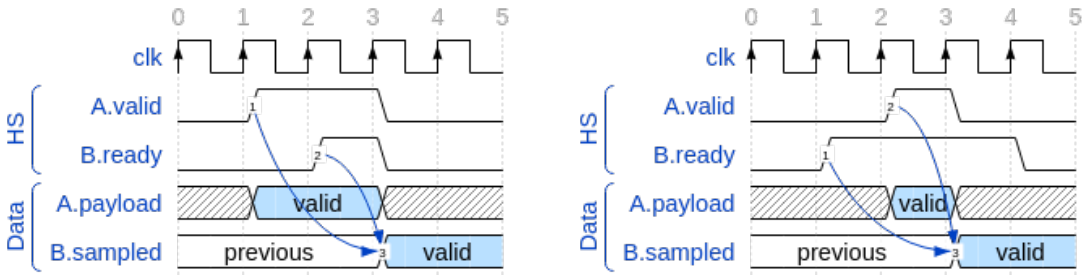
4. The receiver may or may not wait for the sender *valid* to be raised before asserting its *ready*. This means that a receiver that is currently able to accept a request could also maintain its *ready* low until the sender actually performs a request.

5. If a unit has its ready asserted, it can deassert it before a *valid* is received by the sender. This is particularly useful when dealing with shared units or buses: if no requests are received in a certain cycle (i.e. all the *valid* from the source units are low), the arbiter can assert the ready for all the source units. If one or more valid is asserted in the next cycle, by the end of that cycle the arbiter can deassert every *ready* except for the one for the unit whose request will be processed (e.g. the one with the maximum priority).

From the above description, it is clear that each couple of *valid-ready* signals is unidirectional from unit A, which controls the *valid*, to unit B, which controls the *ready*. If a bidirectional channel between two units is needed, a second couple of *valid-ready* signals is required, this time going in the optional direction: a *valid* controlled by B and a *ready* controlled by A. In this case, also a second data bus is required: no tri-state bus or logic is used in *LEN5* in general.

To better understand the use of *valid* and *ready* signals, two examples are shown in the timing diagrams in fig. 3.2.

Notice that the *valid* signal can remain asserted after the first transaction completed. This permits burst transfers of multiple payloads in consecutive cycles besides single transactions. An example of this use is shown in fig. 3.3. As a consequence, the channel maximum data-rate can always be reached. This only requires the receiver to be ready in each clock cycle. Almost every component in the *LEN5* execution pipeline can achieve this provided that it has enough storage to save the incoming data. In a single issue pipeline, where the maximum theoretical throughput is one completed instruction per cycle, having such a possibility is mandatory to entirely exploit ILP.



(a) *Valid* asserted before *ready*. (b) *Ready* asserted before *valid*.

Figure 3.2: Simple handshake examples: in fig. 3.2a the *ready* is asserted after the *valid*, while in fig. 3.2b the *ready* is already asserted when the sender raised its *valid*.

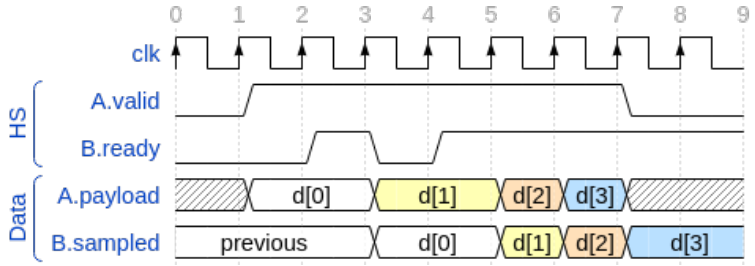


Figure 3.3: AXI-like handshake protocol in burst transactions: single transaction in cycle 3, burst transaction from cycle 5.

Notice that after the first single transaction is completed (cycle 3), the *valid* from A is not deasserted, and the payload is simply changed and maintained stable until B is ready again. In cycle 4 B becomes ready again and keeps being ready in the next cycles too while A keeps sending new valid data in each cycle. Since B is ready, all the data are accepted and the maximum data-rate of one transaction per clock cycle is maintained during the two-data burst ($d[2]$ and $d[3]$).

One last remark on the nomenclature: handshake *valid* and *ready* signals will always appear in *italic* to distinguish them from the `valid` status signals that are generated by many data structures and components.

3.1.2 Control

When dealing with control-dominated applications, the use of Moore Finite State Machines (FSMs) is definitely highly recommended. They ensure that the output control signals depend only on the current state of the machine and not on the input signals. Since the state is updated synchronously, this avoids any sort of combinational loops in the control domain. The only downside is that the state update latency must be paid whenever a state change is required. This basically

means introducing a one-cycle delay in the request processing, which is hardly ever significant in most designs.

However, when dealing with many different data flows and structures, Moore FSMs may not represent the best solution. Let us take the *load buffer* of the *LEN5* execution pipeline as an example. This very complex component must process many different operations in each cycle, each of which may apply to only one of its entries or to all those meeting a specific requirement, in what could be defined as a CAM-like access. All these operations are detailed in section 3.7.2. In each cycle, the amount and combination of the operations are totally arbitrary and unpredictable, since they depend on other components whose availability and latency are unknown. Despite this, the *load buffer* must always be ready to process each answer from those components in any circumstance[1]. If this mechanism was combined with a Moore FSM to process one operation at a time, the load buffer would by far represent the most significant bottleneck of the core, possibly stalling the entire pipeline. Moreover, this would mean handling requests and answers sequentially, while the intent of the chosen execution algorithm is to do that concurrently and possibly out of order.

In such data structures, the operation to be performed is decided based on some status fields that are present in all the entries. Since these are always updated synchronously, also the operations are. Usually, there is only a very simple combinational network that produces the control signals based on those status signal. This is exactly what happens in a traditional FSM where control signals are generated by a combinational network according to the state of the machine, that is encoded in some synchronous logic elements. Under a certain perspective, the *load buffer* and all the similar structures in the execution pipeline can be seen as FSMs where the status is encoded in the entries of the data structure. The real difference is that the state is not explicitly defined. Still, there is some synchronous object (i.e. the data structure itself) that separates the input signals from the output signals and the operation control, as it happens in a Moore FSM. The number of possible states if a function of all the combinations of values of the status bits in those structures. So in a N entry *load buffer* with M status bits per entry, the number of possible states is $2^{N \cdot M}$. In a realistic scenario with $N = 8$ and $M = 5$, the number of possible states is in the order of 1×10^{12}. Of course, not all these combinations are allowed and even possible. Each entry is updated according to the requests from other units, which may be considered as the input control signals of a traditional FSM. When this happens, only specific status bits of specific entries are updated at the next active cycle. This is the key difference that allows saving one cycle when updating the data in this components: the input signals are not used to update the state of a machine (by means of a next state generation

[1]Because the memory system does not wait for the *ready* signal from the *load buffer* when answering a request, even if this is provided (always high as a consequence of this).

combinational network) that is going to modify the content of the structure only in the next cycle, effectively requiring two cycles for the update to take place. Instead, the input signals are processed by a combinational update logic that acts directly on the content (i.e. the status) of the structure. The updated status and values will be available from the next cycle.

Of course, describing such control systems requires particular attention. Many redundant checks and *default* statements have been used while writing the code. See appendix A.2 for more details about the System Verilog description of this kind of component.

A few more words must be spent on the implementation of the AXI-like handshake protocol described in section 3.1.1. Each entry of the *load buffer* and similar structures contains a `valid` bit that indicates whether the entry contains valid data or not, similarly to what was shown in table 2.2 for the RSs. When an entry is selected to be sent to a functional unit or another block, its `valid` status bit is checked. The output *valid* signal of the block is raised only if the `valid` status bit of the selected entry is raised. Therefore, the output *valid* signal of a unit is always updated synchronously, as the control signals required to update the data structure that were discussed before. The *valid* is generated regardless of the value of the *ready* signal from the destination unit, according to the constraints reported in section 3.1.1. In a similar way, the output *ready* signal of a unit is generated based on the status signals from the data structure. Most of the time the *ready* signal is asserted whenever the structure is not full (i.e. incoming data can be accepted), or if the entry where to store the incoming data has already been allocated in the structure[2].

An example timing diagram of how the *valid-ready* handshake protocol is implemented in a FIFO buffer like the *issue queue* is reported in fig. 3.4. The output `ready_o` signal is asserted whenever the data FIFO contains at least one empty (i.e. not valid) entry. For what concerns accepting new data, the example follows the process shown in fig. 3.2b, with the only difference being that new data do not overwrite old valid data. Instead they are allocated in the entry pointed by the `tail` index of the FIFO (see cycle number 3). The `tail` index is incremented each time a new instruction in pushed in the FIFO. On the other hand, the output `valid_o` signal is asserted only if the entry pointed by the `head` index contains valid data (see cycle number 2). If the destination unit can accept the data (i.e. `ready_i` is asserted), the `head` index is incremented and the current head entry is marked as empty (i.e. not valid) as shown in cycle number 8.

An interesting case is the one showed in cycle number 8 for the `d[0]` FIFO entry. By the end of cycle 7, the destination unit was able to accept the valid data from

[2]As an example, when the *data cache* produces the value requested by a *store*, this is simply saved in the entry of the *load buffer* that performed the memory access request and that was allocated in the *load buffer* during issue.

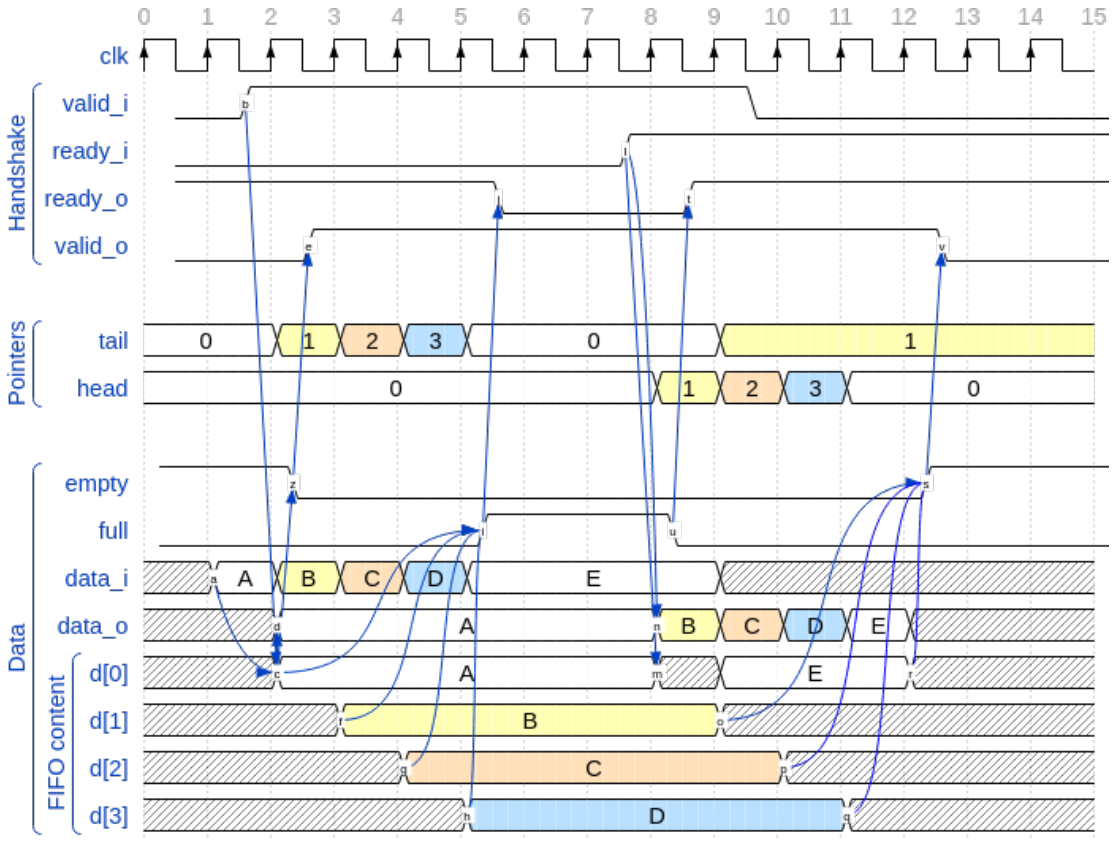


Figure 3.4: An example of *valid-ready* handshake in a FIFO data structure.

this entry, so at the beginning of cycle 8 the entry is marked as not valid. In the meantime, the source unit is sending a valid data (E) on the bus. This new data could be stored in entry `d[0]` already in cycle 8. However, this doesn't happen because by the end of cycle 7 the FIFO `ready_o` signal wasn't asserted because all the entries were full. In this case, a one-cycle latency must be paid before e new data can be pushed in the FIFO. The only solution is to modify the logic that generates the `ready_o` signal to take into account this special case: if the buffer is full (i.e. the head and tail pointers have the same value and all the entries are valid) and the destination unit can accept the data from the current head entry, then a new data can be accepted by the source unit at the end of the current cycle, and so `ready_o` can be asserted. However, this solution breaks the design rule stating that a synchronous element should be present between the inputs and the outputs of certain block: the `ready_o` signal becomes also a function of the input `ready_i`, and not only of the status bits from the data structure. For this reason, this solution has only been adopted in the most simple and easy to verify structure of the entire execution pipeline: the *issue queue*. In all the other units, a one-cycle delay is paid whenever some data must be written in an entry from

a full data structure. Given the large amount of functional units that introduce significant delays in the execution pipeline, and that the number of entry in each data structure can be tuned to have enough space most of the time, this one-cycle latency doesn't represent a real limitation for the entire system.

3.1.3 Arbiters

Some of the components in the *LEN5* execution pipeline are shared among many units. This is the case of the *CDB*, the *virtual address adder*, and also the memory system. Of course, only one source unit can be served in each cycle. Thanks to the *valid-ready* handshake protocol, dealing with these conflicts is very easy: it is enough to insert an arbiter between the source units and the destination unit. The arbiter selects the source unit that is going to be served and propagates it's *valid* signal to the shared unit. At the same time, it will propagate the *ready* signal from the shared unit only to the served source unit. Besides, the arbiter generates a control signal to drive a multiplexer (MUX) that selects the correct source of data for the channel. As an example, the schematic of a 2-way arbiter is shown if fig. 3.5.

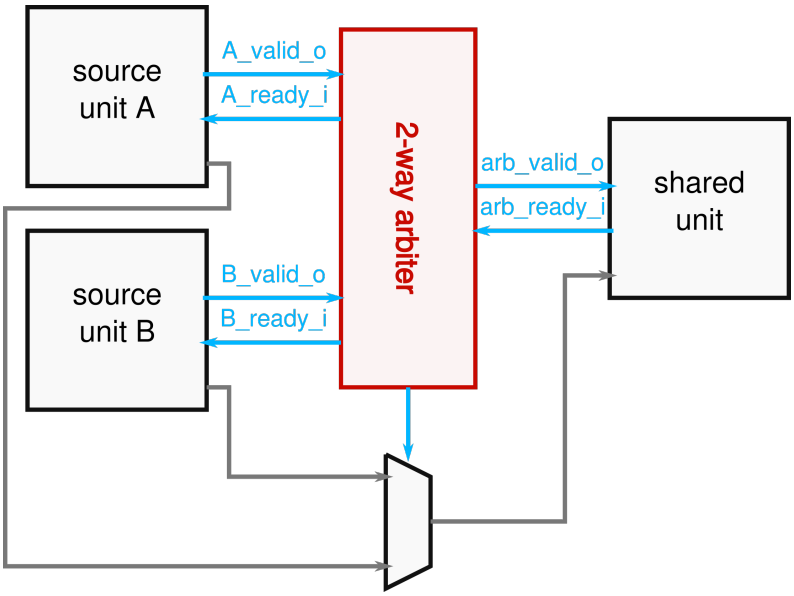


Figure 3.5: Two-way handshake arbiter. It generates the handshake signals for the source and destination (shared) units and the control signals to selct the data from the chosen source unit.

Two different 2-way arbiters are available in *LEN5*. The first is a fixed priority arbiter that serves always the same unit in case of conflict. It is basically a 2-way priority encoder. The second contains a flip-flop that stores which was the last served unit in case of conflict, that is when both the source units assert their *valid*

signals for the shared unit in the same cycle. When this situation occurs again, the arbiter will serve the unit that wasn't served the first time. In the extreme case where both the *valid* signals are asserted in each cycle, the source units are served in an alternating fashion. In the same situation, the fixed priority arbiter processes only the request from the first unit, stalling the other one.

Notice that using the fixed priority encoder in the execution pipeline can increase the latency of some instructions. Still, this doesn't cause these instructions to starve. Let us take the *load-store unit* as an example. A priority arbiter is used to access the shared *virtual address adder*, with priority given to *store* instructions over *load* instructions. Suppose that only *store* instructions are issued after a *load* and that the *load* and the first *store* get their operands ready in the same cycle. This means that the *load* took an additional cycle to fetch its operands than the first *store*. Suppose also that there are no dependences between all these instructions, and so that all subsequent *stores* have their operands ready from the moment they are issued. Remember that all these instructions are allocated in program order in the *ROB* during issue, besides being inserted in the *load buffer* or *store buffer*. Because *stores* have higher priority than *loads*, the first *store* (the second instruction of the sequence) is executed first, having its virtual address computed by the *virtual address adder*. Then, the second issued *store* executed and so on. If all the *stores* hit during the memory accesses, the *store buffer* is never full, and so new *store* instructions can be pushed in, preventing the *load* from executing. Notice that since the *load* is never executed, it cannot commit even when it has reached the head of the *ROB*. After a number of cycles equal to the number of entries in the *ROB*, no more instructions can be issued, because the *ROB* is full. Still, already issued *store* instructions can be executed. After the last *store* computes its virtual address, the conflict accessing the *virtual address adder* terminates. At this point, the *load* request is eventually accepted by the *virtual address adder*, and the instructions can perform the memory access and finally commit from the head entry of the *ROB*.

In the default configuration, this is actually how the *LEN5 load-store unit* works: *store* instructions are always given precedence over *load* instruction when accessing the *virtual address adder*, the *DTLB* and the *data cache*[3]. The type of arbiter instantiated can be configured using the dedicated switch as explained in appendix A.1.1.

Besides the case where two units need to access the same shared resource, there's also the opposite situation: a shared unit that can communicate to more than one destination unit. In this case, it must provide a signal to discern which unit the payload is for. A simple decoder will assert only the *valid* signal for the actual destination unit and will propagate its *ready* signal to the shared source unit. The

[3]See section 3.7 for more details about the execution process of *loads* and *stores*.

data are broadcast to all the destination units, but only the one whose input valid is asserted will sample them.

3.1.4 Entry selectors

These *selectors* are the components responsible for choosing the next instruction to be executed from the ones allocated in a certain *RS*. The choice is made according to the status signals of each entry. Each selector provides two output signals: the index of the selected entry and a valid signal that is used to encode the situation where no entry is selected, in which case the index just points to the default entry (i.e. entry *zero*). Since only the entries meeting certain requirements are allowed to be executed, the input of the selectors is usually a combination of status vectors from the data structure. As an example, an instruction in the *load buffer* can perform the cache access only if it's valid, its virtual address has already been computed from its operands and it has already been translated into the corresponding physical address according to the OS provided page table.

The produced index and valid signals are used by the control logic of each component to generate the necessary handshake and operation-control signals required to execute an instruction or to update the data structure, as described in section 3.1.2.

In *LEN5*, two different implementations of the selectors are provided, while a switch in the execution pipeline configuration file is used to instantiate one or the other, as explained once again in appendix A.1.1.

The first selector is a simple *priority encoder*: if many entries of the *RS* meet all the requirement for execution, the selector chooses the one with the higher fixed priority, that is the one pointed by the lowest index in the *RS*. Notice that since the execution and replacing of instructions is performed OoO, there is absolutely no relation between the program order of instructions and the index of the assigned entry of the *RS*. So a suitable instruction could be selected for execution even if it appears after another suitable one in program order. Due to the OoOE scheme, this can be iterated for many cycles. However, while the latency and overall performance of the execution pipeline may slightly get worse because of this, *starvation* is avoided by construction: sooner or later each instruction in a *RS* is guaranteed to execute, because eventually no more instruction will be issued if no instruction commits from the *ROB* for a number of cycles equal to the depth of the *ROB*. A similar situation can be easily reduced to the *starvation* example of section 3.1.3.

To possibly remove the latency drawbacks of a fixed priority encoder, the second type of selector takes into account the *age* of each entry in the *RS*. First, a definition of age must rely on some metric that depends on the program order of instructions. For this reason, the most exact definition of the *age* of an instruction would be the number of cycles that have elapsed from issue. However, the number of cycles an instruction is kept in a *RS* before it is executed is not known and nor it's its maximum value, that depends on the hazards and on the latency of the *functional*

unit producing the source operands for the considered instruction. The definition of age can be relaxed and made independent on the absolute number of elapsed clock cycles. Instead, it could take into account only the number of instructions that have been issued *after* the instruction whose age must be computed. Remember that even if the maximum IPC of the pipeline is one instruction per cycle, due to cache misses and structural or data dependences, the effective IPC is very likely to be lower, so the number of instruction issued in a certain amount of time is always lower or equal to the number of elapsed clock cycles. The great advantage of this second definition is that the maximum number of instruction that can be issued before a certain instruction commits is known, and it's equal to the depth of the *ROB*: once the *ROB* is full, no instruction can be issued until the oldest instruction in the execution pipeline (i.e. the one that occupies the head entry of the *ROB*) commits, regardless of when this happens. A further optimization is taking into account only the instructions issued to the same *RS* of the considered instructions. In this case, the age of instruction is defined only in relation to other instructions of the same type and can be computed inside each *RS* regardless of what is being done in the rest of the execution pipeline. Notice that even if the number of entries in a *RS* is lower the number of entries in the *ROB*, the maximum *age* remains equal to the depth of the *ROB*, not of the *RS*, as a consequence of OoO execution and completion. If an instruction whose result has been computed is popped from its *RS*, another instruction can take its place as long as it can also be allocated in the *ROB*, even if older instructions haven't been executed or completed yet.

The second type of selector is referenced as *age-based selector*. In addition to the status bits of the data structure entries, it also takes the entry ages in input and produces the index of the oldest one meeting all the execution requirements. The age of each instruction is tracked by a dedicated field in each entry of a *RS*. This field acts like a counter that is cleared when an instruction is inserted in the corresponding entry (and of course during flush and reset) and incremented each time a new instruction is inserted in the *RS*. The *age-based selector* was coded in a behavioral way to let the synthesis tool perform all the necessary optimizations. Still, a structural tree implementation was manually designed to estimate the impact of this complex combinational component in the cycle time of the entire system when compared to a priority encoder like the first type of selector. Since many comparators and MUXs are required to propagate the highest age value through the layers of the tree, the *age-based selector* is expected to be much slower than a priority encoder with the same number of inputs. The synthesis results reported in section 4.2.2 show that the age-based selector is, in fact, more than one order of magnitude (20 times actually) slower than a simple priority encoder of the same size, while its area is about 7 times higher. So the use of this component is advised only if significant improvements are brought system-wide, and possibly if this component doesn't enter the critical path. For this reason, by default, it is not instantiated in the execution pipeline of *LEN5*, as explained in appendix A.1.1.

The component that would probably take the most advantage from an *age-based selector* is the *load-store unit*, where it might increase the hit ratio during the store-to-load forwarding process, possibly saving some *data cache* and *DTLB* access for *load* instructions.

3.2 Issue queue

The first component that a fetched instruction encounters during its execution is the *issue queue*, as shown in fig. 3.1. It works as a buffer between the frontend and the execution pipeline itself. In particular, it takes fetched instruction from the *fetch unit* and sends them to the *issue logic* for decoding and allocation in the associated *RS*. The *issue queue* corresponds to the pipeline register between the *instruction fetch* and the *instruction decode* stages in a traditional pipelined processor. Both the communication with the *fetch unit* and with the *issue logic* are implemented using the AXI-like *valid-ready* handshake protocol described in section 3.1.1.

Since this first version of *LEN5* is a *single-issue* processor, the *issue queue* is a simple circular FIFO buffer where one entry contains a single instruction. As a consequence, the maximum IPC is one instruction per cycle, with the actual one being lower due to cache latency and structural or data hazards[4].

The main purpose of the *issue queue* is to partially mask slowdowns that arise from hazards. In particular, the *issue queue* reduces the need for a pipeline stall in the following two situations:

- Structural hazards or hazards among in-flight instructions might fill one or more RSs or the *ROB*. In either case, instructions can no longer be issued until an entry is available. Thanks to the *issue queue*, the frontend is not prevented from fetching new instructions from the *instruction cache* as long as this has available entries. In the meantime, hazards inside the execution pipeline might resolve, and new fetched instructions can be issued again.

- Without the *issue queue*, a miss in the *instruction cache* would imply stalling the issue phase until a new instruction is fetched from the *L2 cache* or the memory. The *issue queue* allows the execution pipeline to be fed with new instructions as long as previously fetched instructions are available (i.e. until it is not empty).

The *issue queue* depth can be sized according to the application and/or application needs. In a well-dimensioned system, the *issue queue* should on average be neither full or empty. Notice that the *issue queue* state can be used to evaluate bottlenecks in the system. If the *issue queue* is full most of the times it means that the execution pipeline is not fast enough to exploit the entire memory system instruction throughput. As opposite, if the *issue queue* is empty most of the time, it means that the memory system cannot provide enough instructions to fed the

[4]If a more sophisticated implementation requires *multiple-issue*, the *issue queue* could be simply expanded to accept twice as much data from the *fetch unit*, as proposed in section 5.3.2

execution pipeline to its maximum IPC. The information about the status of the *issue queue* is given by the *LEN5* code as described in appendix A.3.

The high-level block diagram of the *issue queue* is reported in fig. 3.6.

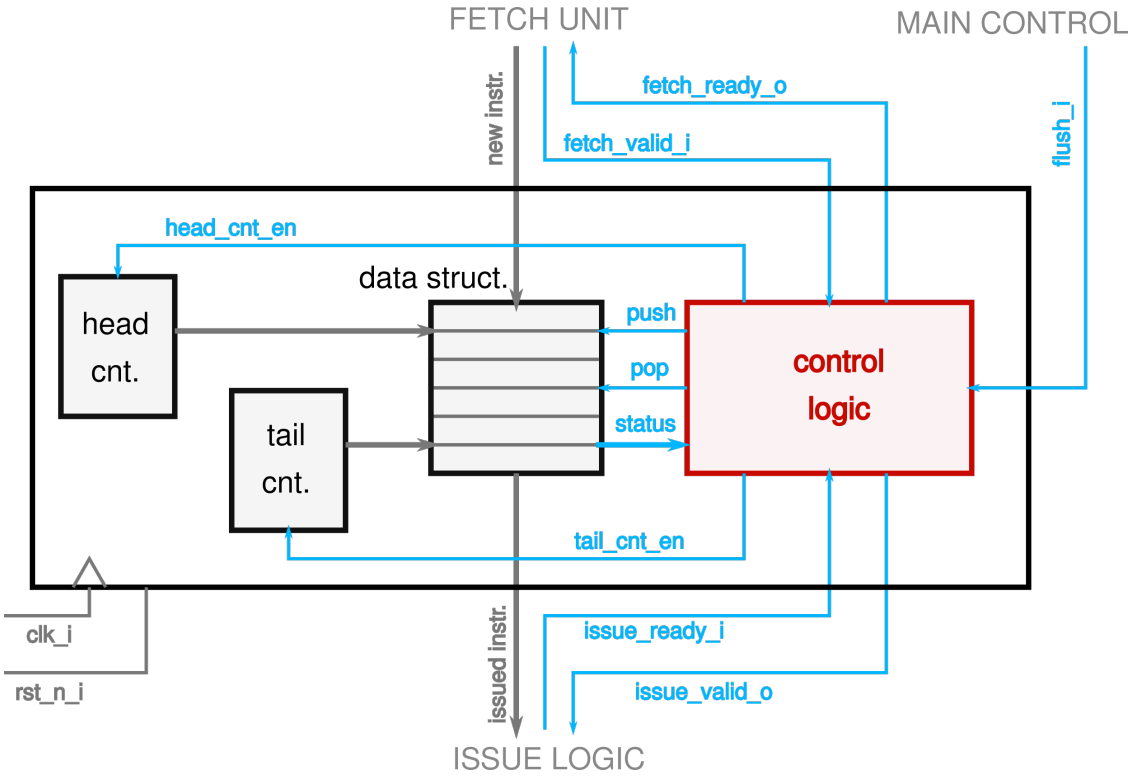


Figure 3.6: *issue queue* block diagram.

Besides the actual FIFO where instructions are buffered, the *issue queue* contains two *modulus counters* whose output is used to address the *head* and *tail* entry of the buffer. The pointer counters maximum value is the *issue queue* depth, after which they start over. Since modulus counters are used, the *issue queue* depth may or may not be a power of 2. The following section explains how these counter and the data structure are updated by the *issue queue* control logic.

3.2.1 Issue queue data structure

The *issue queue* takes from the *fetch unit* much information about the fetched instruction as shown in fig. 3.6, most of which is used to address and update the branch predictor tables after a *branch* instruction has been processed by the *branch unit* as explained in section 3.6.2. This data is stored in dedicated fields in the tail entry of the *issue queue*. Here follow a list of these fields with a short description of each one fo them.

- **valid:** a bit indicating whether the current entry contains a valid instruction. It is set when an instruction from the *fetch unit* is pushed into the *issue queue* and cleared when the same instruction is popped from the *issue queue* and sent to the *issue logic* or when the *issue queue* is flushed, as described in section 3.2.2. The `valid` bit of the head entry of the *issue queue* is used to generate the *valid* signal to the *issue logic*. It is important to notice that the *empty* and *full* conditions can't be distinguished by checking whether the head and tail pointers have the same value. Instead, they are evinced from the `valid` status bit of all the *issue queue* entries.

- **current PC:** contains the 64-bit PC of the current instruction. This value is required for exception handling as described in section 3.2.3 and, if the instruction is a *branch*, by the branch predictor to update its table if a misprediction is detected by the *branch unit* during the *branch* execution.

- **instruction:** this field contains the 32-bit instruction itself. During issue, this data is processed by the *issue logic* to fetch the operands and to send them to the correct *RS*.

- **predicted target:** the 64-bit predicted address of the next instruction of a *branch* from the Branch Target Address (BTB) of the frontend branch predictor[5]. This information is used by the *branch unit* to detect a misprediction by comparing this value to the one computed during the *branch* execution.

- **predicted taken:** a bit that is asserted when the current *branch* instruction was predicted to be a *taken* branch by the frontend branch predictor. This information is used together with the previous one to detect a misprediction.

- **exception raised:** a bit indicating if an exception was raised during the current instruction fetch. The exception handling in the *issue queue* is briefly described in section 3.2.3.

- **exception code:** contains a 4-bit code associated to the exception that was possibly raised during the instruction fetch.

An example of the content of the *issue queue* in a given cycle is reported in table 3.1.

3.2.2 Issue queue control logic

The control logic of the *issue queue* follows the general principles described in section 3.1.2. The *issue queue* is one of the most simple data structures of the *LEN5*

[5]See [3] for details.

	valid	**curr. PC**	**instr.**	**pred. trgt.**	**pred. tkn.**	**ex. rsd.**	**ex. code**
	no	-	-	-	-	-	-
$h \rightarrow$	yes	`0x0...deaf0`	**bne**	`0x0...beef0`	yes	no	-
	yes	`0x0...beef0`	**add**	-	-	yes	`0x1`
$t \rightarrow$	no	-	-	-	-	-	-
	...	...	...	...	...	...	...
	no	-	-	-	-	-	-

Table 3.1: *Issue queue* content example. h and t are the *head* and *tail* pointers. A '**-**' means that the field content is a *don't care*. In the example the head entry contains a *branch* instruction with its auxiliary data. Since the branch was predicted taken, the frontend fetched the next instruction from the predicted target address of the first. However, this instruction caused a *instruction access fault* exception (code `0x1`) during the fetch phase.

execution pipeline. There are only three main operations that can be performed, as reported in the following paragraphs.

Push a new instruction The fetch unit signals the *issue queue* that a new instruction has been fetched asserting its *valid*. If the *issue queue* has at least one empty entry its output *ready* signal is asserted. If the handshake completes successfully (i.e. both *valid* and *ready* active as described in section 3.1.1), the *fetch unit* can proceed and fetch a new instruction in the next cycle, while the *issue queue* stores the current input instructions and associated data in its *tail* entry and increments its *tail* counter. The newly allocated entry is marked as valid. This process is similar to the one shown in fig. 3.4. If the *issue queue* is full, its output *ready* signal is not asserted and the handshake fails. In this case, the tail pointer is not incremented, and the *fetch unit* is stalled, meaning that it will keep the same fetched instruction at his output also in the next cycle. Notice that if the *issue queue* is full in a certain cycle, but its head entry is being popped, a new instruction from the *fetch unit* is accepted. In this case, the output *ready* of the *issue queue* is asserted to implement the optimization explained in section 3.1.2. This introduces a combinational path between the input *ready* signal from the *issue logic* and the output *ready* signal of the *issue queue*, and the *issue queue* represents the only situation in which this is permitted.

Pop the first instruction When the entry pointed by the *head* counter is valid, the output *valid* signal to the *issue logic* is asserted. If the input *ready* signal from the *issue logic* is asserted too (meaning that the transaction can be completed) the *pop* operation is triggered. At the next active clock edge, the `valid` bit of the current head entry of the *issue queue* is cleared, meaning that the entry does no longer contain a valid instruction. At the same time, the *head* counter is incremented, so the next instruction in the queue is sent to the *issue logic* in the next cycle. Notice

that since instructions are fetched in program order (apart from mispredictions), the head entry of the *issue queue* always contains the *oldest* instruction. As a consequence, instructions are issue in program order from the *issue queue*, according to Tomasulo's algorithm (section 2.3.2). If the *ready* signal from the *issue logic* is not asserted[6], the handshake fails: the head counter is not incremented and the same instruction is sent to the *issue logic* in the next cycle.

Stall Thanks to the handshake protocol, stalling the *issue queue* is as simple as deassert or mask the *ready* signal from the *issue logic*. In this case, no instruction is popped and the head pointer is not updated, as it happens when the execution pipeline cannot accept any new instruction. A structural hazard is, in fact, one of the reasons to stall the issue phase. Notice that in some cases the *issue queue* stall is not mandatory but it's highly encouraged. This is true for all those instructions that modify the processor *status* or require subsequent instructions to be replayed. When those instructions are issued and processed by the *issue decoder*, a *stall* signal is generated and used to mask the ready signal of the *issue logic* from the next cycle. As mentioned at the beginning of this section, even when the issue is stalled, the instruction fetch can continue as long as the tail entry of the *issue queue* is empty.

Flush In some situations, in-flight instructions must be discarded. Besides the trivial case of misprediction, this happens when an instruction modifies the processor CSRs or the page table. Since the change takes effect when those instructions commit, all subsequent ones must be replayed, as they could have been executed before the status update. In all the previous cases, already fetched instructions that are buffered inside the *issue queue*[7] must be flushed before the correct instruction is fetched. The flush operation is controlled by a dedicated input signal generated by the main control logic of the execution pipeline according to the inputs from the *commit logic*. The *flush* control signal has the same effect that a synchronous control signal would have: at the next active clock edge the *issue queue* buffer is cleared and the head and tails counters are reset to their default value 0. Notice that it's enough to clear the `valid` field of each entry of the *issue queue* and the counters to achieve the desired result: if no entry is valid the output *valid* signal to the *issue logic* won't be asserted until a valid instruction from the *fetch unit* is pushed into the buffer.

[6] As a consequence of a structural or data hazard, or a stall required by the issue or the main control logic.

[7] and in any other structure in the execution pipeline.

3.2.3 Exception handling

It is possible that during the instruction fetch an exception is raised. During issue, the PC is used to access the *instruction TLB* to get the physical address of the next instruction. If the pointed memory location is not accessible with the current execution privileges, an *instruction access fault* exception is raised. If the requested instruction is not mapped into any page of the executing process virtual address space[8], an *instruction page fault* exception is raised instead. Also, if the memory address is not aligned to four bytes, an *instruction address misaligned* exception is raised[9]. When an instruction is raised during an instruction issue, the `except_raised_i` signal is asserted and the code associated to the raised exception[10] is produced in `except_code_i` signal by the *fetch unit*. This information is stored in the dedicated fields of the *issue queue* mentioned in section 3.2.1. In addition, the RISC-V specification requires that the offending virtual address (i.e. the PC value that raised the exception) is copied into the *machine mode* or *supervisor mode* `mepc/sepc` and `mtval/stval` CSRs. This value is already available in the `curr_pc_i` input from the *fetch unit*, so no additional fields are required in the *issue queue*. If the head entry of the *issue queue* has a registered exception, it is directly sent to the *ROB* without any further operation, and the offending PC is copied in the *result* field of the *ROB* so it can be copied in the previously specified CSRs and used by the OS exception service routine.

As mentioned before, if an exception is raised during the fetch phase, as soon as the offending instruction is issued, the *issue queue* can be stalled by the main execution pipeline control logic since all in-flight instructions would be flushed when that instruction commits. Stalling the *issue queue* saves energy since no instruction is executed in vain.

[8] The address space is defined in the page table of the current process, see [10] for details.

[9] See [16], [8] and [10] for more details about how the memory system is implemented in *LEN5* and how instructions are handled during the instruction fetch.

[10] Actually only the four MSBs of the codes defined by [16] are saved in the *issue queue* to save some register.

3.3 Issue logic

The issue logic is the first important control block of the *LEN5* execution pipeline. It implements the ISSUE and the OPERANDFETCH procedures of the extended version of Tomasulo's algorithm from algorithm 2.11. It performs the following operations accordingly:

1. If the instruction from the head entry of the *issue queue* is valid, select the proper *RS* and functional unit according to the instruction type, that is evinced by the *issue decoder*, and assert the corresponding *valid* signal. If the *issue decoder* raises an *illegal instruction* exception, the instruction is only issued to the *ROB*. Exception handling in the *issue logic* is discussed in section 3.3.2.

2. Check the *ready* signal from the assigned *RS* and from the *ROB* to see if both have an available entry where the issuing instruction can be allocated. If so, assert the output *ready* signal to inform the *issue queue* that the current instruction can be popped and a new one can be issued in the next cycle, as described in section 3.2. The issue process follows the handshake scheme from fig. 3.2a: the *ready* signal is asserted *after* the *valid* signal from the *issue queue*.

3. If needed, update the *register status* data structure with the entry of the *ROB* the instruction has been assigned to, marking the instruction destination register as *busy*.

4. Check the *register status* data structure to know if each instruction source operand is available in the RF or in the *ROB*. If so, read its values and mark the operand as ready. Otherwise read from the *register status* the entry of the *ROB* that will contain that operand as soon as it is computed by the corresponding instruction. In this case, the operand will be fetched from the CDB as soon as it will be produced by the source functional unit.

5. Send the issuing instruction to the selected entry of the *ROB*, asserting the dedicated output *valid*.

6. If no exception has been raised, send the issuing instruction and its available operands to the *RS* selected by the *issue decoder*.

The *issue logic* also performs the sign extension or zero-padding of the immediate values when required. Notice that an immediate value can be sent to the *RSs* as a second operand, without requiring any dedicated registers. From the execution pipeline perspective, an immediate value is just an operand exactly like the register values.

The *issue logic* output data signals are broadcast to all the *RSs*. Only the *RS* that receives an asserted *valid* signal will actually insert the incoming data in its

data structure. Moreover, the *issue logic* is a completely combinational block. This means it cannot be flushed or reset. Instead, it generates the control signals that are used to stall the *issue queue* if needed by the instruction being issued. In all the other cases, the *valid-ready* handshake protocol ensures that the transactions between the *issue queue*, the *ROB* and the *RS* selected by the *issue logic* are completed correctly.

A high-level block diagram of the *issue logic* is reported in fig. 3.7.

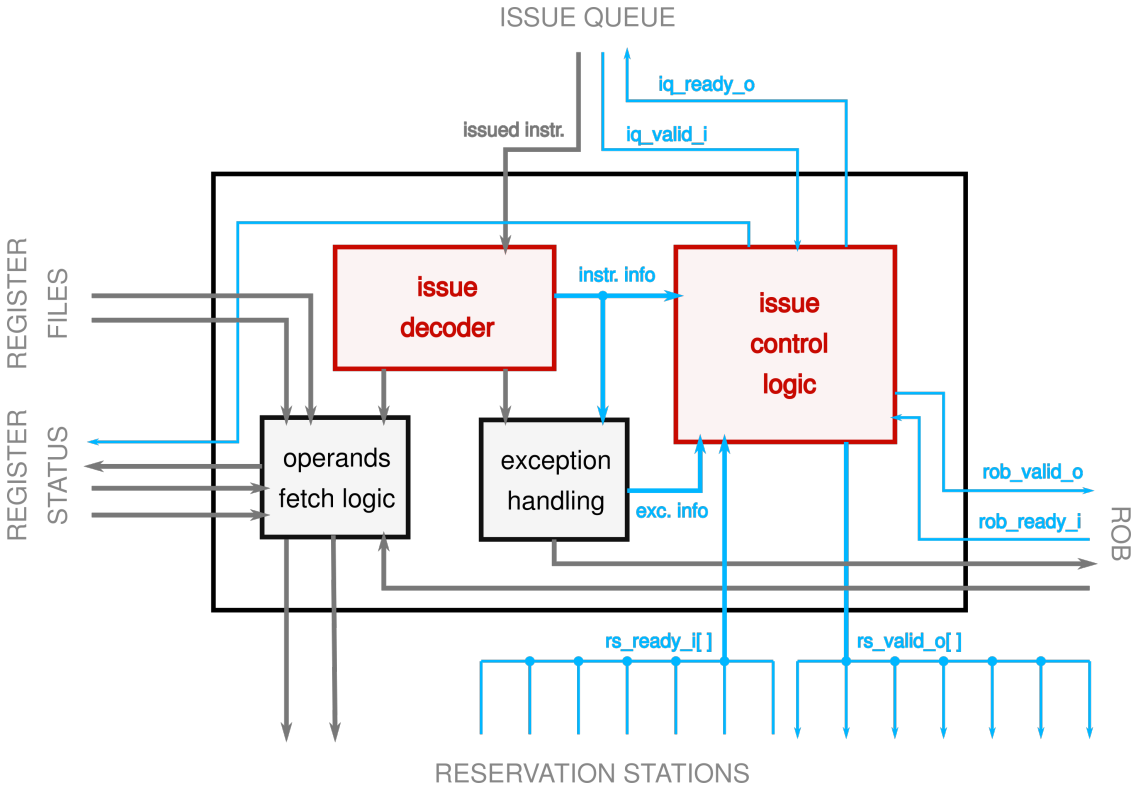


Figure 3.7: *Issue logic* block diagram. The ports shown in this diagram are not totally accurate. Please refer to the code for the complete interfaces. Since it contains only combinational logic, no clock, reset or flush signals are required.

The most important component of the *issue logic* is the *issue decoder*. The next session briefly describes its purpose and features.

3.3.1 Issue decoder

Before describing how this component works, some basic information about the RISC-V ISA and instruction format are summarized.

RISC-V uses fixed-length, 32-bit instructions[11]. Based on the type of instruction, the format can be different and so is the contained information. As an example, *store* instructions don't have any destination register `rd` unlike register instructions like ALU ones. In this case, the bits that usually contain the index of the destination register are used to store the five Least Significant Bits (LSBs) of the immediate offset that is used to generate the virtual address of the destination memory location. Similarly, not all instructions use all the possible fields for the *operation code*. The RISC-V ISA reserves at most three fields for this code:

- A 7-bit `opcode` field that occupies the seven LSBs, where the main *operation code* is contained. For example, most of the instructions that are executed by an integer ALU have the `opcode` field equal to `0110011` while in CSR instructions it is `1110011`.
- To further identify the instruction, an additional field may be present in the instruction format: `funct3`. It is a 3-bit wide field in the instruction bits 12 to 14.
- R-type instructions have another 7-bit wide `funct7` field in the seven Most Significant Bits (MSBs) to specify the operation to be performed.

Based on the instruction format and the fields employed, the remaining bits of the instruction are used to encode the source and destination register or an *immediate* value, which is a user-defined number which is used during the instruction execution without having to load it from a register. The six different instruction formats defined by the RISC-V ISA and all the details about them are reported in table 3.2. For all the details about the instruction codes and formats refer to the official specification [17].

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

Table 3.2: RISC-V instruction formats and content.

The preliminary version of the *issue decoder* of *LEN5* supports all instructions from the RV64I except for those that require the integration with the frontend,

[11]Unless the compressed instruction set extension *RVC* is implemented, in which case instructions are 16-bit wide.

like the ones modifying the program counter (`jal`, `auipc`), and those requiring full support for the *machine* and *supervisor* modes.

Starting from the `opcode` field, the *issue decoder* analyses the instruction according to the expected fields and values for that operation code. Some instructions are required to have specific values in the source or destination register fields, and others are distinguished only on this basis. As an example, the *wait for interrupt* (`wfi`) instruction differs from the *supervisor mode return* (`sret`) instruction only because of the `rs2` field, that contains `00101` in the former and `00010` in the latter. The decoder must, therefore, *conditionally* analyze the instruction fields to decide what *RS* it should be sent to and what information and data are required for its execution. If the issued instruction is not among the *known* instructions, an *illegal instruction* exception is raised. As a consequence, the *issue logic* will behave in the same way it does when an exception is raised during the fetch stage. In *LEN5* code, all the supported instructions are defined in a configuration package as described in appendix A.1. The definitions are used in the *issue decoder* and the *commit decoder*.

If the instruction is known, the *issue decoder* sets some control signals for the *issue logic*. The *issue logic* will perform the operations mentioned before according to them. Each control signal triggers a specific operation as listed below.

- An **exception raised** signal is asserted if the received instruction is not among the known ones. The *issue logic* will consider this exception only if no exceptions were raised during the fetch stage. Of course, only one exception per instruction can be processed at commit time, and earlier exceptions have higher priority, since an exceptional instruction is supposed to behave *exceptionally*, so other exceptions might be raised during decode. Remember that instructions that raised an exception during the fetch stage or the decode stage are not executed or sent to any *RS*. Instead, they are only pushed into the *ROB* together with the exception code and auxiliary data (like the offending virtual address).

- The **exception code** that corresponds to the raised exception is sent to *issue logic*. In the current implementation, only an *illegal instruction* exception can be raised during decode, unless the instruction being decoded is an `ebreak`. In this case, the corresponding *breakpoint* exception is registered.

- Some instructions don't need any additional computation until they commit. This is the case of the `fence` or some CSR instructions. When instructions of this type are decoded, **result ready** control signal is sent to the *issue logic*. As a consequence, the instruction won't be sent to any *RS* but only to the *ROB*, where it will be marked as completed so it will be committed without any additional check as soon as it reaches the head of the *ROB*. This behavior is similar to the one adopted for instructions that raised an exception.

- As mentioned before, some instructions require the execution pipeline to be flushed when they commit, the ones modifying normal instruction flow of the executing program such as `jal` and `jalr`. When these instructions are decoded, a **stall possible** control signal is asserted. The *issue logic* will *or* this signal with other possible reasons to stall the pipeline, and the resulting signal is passed to the top-level control logic of the execution pipeline, so the *issue queue* can be stalled already from the next cycle. This saves energy avoiding to execute instructions in vain as mentioned in section 3.2.3. Notice that speculative instructions never cause the pipeline to stall. This is supported by the assumption that the prediction logic is accurate enough to minimize the misprediction probability.

- According to the decoded instruction, the destination **functional unit** is selected by the decoder and communicated to the *issue logic*. This information is used to assert only the correct *valid* signal for the *RS* associated to the selected *functional unit*.

- Also the **control signals** for the destination *functional unit* are generated by the *issue decoder*. Notice that different execution units accept different control signals. If the decoded instruction is a *branch*, the type of branch is encoded in this signal. If the instruction is an ALU operation, the ALU control signals are encoded instead and so on. For this reason, the width of the *functional unit* control signal is the maximum of the ones accepted by each unit. Only the necessary portion of this data will be read and used by each *functional unit*.

- A **floating-point source operand** control signal is asserted if the current instruction must fetch its operands from the *floating point RF* instead of the *integer RF*.

- The **rs1 required** and **rs2 required** signals are set if the instruction actually needs the operands from the registers pointed by its `rs1` and `rs2` fields (see table 3.2). This information is used by the *issue logic* to issue the know if an operand must be fetched or not from the *RF* or the *ROB*.

- A similar **immediate required** signal is asserted if the instruction contains an *immediate* field that must be used during the execution.

- When an *immediate* value is needed, the format of the instruction must be known to correctly reconstruct its value according to the organization of this field reported in table 3.2. This information is carried by an **immediate format** control signal.

- The last information that is needed by the *issue logic* is whether or not the *register status* data structure must be updated for the current instruction,

that is if the current instruction will actually write a destination register in the integer or floating-point *RF*. A **register status update** signal is asserted if this is the case.

Notice that some instructions like the CSR instructions and the `sfence.vma` must be executed only during commit since they modify the processor status. The operands of these instructions are not fetched during issue. Instead, they are fetched during commit from the *RF*. Notice that the *RF* at commit time contains the most updated values since all previous instructions have committed. Moreover, no additional read ports in the *RF* are required. The *commit logic* can simply take control of the existing read ports without causing any error in the execution pipeline. At that moment, no other instruction is accessing the *RF* since the issue was stalled when the committing "special" instruction was issued. Even if this wasn't true, instructions issued after that would be unconditionally flushed, so the fact that they failed to load the right value from the *RF* has absolutely no importance. Therefore, a simple MUX controlled by the *commit logic* is inserted at the *index* input of each read port of the register file.

3.3.2 Exception handling

At this point of the execution pipeline there are only two possible situations where an instruction may have been raised: the instruction fetch process and the instruction decode. The former was discussed in section 3.2.3: the *instruction address misaligned*, *instruction access fault* and *instruction page fault* can be raised during the fetch process. The latter was mentioned above in this section. As pointed out, only two different exceptions can be raised during the instruction decoding:

- *illegal instruction* (code `0x2`)
- *breakpoint* (code `0x3`)

The *breakpoint* instruction is intentionally raised by the `ebreak` instruction. This kind of exception is also called *trap*. Traps are inserted by debuggers in the program code to interrupt the execution at some user-defined locations (*breakpoints*). Usually, when this happens the service routine, that is loaded from the memory location indicated by the `mtvec/stvec` CSR, transfers the execution control to the debugging environment that checks the current execution context (e.g. the value of variables in the registers or in the memory). When the user requires the program execution to be resumed, the debugger restores the processor status and the next instructions in program order are fetched and executed until another breakpoint is reached. From the processor perspective, a *breakpoint* exception is no different from any other: it must be processed in program order during commit, and the entire execution pipeline must be flushed before the correspondent service routine is fetched. So, when an `ebreak` instruction is decoded, the corresponding exception

is raised, and the instruction is simply allocated in the *ROB* with the *exception raised* bit asserted and its PC is copied into the dedicated field of the *ROB*. The *exception handling logic* will process the exception when the instruction reaches the head of the *ROB*[12]. No additional computation is required, so the instruction is not sent to any *RS*. Besides, the *issue queue* can be stalled as soon as the instruction is decoded, as it will be flushed unconditionally when the `break` commits.

The *illegal instruction* exception is raised whenever at least one field of the issuing instruction does not match the required format and values for a specific *operation code* as defined by the RISC-V specification [17]. Also, in this case, the instruction is not sent to any *RS*. It is only allocated in the tail entry of the *ROB* with the *exception raised* bit asserted. Its PC is copied in the dedicated field of the *ROB*.

Notice that the *issue logic* has an *exception auxiliary data* output. This is meant to contain additional information about the occurred exception whenever it is required. The content of this output is copied in the *result* field of the tail *ROB* entry when the instruction is issued. In the current implementation of *LEN5*, no auxiliary data is required by any of the exceptions that may be raised before the execution phase, as discussed above. For this reason, the auxiliary field is used once again to contain the current instruction PC. Notice that this mechanism can also be used to generate additional data for special instruction during the issue phase. For example, the `sfence.vma` instruction has different effects depending on the value of its `rs1` and `rs2` fields: they are used to identify the virtual address spaces and the virtual address values that must be prevented from executing out of order. This is particularly useful in multi-core and multi-thread implementations. What is relevant here is that the information about the value if `rs1` and `rs2` is required at commit time and must, therefore, be stored in the *ROB*. At the moment, the whole instruction is allocated in the *ROB* during issue and its source operands indexes are simply accessed during commit to decide what to do. However, a more optimized solution could make this decision during issue and encode it in the auxiliary data that are saved in the *result* field of the *ROB*. This saves a decoding step during commit and doesn't require the whole instruction to be saved in the *ROB*.

[12]Notice that the *LEN5* code does *not* include the *exception handling logic*, since the integration of the frontend and the memory system must be finalized before all the possible instructions can be properly processed.

3.4 Register status

The *register status* data structure is implemented exactly as described in section 2.4.1. Two different components are instantiated to keep the information about the *integer RF* and the *floating-point RF*. Each *register status* has as many entries as the number of ISA registers, that is 32 for both the *RFs*. Update request to the *register status* are issued using once again the AXI-like handshake protocol.

This component is accessed by the *issue logic* whenever an instruction is issued to know if its operands are available and where they can be found. It is updated during issue to register which instruction will write a specific register, and during commit to state that the value of that register is going to be available in the *RF* starting from the next cycle. As a consequence, the *register status* has two read ports to access the information about both an instruction operands during issue and two write ports to update the information about an instruction destination register during issue and commit.

Since this is a very simple component, no block diagram is provided. Its interface should be clear from fig. 3.1.

In the following sections, the content and control logic of the *register status* are discussed.

3.4.1 Register status data structure

Each entry of the *register status* contains only two fields that are briefly described below.

- **busy:** a bit indicating if the value of the associated register has been or is going to be produced by an in-flight instruction.
- **ROB index:** the entry of the *ROB* containing the instruction that will write the associated register during commit.

An example of the content of this data structure is reported in table 3.3.

The *busy* bits of the source registers of an issuing instruction are read by the *issue logic* during issue. If it is set, the corresponding *ROB index* is used by the *issue logic* to check if the requested operand has already been produced by the last instruction writing that register. If so the result can be fetched from the *ROB* entry assigned to the producing instruction. If not, the *ROB* entry is sent to the destination *RS* and the *CDB* is monitored. As soon as a *functional unit* writes on the *CDB* the index of the target *ROB* entry and the produced result, this is saved in the *RS* entry assigned to the issuing instruction.

At the same time, the *busy* bit of the issuing instruction destination register is set by the *issue logic* and the tail entry of the *ROB* is stored in correspondent *ROB index* field. This way, subsequent instructions whose operands must be fetched

Register index	busy	ROB index
0	no	-
1	yes	3
2	no	-
...	...	...
31	yes	4

Table 3.3: *Register status* content. In this example, the value of register 0 is already available in the *RF*, and no in-flight instruction is going to write it (the *busy* bit is not set). The corresponding *ROB index* is a *don't care* in this case. Instead, register 1 will contain the result of the instruction allocated in the third *ROB* entry. Any instruction using this register value as operand must check that entry of the *ROB*. If the result is not available yet, they must wait for it to be produced and sent on the *CDB*. The same is true for register 31.

from that register will access the registered entry of the *ROB* instead as described in the previous paragraph.

The *busy* bit of the destination register can be cleared when the producing instruction commits, but only if no subsequent in-flight instruction is going to write the same register. This is crucial in Tomasulo's algorithm to handle Write After Write (WAW) and Read After Write (RAW) hazards.

3.4.2 Register status control logic

The two write operations to update the *register status* data are triggered only if the *valid* signal from the requesting unit, the *issue logic* or the *commit logic*, is asserted.

During issue, no additional check is required to update the *busy* and the *ROB index* fields of the destination register of the issuing instruction. In other words, the *valid* signal simply acts like a *write enable* control signal in this case. Notice that if a previous instruction writing the same register has not committed yet, the *busy* bit might be already set. Still, the write operation takes place normally, and only the *ROB index* field is actually modified.

During commit, the *busy* bit can be cleared only if the committing instruction is the last one among all the in-flight instructions that writes its destination register. The control logic simply verify this condition comparing the head index of the *ROB*, that is the entry from which the current instruction is committing, to the *ROB index* associated to the destination register of that instruction. If they match, no instruction has modified that value since the committing instruction was issued. If they don't match, an instruction issued **after** the committing one is going to write the same destination register (WAW hazard). In this case, the corresponding *busy* bit must not be cleared, even if the committing instruction is writing its result

in the *RF*. If it was, subsequent instructions whose source operands refer to that register would fetch the outdated value from the *RF* instead of the **last** value from the entry of the *ROB* indicated by the *register status* data structure or from the *CDB*. This would cause a RAW hazard.

Both the *issue logic* and the *commit logic* have their own dedicated write port that can be used in any cycle, exactly like a write port in a traditional register file. For these reasons, write operations to the *register status* never fail. As a consequence the *ready* signal is always asserted. This is the extreme case of the situation shown in fig. 3.2b.

For what concerns read operations, there is no reason for the control logic to interfere since each read request is performed by a dedicated asynchronous read port with its own address (register index). Besides this, the output values from the *register status* are always valid, so handshake is used for read requests.

3.5 Register files

Two register files are instantiated in the execution pipeline of *LEN5*: the RISC-V ISA expects a dedicated register file for floating-point instructions if the floating-point ISA extension is supported. This should speed up the execution since fewer memory accesses are required. Even if Tomasulo's algorithm uses register renaming, the same amount of architectural registers must be provided: the implementation should be transparent to the compiler, and the code must be able to use all of the registers. The only advantage of register renaming is that instructions can be executed OoO even if they show some data dependence, without having to wait for previous instructions to be committed. An interesting fact is that Tomasulo's algorithm was also introduced to compensate for the floating-point registers shortage of the IBM System/360 family, which limited the possible compiler optimizations. Due to the limited amount of registers, many slow memory accesses were required. Tomasulo's original algorithm could hide the latency of memory instructions by executing other instructions while buffering their results in the Reservation Stations until the memory operations committed.

Both the *RFs* provide 32 registers. Integer registers are called `x0` ... `x31`, while floating-point registers are called `f0` ... `f31`. While `x0` is hard-wired to 0, `f0` is like every other register.

The RISC-V specification [17] define a calling convention for integer registers that associates each register to a specific programming function. For example, register `x2` is also called `sp` since it is conventionally used to store the stack pointer, while register `x1` is also called `ra` because the convention uses it to store the return address after a procedure call implemented with the *jump-and-link* instruction (`jal`).

In RV64I the integer registers are 64-bit wide. Since *LEN5* supports both the single-precision (F) and double-precision (D) floating-point extensions, also the floating-point registers are 64-bit wide. So, from the hardware point of view, the two register files are completely identical except for the hard-wired `x0` integer register.

Each *RF* has one synchronous write port to write the produced result into the destination register during commit and two asynchronous read ports to read the operands during issue. The input *valid* signal from the *commit logic* has the same purpose of a *write enable* control signal for the *RF*, as it happens in the *register status* update during issue. The register values can be read in any cycles, and it's always valid, so the output *ready* signal is always asserted and no output *valid* signal is required for read accesses.

As for the *register status* structure, the very simple block diagram of the *RFs* is omitted to save space and money when printing this document. Please refer to fig. 3.1.

3.6 Reservation Stations

The Reservation Stations are, as explained in section 2.3.2, basically buffers where the operands are hold before the instruction begins its execution in the *functional unit* associated with its *RS*. Each *RS* has a certain number of entries that can be configured by the user in the configurations files as described in appendix A.1. In *LEN5* there is a *RS* for each *functional unit* and one *functional unit* for each type of instruction. However, multiple *functional unit* and *RSs* of the same type are fully supported. Moreover, an arbitrary number of *RSs* and functional units can be instantiated. The case limit is to have many *RSs* per type with a single entry and a dedicated *functional unit*. Of course the scaling of performance is not linear with the number of functional units unless other parts of the processor are modified too: multiple units can increase the IPC over one instruction per cycle, meaning that the frontend and the memory system must support a higher instruction throughput, and the *issue queue*, the *issue logic* and the *commit logic* must be modified to support the issue and commit of multiple instructions per cycle. Another limit is represented by the single *CDB*. To maintain a maximum theoretical IPC greater than one, more than one result must be written to the *ROB* in each cycle. This requires more than one *CDB*. A more detailed analysis of possible improvements is available in section 5.3.2 and section 5.3.2.

For what concerns the *functional units*, there is no constraint about the latency or the depth of the pipeline. The only requirement is that the *functional unit* supports the *valid-ready* AXI-like handshake protocol both for the beginning of execution when the operands are sent by the *RS* to the *functional unit*, and the execution completion, when the result is sent by the *functional unit* to the *RS*. Besides, the *functional unit* must accept also the index of the source entry of the *RS* when it receives the operands. The same index must be produced together with the result when the execution completes. This is required to support units that can execute more than one instruction at the same time, even OoO. If not, a simple register can be employed to store the index every time a new instruction is executed. As an example, the *memory system* of *LEN5* can handle an additional *data cache* access request of the first one resulted in a *cache miss*, and even a third one of the second missed too (*miss-under-miss*). Therefore, the memory system answers the *load buffer* requests with an order that is different from the one in which they are issued, and the index of the source entry of the *load buffer* is returned together with the loaded value. In fact, the *load buffer* is just another *RS*. However, both the *load buffer* and the *store buffer* have additional constraints and interactions that make them much more complex than the other *RSs*. So, they are instantiated in a standalone unit, the *load-store unit* and discussed separately in section 3.7.

RSs, together with the AXI-like handshake protocol, allow a significant level of customization and modularity in the execution pipeline while keeping the general complexity quite low. These aspects meet the main motivations that push towards

Reduced Instruction Set Computer (RISC) systems in general. *RSs* are one of the main reasons why Tomasulo's approach was chosen to implement a RISC-V processor.

As repeated many times in this document, *RSs* are one of the key components to handle data hazards between instructions: all the *RSs* are connected to the *CDB* both to read and write data. When a result is written on the *CDB*, every other in-flight instruction can save it if they need it as a source operand. This ensures that an instruction whose operands were not ready during issue has always access to the most updated values of its operands while respecting RAW dependences. When all the required operands are available in one entry, they are sent to the *functional unit* unit as soon as it is free (no structural hazards) and the instruction is executed.

Apart from the *load buffer* and the *store buffer*, the *RSs* included in this first version of the execution pipeline of *LEN5* are listed below. Each *RS* takes its name from the associated *functional unit*. The block composed by a *functional unit* and its *RS* is referred to as *execution unit*.

- **ALU *RS*:** the *RS* that feeds the integer ALU. This is the standard *RS* for arithmetic units, which is detailed in section 3.6.1.
- **Multiplier *RS*:** the *RS* where *multiply* instructions belonging to the integer *Multiply* ISA extension (RV64M) are hold before they are executed in the associated integer multiplier. This *RS* and the next two (divider and floating-point) are just two instances of the previous one. Only the associated *functional unit* and control fields are different.
- **Divider *RS*:** the *RS* where *divide* instructions belonging to the integer *Multiply* ISA extension (RV64M) are hold before they are executed by the associated integer divider.
- **Floating-point *RS*:** the *RS* where the instructions from the single and double-precision floating-point RV64F/D ISA extensions are hold before they are executed by the FPU[13].
- **Branch unit *RS*:** the *RS* where *branch* instructions are buffered. It differs from the previous *RSs* in that it is compiled in-order. The reason for this is discussed in the dedicated section 3.6.2.

The *RS* types provided in *LEN5* and summarized in the list above should be able to support all the *frozen* RISC-V ISA extensions from [17]. Custom instructions

[13]In this first implementation of *LEN5*, no FPU is included, since its design might represent another master thesis alone. The focus of this project is on the execution core itself and not on the surrounding logic like the arithmetic units.

and execution units may need ad-hoc *RSs*, but in general, the principles on which the provided *RSs* are based should hold also in possible future extensions.

Figure 3.8 reports the block diagram of a generic *RS*.

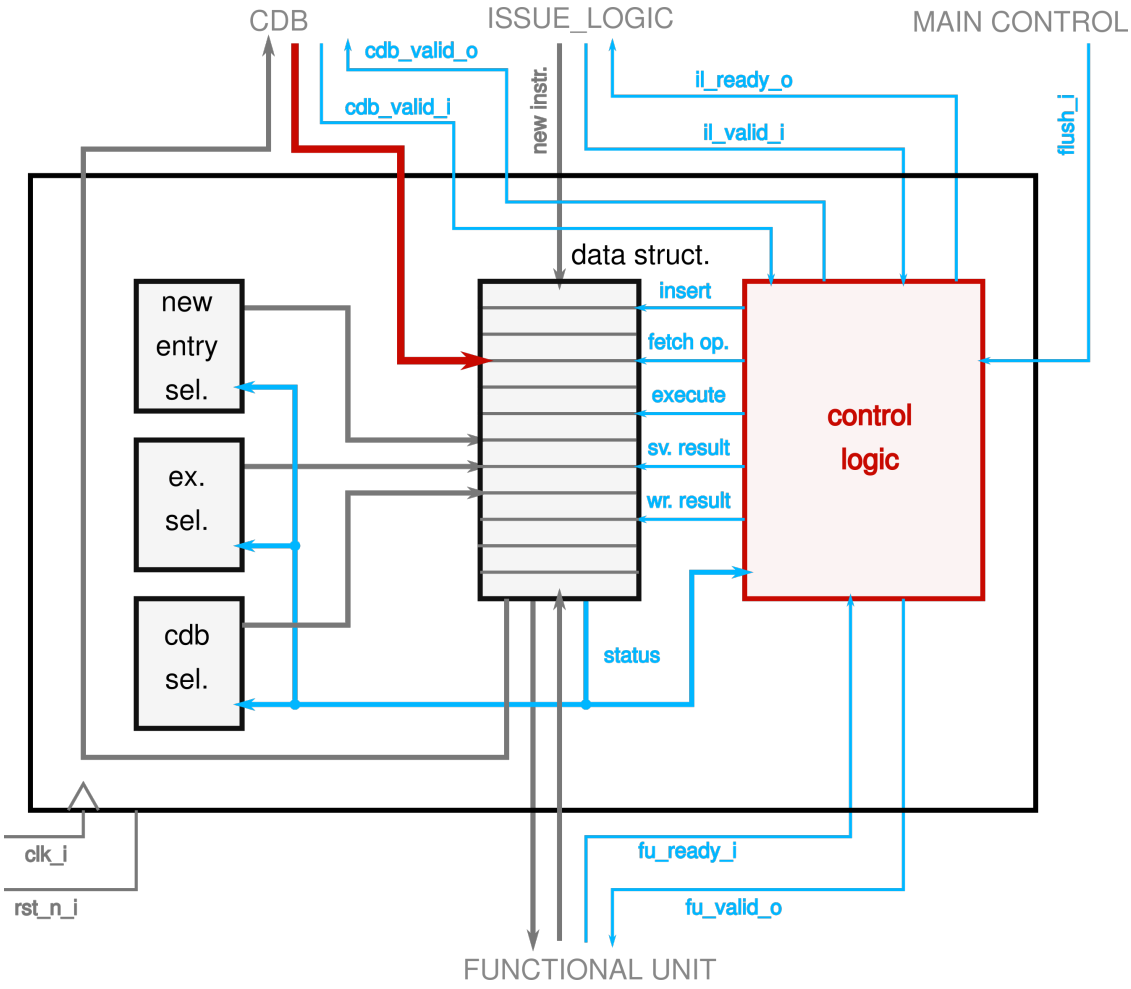


Figure 3.8: *Reservation station* block diagram.

Before proceeding to the details of each operation that can be performed in a *RS*, the content of a generic arithmetic *RS* and of the *branch unit RS* are reported in the following sections.

3.6.1 Arithmetic RS data structure

Arithmetic operations require two source operands that can be both register values or one register value and one immediate operand. In the latter case, the immediate value can be stored in the same *RS* field dedicated to the second operand. The sign extension or zero-padding is performed by the *issue logic*. Each entry of an

arithmetic *RS* contains the data and status fields listed below.

- `valid`: a status bit indicating if the entry contains an instruction that hasn't completed or written its result to the *ROB* yes.
- `busy`: a status bit indicating if the contained instruction is being executed by the *functional unit*.
- `[entry age]`: an optional field that is provided only if the *age based selectors* are enabled in the execution pipeline. It is reset to zero when a new instruction is inserted in the entry and incremented each time an instruction is inserted in a different entry, as explained in section 3.1.4
- `fu_ctl`: contains the control signals for the *functional unit*, such as the *operation code* for the integer ALU.
- `rs1_ready`: a status bit indicating whether the first operand is available in the `rs1_value` field or not. If the operand was fetched during issue by the *issue logic*, this bit is set when the instruction is inserted in the *RS*. Otherwise, it is set when the operand value is read from the *CDB* as described in section 3.6.3.
- `rs1_idx`: the index of the *ROB* entry containing the instruction that will produce the first source operand for the current one. Its value is used to monitor the *CDB*: if it matched the *ROB* index transmitted in the *CDB*, the operand value is saved in the `rs1_value` field.
- `rs1_value`: the actual value of the first operand of the instruction.
- `rs2_ready`: same as `rs1_ready` but regarding the second operand. If this is an immediate, this bit is set when the instruction is inserted.
- `rs2_idx:` same as `rs1_idx` but regarding the second operand.
- `rs2_value`: the actual value of the second operand of the instruction. If it is an immediate, its sign-extended or zero-padded immediate value from the *issue logic* is stored here.
- `res_ready`: a status bit indicating whether the result has already been computed by the *functional unit* (i.e. if the instruction has completed its execution).
- `res_idx`: the index of the *ROB* entry where the result of the contained instruction must be stored during the write result phase. This value is sent on the *CDB* together with the result itself. Instructions whose source operand indexes match this value can read and save the result in the operand value fields of their *RS* entry.

- `res_value`: the actual result produced by the *functional unit*.
- `except_raised`: a status bit indicating if an exception was raised during execution. It can be set when the *functional unit* completes the execution of an instruction.
- `except`: the exception code associated with the raised exception. Its value is sent to the *ROB* during the write result phase so it can be used to handle the exception when the instruction commits (i.e. it reaches the head of the *ROB*).

An example of the content of a reservation station is provided in table 3.4. In the reported example, entry number 2 has already been executed but it has not written its result in the *CDB* entry 4 yet. So the first operand of the last entry, that must be fetched from that entry of the *ROB*, is not yes available preventing this instruction to be executed. When entry 2 writes its result on the *CDB*, the last entry will be able to fetch it. Finally, entry 0 has its operands ready and it's waiting to be selected for execution.

#	v	bsy	fu_ctl	rs1_r	rs1_i	rs1_v	...	res_r	res_i	res_v	ex_r	ex
0	yes	yes	sub	yes	3	10	...	no	2	-	no	-
1	no	-	-	-	-	-	...	-	-	-	-	-
2	yes	no	add	yes	1	-	...	yes	4	13	no	-
...	...	...	...	...	...	...	...	...	...	...	...	...
n-1	yes	no	or	no	4	10	...	no	5	-	no	-

Table 3.4: *Reservation station* data structure content example. The fields containing the information about the second operands have been omitted here to save space.

3.6.2 Branch unit RS data structure

As mentioned before, the *RS* holding *branch* instructions to be executed in the *branch unit* differs from all the others since it must be compiled in-order like a FIFO buffer. The reason is to be found in the frontend. The *gshare* branch predictor combines information about the specific *branch* instruction found in an associative table with the global branch history. If *branch* instruction were allowed to executed OoO in the execution pipeline, the history table would lose coherence with the program order of branches (the *trace*). Besides this, since also *branch* instruction could be speculative, executing them OoO could produce different results in misprediction detection. Imagine that a branch is predicted as taken and issued to the execution pipeline. Suppose its operands are not ready at issue time, so the *branch* is allocated in the *branch unit RS* waiting for its operands to be produced on the *CDB* by some other *RS*. At this point, imagine that another *branch* instruction is issued, this time having its operands ready since the issue phase. Also, this second *branch* is inserted in the *branch unit RS*, and unlike the first one is marked as ready to

execute. Notice that since this second *branch* depends on a previous *branch*, it is speculative. When it is executed, the *branch unit* might detect a misprediction. As a consequence, the *issue queue* is stalled and the branch predictor is informed that a misprediction occurred for that instruction PC, so the tables are modified accordingly. In the next cycle, also the first *branch* receives its operands becoming ready for execution. It might happen that also the first *branch* was mispredicted, so the second *branch* should not have been executed. This means that the predictor tables contain wrong data about the second *branch* instruction and about the global history. Those changes cannot be reverted. Notice that this doesn't have any consequence on the correct program flow since the *branch* instructions are still processed in-order during commit[14]. So as soon as the first *branch* commits, the entire pipeline and the *issue queue* are flushed and the correct program counter is loaded, regardless of the outcome of the second *branch* instruction. So the only consequence of executing branches OoO is a not negligible loss in the predictor accuracy. A second solution to solve this issue is updating the predictor tables only during commit, when the pipeline is flushed in case of misprediction. Also, in this case, the *issue queue* can be stalled as soon as a misprediction is detected by the *branch unit*. Because of how the *branch unit* and the frontend of *LEN5* are implemented and connected, this second solution was not adopted. Instead, the *branch unit RS* is managed as a FIFO. For more information about the *branch unit* and the frontend of *LEN5* please refer to [3].

Besides the head and tail pointers, two additional counters are employed to select the next instruction to be executed and the next one to accept the branch result from the *branch unit*. Each counter is incremented as soon as an instruction performs the corresponding operation (insert, execute, write result, pop).

The information about the actual *branch* outcome produced by the *branch unit* is stored in a dedicated field of the *RS*. Before being written on the *CDB*, this bit is padded with zeroes so it can be sent in the *value* field of the *CDB* and stored in the *result* field of the *ROB*. When the instruction commits, this field is checked by the commit logic to flush the pipeline a misprediction is registered.

The fields of the *branch unit RS* entries are listed and briefly described below.

- `valid`: a status bit indicating if the entry contains a *branch* instruction that hasn't been processed by the *branch unit* yet.

- `busy`: a status bit indicating if the contained instruction is being executed by the *branch unit*.

[14]Remember that the execution pipelined flush must be performed only on instructions issued *after* the *branch* instruction at the beginning of the basic block. So the flush cannot be issued during the execution of the *branch*, because even if *branch* are executed in-order they do not maintain the original order with respect to the other in-flight instructions.

- `branch_type`: contains the encoded type of branch[15].
- `rs1_ready`: same function as in the arithmetic *RS* in section 3.6.1.
- `rs1_idx`: same function as in the arithmetic *RS* in section 3.6.1.
- `rs1_value`: the actual value of the first operand of the *branch* instruction instruction.
- `rs2_ready`: same as `rs1_ready` but regarding the second operand.
- `rs2_idx:` same as `rs1_idx` but regarding the second operand.
- `rs2_value`: the actual value of the second operand of the *branch* instruction instruction.
- `imm_value`: besides the two register operands also accept an immediate value that is used as offset for the PC. Its value is extracted from the instruction itself, so it's available since the issue phase.
- `pred_pc`: the PC of the *branch* instruction allocated in the current entry. It is used to update the predictor tables.
- `pred_target`: the *predicted branch target address* from the frontend branch predictor. It is compared to the computed branch target address to detect a target misprediction.
- `pred_taken`: the predicted *branch* outcome: 0 means predicted taken, 1 means predicted not taken.
- `res_idx`: the index of the *ROB* entry where the *branch* instruction was allocating at issue time, and where the *branch unit* outcome saved in the `mispredicted` field must be stored using the *CDB*.
- `res_ready`: a status bit indicating if the misprediction information was already produced by the *branch unit* and store in the `mispredicted` field.
- `mispredicted`: the actual outcome of the *branch unit*. It corresponds to the `res_value` field in the arithmetic *RSs*.

As it can be evinced from the list above, the content of this data structure has only little variations with respect to the one of an arithmetic *RS* shown in the table 3.4 example.

[15]The possible types are `beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`. For additional information about *branch* instructions refer to [17] and [8].

The source operands in the *branch unit RS* are fetched in the same way they are in the arithmetic *RSs*, through a Content Access Memory (CAM)-like parallel access port. Also, the other operations are similar to the ones described in section 3.6.3, with the only difference being that counters are used instead of selectors. So the control logic is totally similar too.

3.6.3 RS control logic

All the *RSs* in the *LEN5* execution pipeline follow the control paradigm discussed in section 3.1.2. A combinational network is used to *issue* a certain operation on a given entry of the *RS* data structure based on the input, handshake and status signals. The operation takes effect synchronously on the next active clock edge. Another small combinational network produces the output control and handshake signals for the *functional unit* and the *CDB*. In general, instructions complete their execution and write their results to the *ROB* OoO. So instructions are popped from the *RS* OoO too. This means that these structures cannot be addressed with simple sequential pointers like the *issue queue*. Instead, some *selectors* are used. Usually, there is a dedicated selector for each operation that the *RS* can issue on one of its entries. The selector evaluates which are the suitable entries for that operation and picks one of them according to its internal priority scheme (see section 3.1.4). Each *RS* must deal with two main types of events: generating a request regarding the selected entry or process a request or an answer from an external unit. The possible operations in a generic arithmetic *RS* are introduced in the following paragraphs. The *branch unit RS* only implements a subset of the listed operations. Notice that even if multiple read and write ports are needed, they often act on different fields of the *RS* station data structure. For this reason, the data structure can be seen as made of multiple independent register files with a single read and/or write port instead of a single register file with many read and write ports. Another important aspect of the access to the *RS* data is that the operand fetch operation during the writeResult procedure in algorithm 2.9 is performed in parallel on all the entries of a *RS* based on the *ROB* index of the operands and of the result being transmitted. This means that a CAM-like parallel write port must be implemented in each *RS*. A structure like this might have a significant weight on the overall power consumption. Also, the fact that the *CDB* broadcasts data to all the *RSs* contributes to increasing power. However, this type of access is a key element in Tomasulo's algorithm that cannot be removed. All the advantages brought by a simple control and distributed book-keeping operations should at least partially compensate this kind of parallel access ports.

Overall two read ports, two write ports and one parallel access port are implemented in each *RS*.

Insert a new instruction When a valid instruction is available in the head entry of the *issue queue* and both the *ROB* and the assigned *RS* are not full, the instruction is inserted in the selected entry of the *RS*. This entry must be selected among those no longer containing a valid instruction, meaning that the contained instruction has already completed its execution and written the result in the *ROB* using the *CDB*. When this happens the `valid` bit of the entry is cleared. All empty (i.e. not valid) entries of the *RS* are suitable for the incoming instruction, so the *selector* is a simple priority encoder. It provides the address of the first empty entry to the **first write port** of the *RS*. At the next active clock edge, the data from the *issue logic* is stored in the selected location. If the *issue logic* has already fetched an operand, the associated `ready` bit is asserted and its value is stored in a dedicated field. All the other status bits are cleared so the instruction can be selected for execution when it has all of its operands available and the *functional unit* can accept an execution request.

Retrieve the operands from the *CDB* This operation doesn't require any handshake process. Each entry containing a valid instruction that hasn't been executed yet monitors the *CDB* data to fetch its missing operands. When some other *RS* sends on the *CDB* a result to be written in *ROB* entry that the instruction must fetch its operand from, the result is copied in the dedicated field and it is marked as `ready`. This operation can be performed on all the entries whose *destination ROB entry* match the one on the *CDB*. As a consequence, this requires a CAM-like **parallel access port** to all the entries of the *RS*. If the instruction writing the result raised an exception during its execution, the result is not copied. When this happens, the entire execution pipeline will be flushed at commit, so there's no point in executing subsequent instructions or fetching their operands.

Send an instruction to the *functional unit* Once an instruction has fetched both of its operands, it becomes available to begin its execution in the *functional unit*. In each cycle, more than one instruction may be ready for execution, so a *selector* from section 3.1.4 is employed. The output index of the selector is used to drive the **first read port** of the *RS* that is used to provide the operands to the associated *functional unit*. If a simple priority encoder is used, it is not guaranteed that the oldest instruction that has its operands ready is executed first. This might introduce some latency in the execution of subsequent instructions that depend on that result. This is the main reason for the *age based selector* to be introduced. However, as mentioned in section 3.1.4, this has a great timing and complexity penalty, that might lead to overall worse results in the system. Regardless of the selector type, it chooses only among instructions that:

- Are valid: `valid` status bit set.
- Have both their operands ready: both `rs1_ready` and `rs2_ready` set.

- Have not been sent to the *functional unit* yet: `busy` bit clear.
- Have not been executed yet: `res_ready` bit clear.

If a suitable instruction is found *request* to the *functional unit* is performed: the output *valid* signal to the *functional unit* is asserted, and the operands are transmitted together with the index of the source entry of the *RS* (the output of the selector), as explained at the beginning of this section. If the *functional unit* can accept the execution request, the `busy` bit of the source instruction is set, so it won't be selected again in the next cycles.

Save the result from the *functional unit* When the *functional unit* completes the execution, it sends an *answer* to the source *RS* asserting its *valid* signal. The index of the source entry if the *RS* is returned too. The *functional unit* never waits for the *RS* to be ready to process the answer, since the instruction has already been allocated, and only one instruction from the *RS* can be executed by the *functional unit* in a given cycle. Therefore, the output *ready* signal to the *functional unit* is always asserted. When the answer is received, the result is stored in the *RS* entry pointed by the return index (**second write port**), and it is marked as ready so the instruction can proceed to the write result step as soon as the *CDB* is available. If an exception was raised during the execution in the *functional unit*, the `except_raised` status bit of the instruction entry is set and the exception code from the *functional unit* is copied in the dedicated field. If additional information about the exception is produced by the *functional unit*, they are stored in the `result` of the instruction entry. Doing so the additional information will be sent on the *CDB* and written in the `result` field of the *ROB*, so it can be used during commit by the *exception handling logic*.

Send an instruction result on the *CDB* After an instruction has completed its execution and the result has been received from the *functional unit*, the result can be sent to the *ROB* and to all the other in-flight instruction waiting for that result. So, the result is written on the *CDB* together with the *ROB* entry where the producing instruction was allocated during issue. If an exception was raised during execution, its code is sent to the *CDB* too so it can be stored in the *ROB* and processed at commit time by the *exception handling logic*. Since the *CDB* is shared among all the *RS* in the execution pipeline it might not be available in any cycle for a given *RS*. This means that many instructions in the *RS* might have their results ready to be sent on the *CDB* in a certain cycle. For this reason, another *selector* is used to choose one instruction among those that have their result ready (i.e. those that have completed their execution). The output index of the selector is used to drive the **second read port** of the *RS*, connected to the *CDB*. If a simple priority encoder is used as a selector, it is not guaranteed that the oldest instruction that has completed its execution writes its result first.

Notice that another possible solution is to save the index returned by the *functional unit* and use it to select the next entry that writes on the *CDB*. While this saves a selector and the result field in the *RS* entries, it is not always the best solution. In fact, the *functional unit* must be stalled until the previously executed instruction writes its result on the *CDB*. If the *functional unit* has a single-cycle latency, there is no disadvantage in this solution, since the execution unit is still able to execute one instruction per cycle. However, if the *functional unit* is pipelined or has a latency greater than one, the cycles during which the *CDB* is not available cannot be exploited to execute new instructions unlike the first solution, because with this solution the *functional unit* is stalled during those cycles. Instead, using the first solution, as soon as the *CDB* is available all the produced results can be written in a *burst* until the *CDB* is assigned to a different unit or until there are no more completed instructions in the *RS*. It can be easily proven that in the best scenario the overall improvement in the execution throughput of the entire execution pipeline with the first solution is directly proportional to the average latency of the *functional units* that share the *CDB*. Since it's common to have pipelined *functional units*, the first solution has been adopted in *LEN5*, having the only downside of employing an additional selector and an additional field in each *RS*. Remind that priority encoders are used by default, so the impact of this components is negligible in the entire system.

In both cases, when the *CDB* is available and assigned by the *CDB arbiter* to the current *RS* (i.e. the input *ready* signal from the *CDB arbiter* is asserted), the transaction can be completed and the instruction can be popped from its entry in the *RS*. This is simply done by clearing its `valid` status bit. Doing so the entry is no longer selected for execution or to send its result on the *CDB*, while it becomes available to possibly insert a new instruction in the next cycle.

3.6.4 Exception handling in RSs

For what concerns exceptions, it is important to remind that an instruction that raised an exception during fetch or issue is not executed. So it is not sent to a *RS* by the *issue logic*. Instead, it is only allocated in the *ROB*. For this reason, the *RS* don't receive any information about occurred exceptions by the *issue logic*. If an exception is raised during the instruction execution in the *functional unit*, the exception is registered in the *RS* entry where the instruction is allocated, and transmitted to the *ROB* over the *CDB* as it is done when a valid result is produced. Possible exceptions are, for example, the *divide by zero* and *floating point overflow* exceptions that might be raised by the FPU.

3.7 Load-store unit

The *load-store unit* is probably the most complex component of the *LEN5* execution pipeline, working as an interface between this and the *data* part of the *memory system*, namely the *DTLB* and the *data cache*. It contains three main components: the *load buffer*, the *store buffer*, and the *virtual address adder*. The *load buffer* and the *store buffer* can be seen as the *RSs* dedicated to *load* and *store* instructions respectively. They have been separated from the ordinary *RSs* since they differ quite a lot from the others in how they execute instructions and how they interact between them and with the *ROB*.

Before each of these three main components is detailed, some concepts about the memory system and the memory instructions must be recalled.

Virtual Memory VM is the mechanism used by OSs to *abstract* the concepts of memory for the different processes that are concurrently executing on top of it. In particular, each process *sees* the memory as if it was the only executing process on the CPU, and it had all the memory available for its operations, regardless of the size of memory or its organization, like the presence of external storage devices. The OS holds a table for each running process that *maps* each virtual memory address into a physical memory address that is physically available in the system *main memory*. Each process is assigned a *virtual address space* and possibly an Address Space Identifier (ASID) to distinguish it from the other address spaces. The address space contains a set of *pages*, each of which contains contiguous addresses that the process can use. Notice that when VM is enabled also the PC is a virtual address that must be translated into a physical one by the Memory Management Unit (MMU).

Page table The page table is a tree structure that performs the address translation in subsequent steps. The virtual address is divided into slices that are used to access different levels of the page table tree. Usually, the LSBs of the virtual address are not translated and are used as offset in the physical page selected by the highest order slices. A portion of the page table can be transferred to a hardware structure to accelerate the translation process. The data structure that contains this portion of the page table is called Translation Lookaside Buffer (TLB), and acts as a *cache* for the page table. To fetch the correct entries of the page table, the *root page table address* (the base address of the page table) must be known to the processor. It is usually stored in a CSR that is updated on a context switch. Notice that if no ASID is used, the old portion of the page table loaded in the TLB must be flushed on a context switch. Otherwise, the virtual addresses from the new process are translated following the mapping of the old process (homonyms). If ASIDs are supported by the processor, they are used to tag each entry of the TLB, so if a virtual address from the new processor access an entry of the page

table of the old processor in the TLB, a TLB *miss* occurs, and the requested entry from the new process page table is fetched from the memory, so the address can be correctly be translated.

Page fault The page table also contains information about what pages are currently in the main memory. If an instruction requests a virtual memory that is not mapped on a physical address of the main memory, a *page fault* exception is raised, and the corresponding exception handling routine of the OS is called so that the requested page is fetched from the secondary storage and the page table of the process is updated accordingly.

Synonyms and homonyms In principle, the mapping is not unambiguous. Two virtual addresses that are mapped to the same physical address are called *synonyms* or *aliases*, while different physical addresses that are mapped by the same virtual address are called *homonyms*. Homonyms are a direct consequence of virtual memory: different processes (i.e. different ASID) can use the same virtual address to access a different physical memory location. However, they must be supported by the processor memory system (the *cache* in particular) and by the OS, so that instructions from different processes don't access the wrong data. Homonyms within the same address space are possible only when VM is managed in (fancy) software. Otherwise, since the virtual address is used to directly access the page table, each virtual address can be mapped to a single physical address.

RISC-V, as described in [16] and [8] supports multiple paging schemes. The *LEN5* execution pipeline supports both the Sv39 and Sv48 paging schemes. Of course, also *bare* memory access is supported. In this case, the address translation in the *DTLB* is skipped, and the virtual address is directly used to access the *data cache* without any further check. The used scheme, as well as the *root page table address* and the ASID of the current process are stored in the `satp` CSR. The memory system of *LEN5* implements the Sv39 paging scheme. This supports 56-bit physical addresses (2^{26} GiB) and a three-level address translation process. The 12 LSBs of the virtual address are not translated and from the *page offset*. The *cache* is organized on two levels. The *L2 cache* is shared between instructions and data, while the *L1 cache* is divided in the *instruction L1 cache* accessed by the frontend using the PC as virtual address and the *data L1 cache* (or simply *data cache*) that is accessed by the *load-store unit*. A full-associative *DTLB* performs the address translation implementing the three-level translation scheme and supports ASID tags. This means that the OS is not required to insert an `sfence.vma` instruction after a context switch, because homonyms are handles without requiring a flush of the *DTLB*. The *DTLB* is accessed **before** the *data cache*, and so the *data cache* is physically indexed and physically tagged since only the physical address is used during cache requests. This solves the problem of aliases (synonyms) within the same address space that arises when virtually indexed and physically tagged *data*

cache systems are used.

The block diagram of the *load-store unit* is shown in fig. 3.9.

The following section analyses the main features of the *virtual address adder*, the *load buffer*, and the *store buffer*. The overall organization and control logic of the two buffers are similar to the ones of the other *RSs* from section 3.6, but here additional checks are required to guarantee that data hazards in memory are handled correctly, and many optimizations are performed to reduce the number of *DTLB* or *data cache* accesses required. The focus is on these differences and the interface with the memory system.

3.7.1 Virtual address adder

The *virtual address adder* is the component of the *load-store unit* responsible for the virtual address computation starting from the `rs1` and `imm` fields of the I-type *load* instructions and the S-type *store* instructions (see table 3.2 for the complete instruction formats). It is, in fact, a simple 64-bit adder that also checks if the produced virtual address is compliant with the current memory system configuration that is read from the `satp` CSR. Notice that here the value of the CSR is accessed during execution, that is performed OoO. That is the reason why the execution pipeline is stalled after a CSR instruction is issued: all subsequent instructions using that CSR must wait for the CSRs to be updated.

In the *load-store unit* of *LEN5* there is only one *virtual address adder* shared among the *load buffer* and the *store buffer*. The reason is that the maximum IPC of the single-issue execution pipeline is one instruction per cycle. So even if only *load* and *store* instructions are issued, on average only one *load* or one *store* can be executed per cycle. So even under the assumption that no delay is introduced during memory accesses, there would be no need for two dedicated adders. However, since the original reciprocal order of *load* and *store* instruction is not preserved in general, there might be two concurrent requests from the buffers in the same cycle. For this reason, the *virtual address adder* is connected to a two-way arbiter. By default, the priority arbiter described in section 3.1.3 is instantiated, with the precedence given to the *store buffer*. This means that every time a request by the *store buffer* is received, that request is processed regardless of the request from the *load buffer*. So in case of conflict, the arbiter asserts only the *ready* signal for the *store buffer*. The same solution is adopted also for the *DTLB* and the *data cache* requests. The reason for can be found in the implemented *store-to-load* forwarding mechanism described in section 3.7.2.

Exception handling As explained in the introduction of section 3.7, the *LEN5* execution pipeline supports the Sv39 and the Sv48 VM paging schemes. When these schemes are used, the effective width of the virtual address is limited to either 39 or 48 bits, which must occupy the lower portion of the 64-bit address. The MSBs

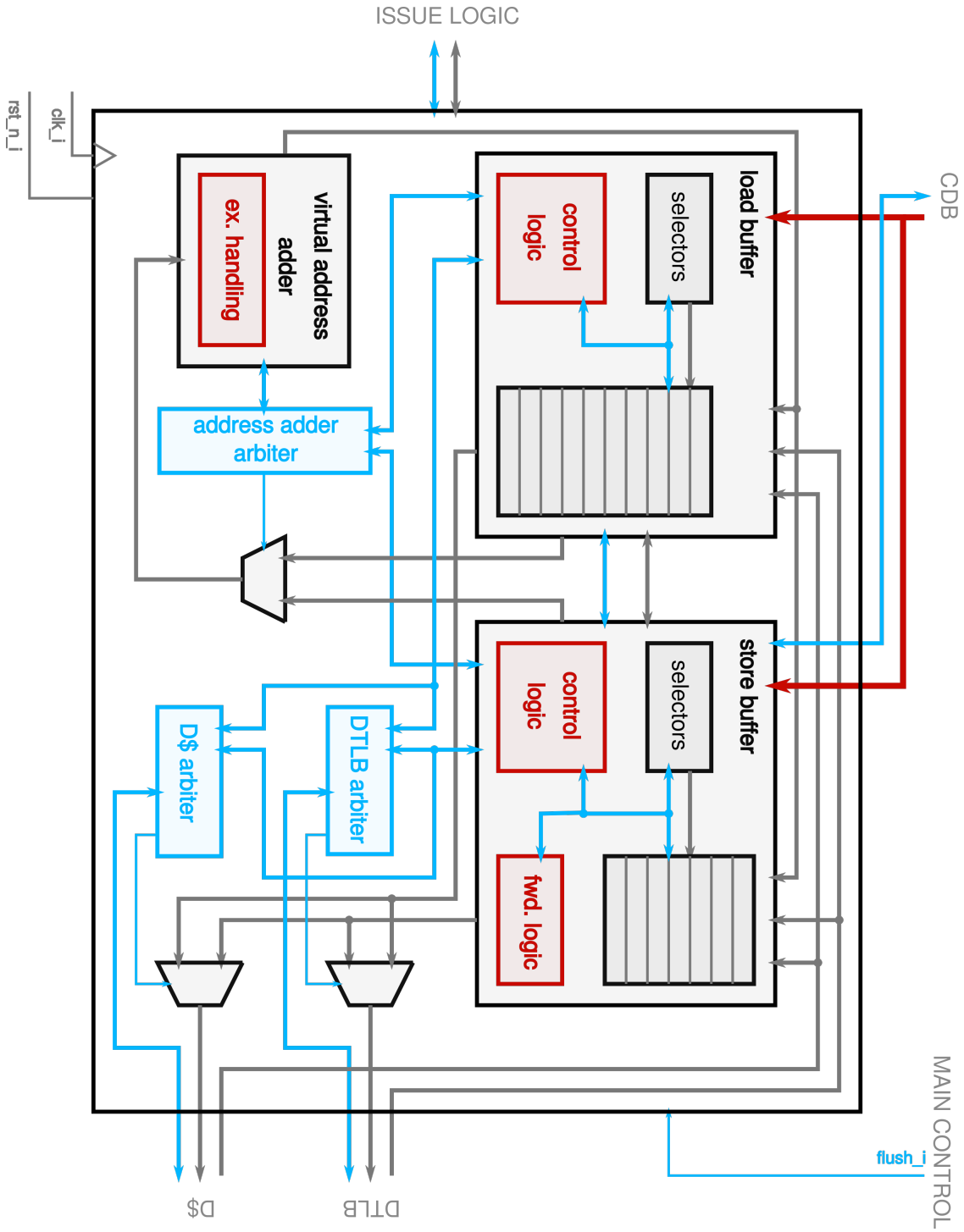


Figure 3.9: *Load-store unit* block diagram.

must be copies of the last valid bit of the virtual address. In other words, the 64-bit virtual addresses produced by the *virtual address adder* have limited ranges:

- Sv39:
  - `0x0000_0000_0000_0000` to `0x0000_003f_ffff_ffff`
  - `0xffff_ffc0_0000_0000` to `0xffff_ffff_ffff_ffff`
- Sv48:
  - `0x0000_0000_0000_0000` to `0x0000_7fff_ffff_ffff`
  - `0xffff_8000_0000_0000` to `0xffff_ffff_ffff_ffff`

The *virtual address adder* checks if the MSBs are equal to bit 38 or bit 47 based on the VM mode registered in the `satp` CSR. If the produced virtual address does not fall into one of the previous ranges, a *page fault* exception is communicated to the buffer that performed the address computation request. In the *load buffer* this exception is translated in a *load page fault* or a *store page fault* exception when it is received by the *load buffer* or the *store buffer* respectively. If VM is disable (i.e. the *bare* paging scheme is set in `satp`), no check is performed on the produced virtual address and so no *page fault exception* is produced. Besides this, another check must be done to verify that the produced virtual address is naturally *aligned* according to the type of load and store instructions. In RV64, eleven memory instructions exist, that can be classified based on the number of *bytes* they read or write. The virtual address of the target set of bytes must be aligned accordingly before being sent to the *DTLB* for address translation. Each type of memory access must satisfy the following requirements:

1. `lb` (*load byte*), `lbu` (*load byte unsigned*), `sb` (*store byte*): operate on a single byte, that can be any in the address space. Since the memory is byte-addressed, this means that there are no constraints on the value of the LSB of the virtual address (i.e. it can be either 0 or 1).

2. `lh` (*load halfword*), `lhu` (*load halfword unsigned*), `sh` (*store halfword*): operate on two consecutive bytes. In this case only byte couples [0, 1], [2, 3], [4, 5] or [6, 7] of a memory doubleword can be accessed. This means that the MSB of the virtual address should be 0.

3. `lw` (*load word*), `lwu` (*load word unsigned*), `sh` (*store word*): operate either on the first or the second word of the memory halfword, that its the byte sets [0 to 3] or [4 to 7]. This means that the *two* LSBs of the produced virtual address must be 0.

4. `ld` (*load doubleword*), `sd` (*store doubleword*): operate on all the memory doubleword. This means that the *three* LSB of the virtual address should be 0.

If some of the previous constraints is not met by the produced virtual address, the *virtual address adder* communicates to the source buffer that an *address misaligned* exception occurred. The source buffer will translate this into a *load address misaligned* or *store address misaligned*. Notice that this kind of exception has a *higher priority* than the previous. So if both the exceptions are detected, only the *address misaligned* is communicated to the *load buffer* or the *store buffer*. Regarding unaligned memory access (e.g. read bytes [2 to 5] with a `lw`) is not supported in hardware in this first implementation of *LEN5*. This is usually a feature that is required to support legacy or application-specific code, and it's hardly ever used in modern computer applications. Still, a possible solution is proposed in section 5.3.3.

3.7.2 Load buffer

The load buffer is the component of the *load-store unit* that prepares *load* instruction for the *data cache* access. It acts similarly to a *RS*, despite the much more complex sequence of operations that is required to execute *load* instructions. The data structure is similar to the one of arithmetic *RSs*, with additional data fields for the virtual and physical address and additional control bits to keep track of the dependences on *store* instructions. No detailed description of these fields is provided, since their purpose should be clear by reading the next section dedicated to the *load buffer* control and there is no way the entire data structure can fit in a page of this document, so refer to table 3.4. For this reason, it is suggested to refer to the *LEN5* code when reading this section. Still, a list of a *load buffer* entry fields is provided with a brief description for future reference:

- `valid`: the entry contains a valid instruction.
- `busy`: the entry is waiting for an operation to complete.
- `store_dep`: the *load* must wait for all previous *store* instructions to commit before it can be executed.
- `pfwd_attempted`: the entry is ready for the cache access.
- `[entry_age]`: the optional age of the entry, used when an *age based selector* in employed.
- `older_stores`: the number of older uncommitted *store* instructions. If 0, the entry is dependency-free.
- `load_type`: one among `lb`, `lbu`, `sb`, `lh`, `lhu`, `sh`, `lw`, `lwu`, `sh`, `ld`, `sd`.
- `rs1_ready`: the value of `rs1` contained in `rs1_value` field is valid.
- `rs1_idx`: the index of the *ROB* that will contain the base address.

- `rs1_value`: the value of the base address.
- `imm_value`: the value of the immediate field (*offset*).
- `vaddr_ready`: the virtual address has already been computed.
- `vaddr`: the virtual address.
- `paddr_ready`: the address translation (TLB access) has already completed.
- `ppn`: the physical page number. The entire physical address is obtained appending the twelve LSBs of the virtual address, that are never translated.
- `dest_idx`: the entry of the *ROB* where the loaded value will be stored.
- `except_raised`: an exception occurred.
- `except_code`: the exception code.
- `ld_value`: the value loaded from memory.
- `completed`: the value has been fetched from the *data cache*.

The control logic of the *load buffer* is similarly organized as in an arithmetic *RS*. In general, *load* instructions can be executed OoO, but only if the accessed memory location is updated.

The block diagram of the *load buffer* would be too large and complex to be appreciated on a page of this document. In general, the internal organization is quite similar to the one of an arithmetic *RS*, which is shown is fig. 3.8. Two additional selectors are required to perform all the three main operations required to execute a *load* instruction: virtual address computation, address translation (in the *DTLB*) and finally the *data cache* access. The control logic must perform additional checks for memory hazards, as described later in this section, and it must handle the forwarding outcome from the *store buffer*.

A few words must be spent to introduce the store-to-load forwarding mechanism. If a data dependency between a *load* and a previous *store* is detected while the *store* has not been executed yet, the value to be stored can be forwarded to the *load*. Notice that from the RISC-V specification [16], if a certain memory location has the write access permissions it also has read access ones. The *writeable not readable* condition is reserved. So a value can *always* be forwarded from *stores* to *loads*, while the opposite is not true, since a *store* can sill raise some exceptions that a previous *load* didn't[16]. Even if forwarding of exceptions would be possible thanks to the *ROB*, there is absolutely no point in implementing it since the *store* exception is

[16]More details in [10].

processed first at commit. So forwarding is only performed from *stores* that have no registered exceptions. All subsequent instructions will be flushed unconditionally. Both the virtual and the physical addresses can be used to verify the dependency, with the latter being able to forward also on synonyms. In *LEN5*, two store-to-load forwarding attempts are performed. The execution of a *load* instruction can be split into three main steps: virtual address computation, *DTLB* access (only if VM is enabled) and *data cache* access. The virtual forwarding is attempted after the *virtual address adder* has produced the virtual address, while the physical forwarding is attempted after the *DTLB* access has completed.

Another important aspect of the store-to-load forwarding is that it can be completed correctly only if the *load* accesses bytes that the *store* is actually writing. The value to be stored of a partial *store* instruction (`sb`, `sh` and `sw`) is fetched from a register and so it doesn't contain the rest of the memory doubleword. This means that the forwarding can take place only if the *load* acts on a portion of that doubleword that the *store* is actually writing. This check is performed by the forwarding logic in the *store buffer* as described in section 3.7.4.

While loading data from the cache, the entire destination doubleword is received by the *load buffer* regardless of the type of *load*, as if the three LSBs of the *load* address (*byte offset*) were zero. For this reason the *load buffer* contains a simple *byte selector* whose schematic is reported in fig. 3.10. Its purpose is to extract the correct set of bytes from the entire doubleword and sign-extend it or zero-pad it based on the type and the byte offset of the *load*.

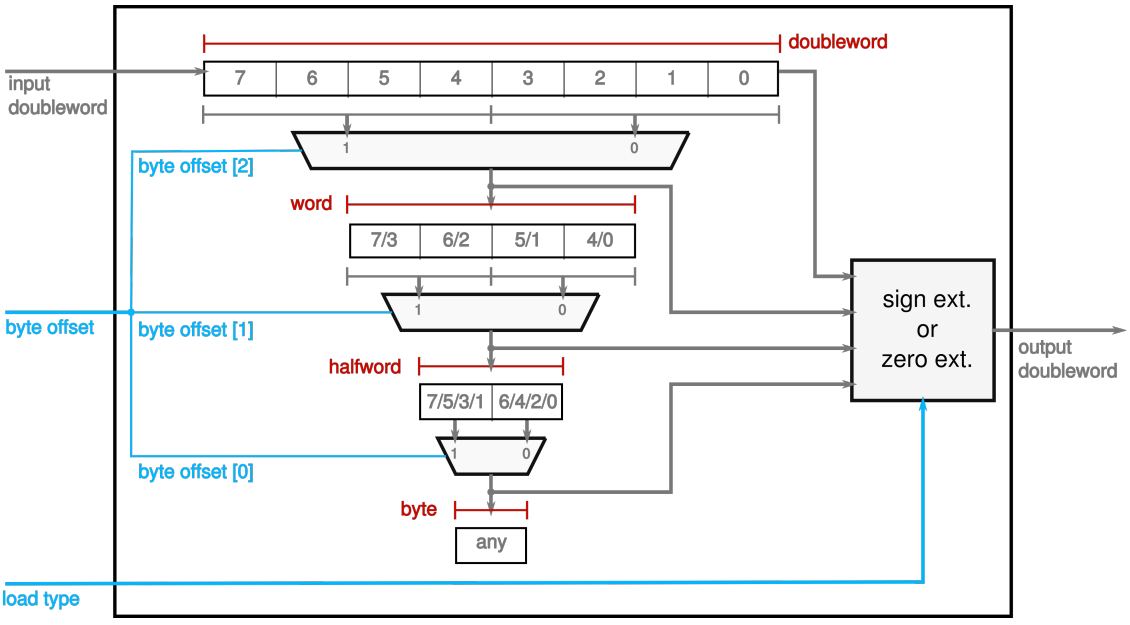


Figure 3.10: Block diagram of a *byte selector*.

In the following paragraphs, each step of the execution of a *load* is detailed,

while the *forwarding* mechanism and hazard check are discussed.

Insert a new instruction This operation is similar to the one described for an arithmetic *RS*. A priority encoder is used to select an empty entry (if any) in the *load buffer* and the *valid* signal from the *issue logic* is used to trigger the insertion. The **first write port** of the *load buffer* is used. The only addition is that the number of in-flight stores in the *store buffer* is saved in the dedicated `older_stores` field of the entry assigned to the new *load*. This information is used during forwarding and before the *data cache* access to know if there are unresolved dependences with some previous *store* instruction in the *store buffer*. This quantity is decremented whenever a *store* is committed (i.e. popped from the *ROB* and marked as not valid in the *store buffer*[17]) until it reaches zero. When there are no previous uncommitted *store* instructions, the *load* can be executed without any further check, because all its dependences on previous *stores* have been solved: the value has already been written in the *data cache* and can be accessed by the *load*. Still, forwarding from the *store buffer* is still possible in this case.

Operand fetch Before the virtual address can be computed, the `rs1` operand (the base address of the *load* instruction) must be available in the *RF* entry where the *load* instruction is allocated. If it wasn't ready in the *RF* or in the *ROB* during issue, it will be fetched from the *CDB* as soon as some other *RS* produces it using the CAM-like **first parallel access port** described in section 3.6, like in any other *RS*. Once it is available, the `rs1_ready` status bit is asserted so the instruction can proceed to the virtual address computation.

Virtual address computation request An entry of the *load buffer* can be selected for the virtual address computation if:

- It contains a `valid` instruction.
- The contained instruction has been selected yes (i.e. it is not `busy`).
- The value of the operand is ready (`rs1_ready`).
- The virtual address of the instruction has not been computed yet.
- The instruction is not completed[18].

[17]The *store* is not removed from the *store buffer* to implement the *cache level zero* mechanism suggested in [1] and described in section 3.7.3.

[18]This check is redundant with the previous, but since the control logic doesn't use explicit states, it doesn't hurt making things a bit more fault-proof.

A first *selector* chooses one entry among the ones meeting all the previous requirements, and a request is sent to the *virtual address adder* arbiter by asserting the dedicated output *valid*. The selector index is used to drive the **first read port** of the *load buffer*, and the operands are sent to the *virtual address adder* (`rs1` and `imm`). The selected instruction is marked as *busy* so it won't be selected in the next cycle. Remember that the *virtual address adder* is not a combinational component. The *valid-ready* handshake protocol, together with the `valid` field in the *load buffer* entries makes it possible to use the *virtual address adder* as any other *functional unit* connected to a *RS*. In fact, also the index of the source entry of the *load buffer* is sent to the *virtual address adder* together with the base address (`rs1_value`) and the offset (`imm_value`). This means the latency and/or the number of pipeline stages of this component can be changed arbitrarily and the *load buffer* will continue to work as expected.

Virtual address adder answer When the *virtual address adder* has completed the virtual address computation and its compliance with the current VM mode, it asserts its output *valid* signal and produces both the virtual address and the index of the source entry of the *load buffer* at its output. Notice that the *ready* signal from the *load buffer* is always asserted, as it happens in the other *RSs*. The reason is that the entry where the virtual address must be saved has already been allocated during issue, so the *load buffer* is always ready to accept answers. This is true for *DTLB* and *data cache* answers too. When the answer is received, the *load buffer* checks if an exception was raised during the virtual address computation. If so, the *address misaligned* or *page fault* exception is translated in an ***load*** *address misaligned* or ***store*** *address misaligned* and the corresponding exception code is stored in the entry. Besides this, the virtual address is copied in the `ld_value` field of the *load buffer* so it will be written in the *result* field of the *ROB* and used during exception handling. The instruction is marked as `completed`, so in the following cycles, it can only be selected to write its result (i.e. the offending virtual address) on the *CDB* together with the exception code. Doing so the faulting instruction is prevented from attempting store-to-load forwarding and from accessing both the *DTLB* and the *data cache*, so no data is introduced in the execution pipeline. If no exception is detected, the virtual address is simply copied in the dedicated `vaddr` field and the `vaddr_ready` status bit is set, meaning that the instruction can proceed to the virtual forwarding attempt. The **second write port** is used to do this. In this case, the `busy` bit is **not** cleared and the incoming index from the *virtual address adder* is stored in a register. This same value is immediately used to perform the virtual forwarding without the need for an additional selector. The only exception is when VM is not enabled. If so, the 56 MSBs of the virtual address are also copied in the `ppn` field of the *load buffer* entries, the associated `paddr_ready` status bit is set and the `busy` bit is cleared. Doing so the instruction will skip the *DTLB* access and proceed directly to the *data cache* access.

Virtual address forwarding As mentioned in the previous paragraph, the store-to-load forwarding using the virtual address is performed in the next cycle after the virtual address is received (if VM is enabled). The index stored in the dedicated register is used to drive the **second read port** so the virtual address of the selected instruction is sent to the *store buffer* together with the number of the currently uncommitted *store* instructions that are available in the `older_stores` field of the same entry. The virtual forwarding can resolve in one of the following situations:

- The twelve LSBs of the virtual address of the *load* are different from the corresponding bits of all the *previous* uncommitted *store* instructions in the *store buffer*[19], the *load* is certainly independent on any of those instructions because those bits are never translated. When this happens the *load* is marked as *dependency-free* (i.e. there are no RAW hazards) setting the number of previous uncommitted *stores* (`older_stores`) to zero. The instruction will skip the physical address forwarding attempt and be inserted directly to the ones suitable for the *data cache* access.

- At least one of the previous *stores* doesn't have its virtual address or the value to be stored ready when the virtual forwarding is attempted or all the data is available but there's no match with any of the virtual addresses of the previous *stores*. It is important to point out that having different virtual addresses (apart from the twelve LSBs) **doesn't imply that the store and the load are independent**: they can still be synonyms. This case is included in the following one. In this case, the forwarding attempt simply fails. The *load* is marked as *store-dependent* setting the dedicated `store_dep` status bit, so it will be chosen for physical forwarding after the *DTLB* access. A second solution could be to speculate that the *load* doesn't depend on the *store*, and proceed to the *data cache* access as in the previous case. However, this would require a mechanism to detect the dependency and replay the *load* if necessary.

- If all the previous *stores* have their data and virtual address ready, the virtual forwarding can *hit*, meaning that at least one *store* in the *store buffer* has the same virtual address as the *load*. On a hit, the value of the **most recent** matching *store* (but still older than the *load*) is copied in the `ld_value` of the *load*, and the instruction is marked as `completed`. Since the data to be stored comes from a register and it's therefore aligned to the LSB, there is no need to select the correct byte set according to the *load* byte offset. No *DTLB* or *data cache* accesses are performed, and the *load* can write its result to the *CDB* and to the *ROB* as soon as it is selected. A **third write port** is used for this purpose. Remember that not all the ports operate on the same field, so the high number of ports doesn't necessary imply a slower structure.

[19]The details on this check are provided in the dedicated paragraph in section 3.7.3.

In all the previous cases the `busy` bit of the instruction is cleared so it can proceed to the next phases.

Notice that the forwarding is also possible from the *stores* that have already committed. In fact, they are not popped from the *store buffer*, but only marked as completed. This implements the *cache level zero* suggested in [1]. Of course, only the most recent value that's going to be written in a certain memory location must be forwarded to a *load*, so in these terms, cached *stores* have lower priority than in-flight instructions. Again, more information on this selection process can be found in the dedicated paragraph in section 3.7.3.

Virtual forwarding is the one with potentially the higher advantage in terms of latency because both the *DTLB* and the *data cache* accesses are skipped. However, it is also the one that shows the lower hit ratio: since it is executed earlier the probability that the matching *store* has already both its virtual address and value ready is lower. This is one of the main reasons why *stores* are by default granted the highest priority in accessing the shared unit (the *virtual address adder*, the *DTLB*, and the *data cache*) using the priority arbiters instead of the fair arbiters from section 3.1.3.

Address translation request If no exception is raised in the previous steps and the virtual forwarding attempt has been completed (i.e. the `busy` bit is clear), the entry can be selected for the address translation in the *DTLB*. Another selector performs this choice. Its index is used to drive a **third read port** to send the Virtual Page Number (VPN) of the selected instruction to the *TLB*. This is the portion of the virtual address that identifies the virtual page in the page table of the current process. In other words, the VPN is given by:

$$VPN = vaddr[vaddr.len - 1 : poffset.len];$$

Where *vaddr* is the virtual address as produced by the *virtual address adder*, *vaddr.len* is its width according to the Sv39 or Sv48 paging scheme and $poffset.len$ is the width of the page offset, that in the case of 4 Kibit pages is twelve bits. An instruction can be selected for the *DTLB* access if:

- It contains a `valid` instruction.
- The instruction has not been selected yet (i.e. it is not `busy`).
- It has it's virtual address ready (`vaddr_ready` set).
- It doesn't have its physical address ready (`paddr_ready` not set).
- It doesn't have registered exceptions (`except_raised` not set).
- It is not `completed`.

If the *DTLB* can accept the request (i.e. it assets is *ready*), then the instruction is marked as `busy`.

DTLB answer The *DTLB* acts as any other *functional unit* taking the index of the source entry of the *load buffer* and returning it when it answers a request. It is possible that the *DTLB* answers the request from the *load buffer* OoO, if it handles hit under miss. Based on the fact that the requested virtual address is in the *DTLB* (*hit*) or not (*miss*), the *load buffer* can receive two different types of answer, both with a dedicated *valid* signal.

1. A proper *answer* regarding the received request.

2. A `wake-up`.

The first type of answer is used as a consequence of a *hit* in the *DTLB*. In this case, the index returned by the *DTLB* is used to drive the **fourth write port** of the *load buffer*, which is used to write the translated physical address and possibly any raised exception in the source entry. The handling of the exception is the same used for the *virtual address adder*, except that here the possible exceptions are the *load page fault* exception and the *load access fault* exception. The second one is used only if advanced virtual memory protection techniques are used. The memory system of *LEN5* does not implement these advanced functionalities, so the *DTLB* can only raise a *page fault* exception. However, the execution pipeline supports both. If an exception is received, the exception code is saved, the offending virtual address is saved in the `ld_value` field and the instruction is marked as complete as usual, so the *data cache* access is skipped. If no exception is raised, the `busy` bit is cleared, the Physical Page Number (PPN) from the *DTLB* is saved in the dedicated `ppn` field of the source entry and the `ppn_ready` bit is set, so the *load* can proceed to the *data cache* access.

The *wake-up* answer is used when the *DTLB* request resulted in a *miss*. Notice that the miss is not explicitly notified by the *DTLB*: the *load* simply remain *busy* until the miss has resolved. When a miss occurs, the *DTLB* access the *level 2 TLB* and the above levels to fetch the requested *leaf* of the page table. When it successfully obtains the requested PPN, it forwards it to *all the load buffer* together with the corresponding VPN. Each valid entry of the *load buffer* that hasn't performed the *DTLB* access and whose VPN matches the one returned by the *DTLB* saves it into its entry. Although this requires a **second parallel access port**, it speeds up the execution forwarding the translation to all the matching *loads*, whose `ppn_ready` status bit is set. These will skip the *DTLB* access and proceeds directly to the *data cache* access. Also, exceptions can be forwarded in the same way. Again, remember that only the exception raised by the first instruction in program order will be processed during commit, thanks to the *ROB*.

Physical address forwarding Because of the possibility to resolve multiple address translations at the same time thanks to the *wake-up* mechanism, the *physical* forwarding cannot be handled the same way the *virtual* address forwarding was.

Here, another *selector* driving a dedicated **fourth read port** is used instead. The physical address of the selected entry is sent to the *store buffer* that performs all the required checks, as it does for the virtual forwarding. In a similar way, the physical forwarding can result in the following situations:

1. All the previous *store* instructions in the *store buffer* have their physical address and their value ready, but no match among those physical address and the one of the load is found. In this case the load can be definitely marked as *dependency-free* setting its previous uncommitted *stores* counter (`older_stores`) to zero. The instruction can automatically be inserted among the ones suitable for the *data cache* access.

2. At least one of the previous *store* instructions in the *load buffer* doesn't have its physical address or the value to be stored ready. In this case, the forwarding fails and the instruction is marked as *store-dependent* setting its `store-dep` status bit. A store-dependent *load* can perform the *data cache* access only when its `older_stores` counter reaches zero. When this happens, the *load* is automatically inserted among the ones that can be selected for the *data cache* access. This is the worst scenario for what concerns the latency of a *store*: both the *DTLB* and the *data cache* accesses must be performed to complete its execution. However, if the cache replacing policy is accurate enough, only the *data cache* latency must be paid most of the time.

3. All the older *stores* in the *store buffer* have their physical address and value ready, and at least one of them matches the physical address of the selected *load*. In this case, the physical forwarding *hits*, and the value or the exception from the **most recent** matching *store* (but still older than the *load*) can be copied into the *load* entry in the *load buffer*. A **fifth write port** is used for this purpose. The *load* is marked as *completed*, meaning that it is inserted among the ones ready to write their result (or exception) to the *ROB* using the *CDB*.

Regardless of the outcome of the forwarding, the `pfwd_attempted` status bit of the selected instruction is set, so it is not selected again in the next cycles, and can be inserted among the ones suitable for the *data cache* access when its older *stores* counter reaches zero. Notice that a virtual or physical forwarding attempt is not replayed when it fails: each instruction can perform at most one virtual forwarding attempt and one physical forwarding attempt. In theory, when an attempt fails in the second situation above the forwarding could be replayed as soon as a *store* gets its value or its virtual or physical address. However, this would require many additional parallel access ports, so it wasn't implemented in the *load-store unit*. Consider that two forwarding attempts, with the help of level zero caching, should be more than enough to reduce the *DTLB* and *data cache* accesses.

The physical address forwarding has a smaller advantage in term of latency than the virtual address forwarding but also shows the higher hit ratio.

Data cache access request If a *load* has no registered exceptions, has gone through both the forwarding attempts without success, and if it was proven to be *dependency-free* from all the previous *stores* instructions in the *load buffer* (i.e. its `older_stores` counter is zero), it can proceed to the *data cache* access. A *selector* chooses a suitable instruction and drives the **fifth read port** to feed the *data cache* with the instruction physical address. An entry can be selected for the *data cache* access if:

- It is `valid`.
- It is not `busy`.
- It is not `completed`.
- It has its physical address ready (`ppn_ready` set).
- It has not registered exceptions (`except_raised` clear).
- It has attempted the physical address forwarding (`pfwd_attempted` set).
- It is not *store-dependent* or there are no older uncommitted store in the *store buffer*.

If the *data cache* can accept the request (i.e. its *ready* is asserted), the `busy` status bit of the selected instruction is set, so the entry is not selected for the *data cache* access in the next cycles. Remember that the memory system of *LEN5* can handle hit-under-miss and even miss-under-miss, so the *data cache* can answer request in a different order from the one they are issued with (that is already OoO with respect to the program order). So the *data cache*, as the *DTLB* and every other *functional unit*, receives also the index of the source entry of the *load buffer* when a request is issued.

Data cache answer The *data cache* uses the same types of answer that are used by the *DTLB*, therefore the process is exactly the same, except that no exception can be raised during the *data cache* access (all checks are made during the address translation in the *DTLB*) and when a *wake-up* answer is received, the loaded value is copied instead of the PPN. Once an answer from the *data cache* is received, the `busy` status bit of the source instruction is cleared and it is marked as `completed`, meaning that it can be selected to write the loaded value in the *ROB* using the *CDB*. The *data cache* answers use the **fifth write port** and the **third parallel access port**. Again, the fact that the *load buffer* uses many ports is not significant of its performance, because most of these ports operate on different fields of the *load buffer* data structure as if they operated in totally different register files.

CDB request Instructions that are marked as `completed` can be selected to write on the *CDB*. Remind that the execution of a *load* is completed when it has performed all the previous steps or as soon as an exception has occurred. One last *selector* is used to drive the **sixth read port** to send the loaded value or the exception code of the selected instruction on the *CDB*. If the *CDB* is currently assigned to the *load buffer* (i.e. the associated *ready* from the *CDB* arbiter is asserted), the selected *load* instruction is finally popped from the *load buffer*: its `valid` status bit is cleared and the entry becomes available to host a new issued instruction from the next cycle.

3.7.3 Store buffer

The *store buffer* is the hosting *store* instructions while they perform all the steps of their execution. It differs from the *load buffer* in that the *data cache* access is performed in program order. This is mandatory to handle WAW hazards among *store* instructions. As a consequence, instructions inside the *store buffer* are kept in-order. A head and a tail counters are instantiated in the *store buffer* as they were in the *branch unit RS* or in the *issue queue*. Apart from the *data cache* access, all the other steps in the execution of a *store* are performed OoO. The execution of a *store* is very similar to the execution of a *load*, since both have to go through the same main steps: the virtual address computation in the *virtual address adder*, the *DTLB* access for the address translation[20] and the *data cache* access.

However, there are some important differences that must be highlighted.

Once again, the data structure of the *store buffer* is too complex to fit in a page of this document, so an example table is omitted. For what concerns the field of each entry of the *store buffer*, they are almost the same of the *load buffer*, despite the fact that *store* instructions need also the second operand `rs2` that contains the value to be stored in memory, and don't have a destination register since they don't produce any result.

The detailed schematic of the *store buffer* would definitely be too large for this document, and it's therefore omitted. Overall, it is quite similar to one of an arithmetic *RS* shown in fig. 3.8. The main differences are the presence of the head and tail counters and of the forwarding logic described in section 3.7.4. Two additional selectors are required for the execution of a *store*, since three different operations must be accomplished: virtual address computation, address translation (in the *DTLB*) and finally the *data cache* access. A high level representation of the *load buffer* is given in fig. 3.9.

The following paragraphs briefly explain what is done in each step of a *store* execution, focusing mostly on the differences from the *load buffer*.

[20] If VM is enabled in the `satp` CSR

Push a new instruction As mentioned above, a tail counter is used instead of a *selector* to point to the entry where a new *store* instruction from the *issue logic* can be allocated. If the pointed entry is still valid, the output *ready* signal to the *issue logic* is not asserted, so the issue of a new *load* is prevented even if the tail entry of the *ROB* is free. As it will be clear in a moment, a *store* is completed when it has successfully completed its *data cache* access. When this happens, the `valid` is not cleared to implement the *cache level zero* technique.

Operands fetch This phase differs from the same one in the *load buffer* only because here also the value of `rs2` is copied from the *CDB* if it wasn't available during issue.

Virtual address computation request A request to the *virtual address adder* is performed OoO as soon as the base address (`rs1`) of a *store* is ready. It is exactly the same as in the *load buffer*, so refer to section 3.7.2 for the details.

Virtual address adder answer An answer from the *virtual address adder* is processed as it is in the *load buffer*, without any difference. Of course, *page fault* and `address misaligned` exceptions are translated in ***store*** *page fault* and ***store*** *address misaligned* exceptions. Since there is no *result* field in the *store buffer*, the faulting virtual address is copied in the `rs2_value` filed. Notice that if an exception occurs, the value to be stored is not sent to the memory, so overwriting it is totally fine. The virtual address will be sent on the *CDB* and saved into the *result* field of the *ROB* as usual. Notice that in the *store buffer* only faulting *store* instructions actually use the *CDB*.

Address translation request Also, *DTLB* requests are issued in the same way they are in the *load buffer*. Remember that the virtual address forwarding is only allowed from *stores* to *loads*. A description of the logic handling both the virtual and the physical forwarding is in the last paragraph of this section.

DTLB answer A proper *answer* from the *DTLB* is processed by the *store buffer* exactly as the *load buffer* did. In fact, the *DTLB* distinguishes between answers for the *load buffer* and for the *store buffer*. A decoder in the top-level module of the *load-store unit* asserts only the correct *valid*[21]. However, a *wake-up* is sent to the *load-store unit* without distinguishing between the *load buffer* and the *store buffer*. As soon as the *DTLB* receives the PPN from the higher levels of the hierarchy, it forwards it to the whole *load-store unit*. This means that a *DTLB wake-up* can be

[21]In fig. 3.9 the *DTLB* and *data cache* decoders are not shown due to the limited space. They can be considered part of the *DTLB* and *data cache* arbiters.

a consequence of wither a *load* miss or a *store* miss. Since additional exceptions can be raised during the *DTLB* access of a *store*, the forwarded PPN cannot be saved in the *store buffer* source entry. So when the *store buffer* receives a *wake-up* from the *DTLB*, all its entries that match the returned VPN are simply moved from their *busy* (i.e. the `busy` status bit is cleared) state to those suitable for the *DTLB* access. In other words, they are woken up and replayed. The next time the request is issued it will almost certainly hit because the target line of the page table has been obtained as a consequence of the *DTLB* miss, so a proper *answer* will be received by the *store buffer*. For more information about this process please refer to [10]. Of course, in case an exception has been raised during the address translation, the received *page fault* and `access fault` exceptions are translated into ***store*** *page fault* and ***store*** *access fault* exceptions.

Data cache access request The main difference between *loads* and *stores* is that the latter must perform the *data cache* access in order. So instead of using a *selector*, a head pointer is used for the execution of *branch* instruction in the *branch unit RS*. Only the physical address of the head entry of the *store buffer* can be sent to the *data cache*. Besides, the *data cache* request can be issued only if the *store* has also reached the head fo the *ROB*, meaning that all previous instructions have been executed. This check is performed by the *load buffer* by comparing the target *ROB* index of its head instruction to the current head index of the *ROB*. If they match, the *data cache* request can be issued. Notice that unlike in the *load* buffer, it is not possible to issue more than one *store data cache* requests, since this would violate the in-order execution of store instruction: until confirmation of the completion of the execution of a *store* by the *data cache* is received, no other *stores* can perform the cache access. This is avoided by the head pointer, which is not incremented until the current *store* is completed.

Data cache answer The *data cache* write request of a *store* can result in a *hit* or a *miss*. If the *store* misses, the *DTLB* answers (after an arbitrary number of cycles) with a *wake-up*. Notice that unlike the *load* case, here no useful information is carried by this answer. It is only used to clear the `busy` bit of the head entry of the *store buffer* (and no others), so the *data cache* request can be replayed. As for the *DTLB* case, this time the request will probably hit because the requested line has been fetched from the higher levels of the memory hierarchy. If the *store* hits, the *data cache* sends an answer containing the index of the source entry of the *store buffer*, as it does for *load* instructions. Here the answer doesn't contain any useful data, and it's only used to communicate the execution completion of the *store* to the *store buffer*. When this happens, a signal is sent to the *ROB* so the *store* can be popped from its head entry. Notice that the *CDB* is not used, because all the information required for the commit of the *store* are taken directly from the *store buffer*. This simplifies the entire execution pipeline since the only component

that communicated with the memory system is the *load-store unit*. When the *store* is committed and popped from the *ROB*, it is not completely removed from the *store buffer*, even if the head counter is incremented. Instead, while its `valid` bit is cleared, it is also marked as completed. In fact, the entry data is still used during the store-to-load forwarding process, implementing what in [1] is referred to as *cache level zero*. A description of this quite simple trick is reported in section 3.7.5.

CDB request As mentioned before, *stores* do not modify any register. Therefore they don't need to write anything to the *ROB*. The only exception is when an exception has been raised during the virtual address computation or the address translation in the *DTLB*. Remember that no exception can arise from a *data cache* access. As soon as an exception is detected, its `except_raised` status bit is set and the instruction is marked as `completed` so it is not chosen for any other operation apart from writing the exception code and the faulting address in the *ROB* using the *CDB*. Notice that instruction marked as completed might also be used for forwarding. Even if this might happens, it has no consequences on the execution flow, since the *load* that might have received a totally meaningless value from a faulting *store* will be flushed unconditionally when the *store* commits and its exception is processed. As usual, the offending virtual address is saved in the *result* field of the *ROB*[22] so it can be used during commit.

3.7.4 Store-to-load forwarding

The logic that performs all the checks for the forwarding on the virtual or physical address is instantiated in the *store buffer*. In fact, the incoming information from the *load buffer* must be compared to all the *store buffer* entries **in parallel**. The possible outcomes of a forwarding attempt have been already discussed in section 3.7.2. Now, the implementation of the required checks is analyzed.

The value that a *store* instruction is going to write to the memory can possibly be forwarded to a *load* only if the *store* is *older* than the *load* to void RAW hazards. But this condition is not enough. The *store* value is taken from must be the **last** in-flight instruction among the *stores* older than the *load* that modifies the memory location accessed by the *load*.

Since the program order is lost right after the issue stage, a method to reconstruct the relative order between *loads* and *stores* from known information is required. In the *LEN5 load-store unit*, this is done by tracking the number of the uncommitted *stores* for each *load*, using the dedicated counters described in section 3.7.2. When a *load* is inserted in the *load buffer*, its `older_stores` counter is initialized to the number of in-flight *stores*, that is the number of valid stores in the *store buffer*.

[22] That is otherwise unused in *ROB* entries assigned to *store* instructions.

This quantity can be simply obtained as the difference between the head and tail pointers of the *store buffer*. An additional bit (the MSB) is used to distinguish the *empty* and *full* conditions of the *store buffer*:

- When the *store buffer* is empty the difference between the head and the tail counters is zero, and the MSB is set to zero too.
- When the *store buffer* is full the difference between the head and tail counters is again zero, bu the MSB is set to one.

The MSB of the in-flight store count is simply the reduction *and* of the `valid` status bits of all the *store buffer* entries. The resulting number is exactly the quantity of in-flight stores in the *store buffer*.

Each time a new *store* is committed, each counter in the *load buffer* is decremented to track the remaining previous uncommitted *stores* for each *load* instruction. When the virtual or the physical forwarding is attempted, this value is sent to the *store buffer* together with the *load* virtual or physical address.

The number of uncommitted *stores* on which the *load* might depend must now be translated into a set of indexes of the *store buffer*. This set must comprise already committed *stores*, that are surely older the *load* (implementing cache level zero, see section 3.7.5), plus a number of uncommitted *stores* equal to the value of the uncommitted store counter associated to the *load*. The oldest *store* in the *store buffer* is found in the entry pointed by its tail pointer, which contains the next *store* that will be replaced by a new one. On the other hand, the head pointer of the *store* buffer points to the next *store* that will commit, and so the oldest of the uncommitted *stores* indicated by the counter of the *load*. In other words, the range of possible *stores* to forward the value from starts from the current value of the tail pointer of the *store buffer* and goes up to the current head pointer, plus the number of uncommitted *stores*, that is the entry if the *store buffer* that the *load* would have occupied if it was inserted in-order with respect to the other *store* instructions. Another way of seeing this is defining the *stores* as the unit of measure of the *age* of a *load* (i.e. the *years*), and thinking about the number of uncommitted stores from its counter as the *negative age* of the *load* with respect to the *store* allocated in the current head entry of the *store buffer*: the *load* is *younger* than that *store* by a number of *store* instructions (*years*) equal to the value of its associated uncommitted-stores counter. In the coordinate system of the *store buffer*, the *load* age is given by the sum of the current *store buffer* head pointer and the *age* of the *load*.

$$ld.idx = ld.cnt + SB.head \tag{3.1}$$

Where $ld.idx$ is the index that the *load* would have occupied if it was issued in the *store buffer* in-order with respect to *stores*. That is its *age* in the coordinate system of the *store buffer*. $ld.cnt$ is the number of older uncommitted *stores* from

the (`older_stores` counter) of the *load buffer* entry where the *load* is allocated, that is its *negative age* with respect to the current head instruction of the *store buffer* $SB.head$.

To avoid overflow issues, all these indexed must be shifted so the oldest *store* instruction has index equal to zero. This means performing a coordinate shifting into a system where all *ages* are positive, since every index is referred to the oldest one. Since the oldest instruction is found in the entry pointed by the value of the tail index of the *store buffer*, all quantities must be translated by that amount:

$$\begin{cases} SB.tail_T = SB.tail - SB.tail = 0 \\ SB.head_T = SB.head - SB.tail \\ ld.idx_T = ld.idx - SB.tail \end{cases} \tag{3.2}$$

To better understanding what is happening, take a look at the example in table 3.5.

index	**entry #**	**valid**	**contained *store***	**suitable**
	0	no	committed	yes
$SB.head \rightarrow$	1	yes	uncommitted	yes
	2	yes	uncommitted	yes
$ld.idx \rightarrow$	3	yes	uncommitted	no
	4	yes	**youngest**	no
	5	no	-	no
$SB.tail \rightarrow$	6	no	**oldest**	yes
	7	no	committed	yes

(a) The original organization of the *store buffer*.

index	**original #**	**entry #**	**valid**	**contained *store***	**suitable**
$SB.tail_T \rightarrow$	6	0	no	**oldest**	yes
	7	1	no	committed	yes
	0	2	yes	committed	yes
$SB.head_T \rightarrow$	1	3	yes	uncommitted	yes
	2	4	yes	uncommitted	yes
$ld.idx_T \rightarrow$	3	5	yes	uncommitted	no
	4	6	yes	**youngest**	no
	5	7	no	-	no

(b) The organization of the *store buffer* shifted by the value of the tail pointer of the *load buffer*.

Table 3.5: Forwarding example. Table 3.5a shows the original position of the entries of the *store buffer*, while table 3.5b shows the same structure after the translation. In this case all the indexes are shifted by 6. Entries containing *store* not suitable for the forwarding are greyed-out.

From the example it is clear how the *store* instruction suitable for forwarding

are the ones whose index falls in the range:

$$0 \leq idx_T < ld.idx_T = SB.tail_T + SB.head_T + ld.cnt \tag{3.3}$$

By applying the inverse transformation by applying the definitions in eq. (3.2), the following expression is obtained in the original coordinates:

$$0 \leq idx - SB.tail < SB.head + ld.cnt - SB.tail \tag{3.4}$$

Among these entries, only the one with the highest shifted index (i.e. the youngest) that matches the address of the *load* can be selected to forward its value.

However, additional checks are needed:

- **All** the *stores* in the selected range must have both their address (virtual or physical) *and* their value to be stored available. Otherwise, the dependency cannot be verified: the *load* could depend on another store whose address and value are not available yet. In this case, if the forwarding is performed, an error is introduced.

- The selected *store* must operate at least on the bytes that the *load* wants to read. This means that a `lw` cannot be forwarded with the data from a `sh`, as a `lb` cannot be forwarded with the data from a `sh` that operates on another halfword of the memory doubleword. The data to store does not include the rest of the memory doubleword when it is found in the *store buffer*. It is just a value from a register aligned to the LSB of the doubleword. It is the *data cache* that overwrites the correct portion of the entire destination doubleword during the write operation, without modifying the rest of it[23].

As pointed out at the beginning of section 3.7.2, store-to-load forwarding doesn't require any correction based on the byte offset of the *load*: the data that is going to be stored comes from a register (`rs2`) and it is aligned at the doubleword LSB. It is the *data cache* that overwrites only the destination doubleword with the value from the register according to the *store* byte offset. So the value received by the *load* after forwarding is already correctly aligned and ready to be stored in the *load* destination register `rd`.

3.7.5 Cache level zero

This very simple trick suggested in [1] is based on the observation that a completed store instruction can be kept in the *store buffer* and accessed during forwarding, exactly as if it was located in a cache line. Since *LEN5* implements a single-core

[23]The entire doubleword is first read from the *data cache*, and then written again with the correct set of bytes overwritten with the new value.

processor, with only one thread executing at a time, it doesn't show the coherency issues of this idea, since the value in the cache cannot be modified by any other *load-store unit*. Therefore if another *store* modifies the same memory location, it will be chosen during forwarding because it will be younger than the first one, as explained in section 3.7.4. Accessing already committed *store* in the *store buffer* instead of the *data cache* saves potentially slow cache accesses. Of course, as soon as a new *store* is issued, it overwrites the oldest already committed store in the *store buffer*, so the space that is available for caching in the *store buffer* is limited, although depending on the depth of the *store buffer*. When memory-intensive code is executed, the *store buffer* is very likely to be full most of the time, leaving no space for caching. However, in most cases, the advantage can be significant, as shown by the result in the original article.

3.8 Common Data Bus

As explained in section 2.3.2, the *CDB* is the *shared* communication link between all the *RSs*, including the *load-store unit*, and the *ROB*. In this implementation of *LEN5*, it carries four data fields:

- `rob_idx`: the index of the destination *ROB* entry where the carried value must be stored. This corresponds to the *ROB* entry that was assigned to the instruction producing that result during issue, which is the tail entry of the *ROB*. This data is used to drive one of the *ROB* write ports, as described in section 3.9 and to allow instructions in the *RSs* to fetch their missing operands as soon as they are sent on the *CDB*.

- `value`: the actual result of the instruction or the auxiliary data if an exception occurred during the execution. As an example, if a *load page fault* exception is raised during the *DTLB* access of a *load* instruction, the faulting virtual address is sent in this field. The carried data is always stored in the *result* filed of the *ROB*, being it the data to be stored in a destination register or the exception auxiliary data to be used during commit for exception handling.

- `except_raised`: this status bit is asserted to inform the *ROB* that an exception occurred, so the instruction in its destination entry can be tagged as faulty too. At commit, the exception handling logic (see section 3.10.1) will perform the necessary operations.

- `except_code`: this field contains the encoded exception. The carried code is copied in a dedicated field in the *ROB* and used at commit by the exception handling logic to choose what to do. Usually, this code is copied in the `mcause/scause` CSR.

Besides, also the *CDB* uses the AXI-like handshake protocol. This is managed by a dedicated arbiter, that is presented in the next section and that drives the multiplexer that selects the data from the selected *RS*.

3.8.1 CDB arbiter

The *CDB* arbiter has one priority channel that is always served. This is meant to be connected to the *load buffer*, so the loaded values are available after a reduced latency to other units. The other channels are managed by a priority encoder with fixed priority. However, to make the scheduling algorithm a bit fairer, the input vector in a certain cycle is saved inside a register in the *CDB arbiter*. No other input vector is accepted until all previous active requests have been processed. In each cycle, the served request is masked and won't be considered in the next cycle, until the saved request vector is empty. When this happens, a new input vector is

processed. The priority encoder acts on the masked request vector stored in the previous cycle. When a request is selected, the associated output *ready* signal is asserted to notify the source *RS* that the request is served, and the instruction can be popped at the next active edge of the clock.

Of course, the priority channel doesn't follow this mechanism, and it is always served regardless of the remaining stored requests.

A schematic of the *CDB arbiter* is reported in fig. 3.11.

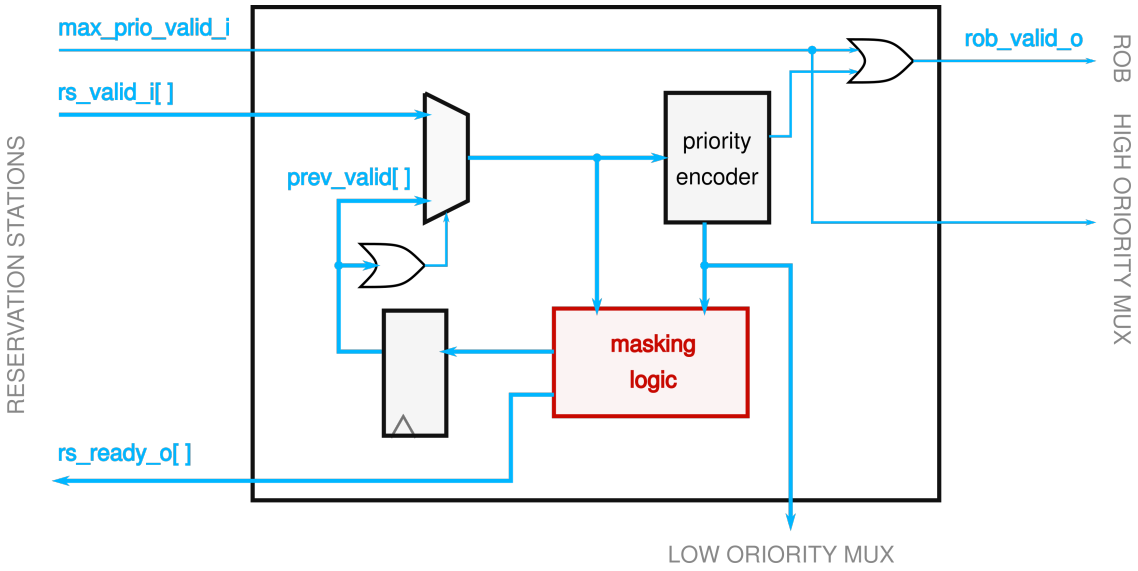


Figure 3.11: Schematic of the *CDB arbiter*. The *masking logic* contains a simple decoder that asserts only the *ready* of the served *RS*, and only if no request is being made by the top priority unit (`max_prio_valid_i` is low). Besides, it produces the remaining valid requests that must be processed in the next cycle. These are simply the ones of the current cycle except the one being served).

3.9 Reorder Buffer

The *ROB* is one of the most important components of the *LEN5* execution pipeline. It is mostly managed in-order, with a head and a tail pointer. The *ROB* is no different from the other data structure encountered in the execution pipeline. The interface and internal organization of the *ROB* are quite similar to the ones of the *issue queue*. Therefore, refer to fig. 3.6 and to the code for the details of its implementation.

3.9.1 ROB data structure

Each *ROB* entry contains the following fields:

- `valid`: a status bit indicating whether the contained instruction is valid.
- `instruction`: the entire instruction.
- `instr_pc`: the program counter of the contained instruction.
- `res_ready`: the result of the instruction produced by the associated *functional unit* and sent to the *ROB* by the *RS* using the *CDB*.
- `res_value`: the actual result of the instruction (if any). Notice that if an exception has been raised during the execution process, this field is used to store the auxiliary exception data, as the offending virtual address of the *load* instruction that resulted in a *load page fault* exception during the *DTLB* access.
- `rd_idx`: the address of the destination register (`rd`, if any).
- `except_raised`: a status bit indicating if the contained instruction encountered an exception during its execution.
- `except_code`: the code associated to the raised exception.

An example of the content of the *ROB* is reported in table 3.6.

3.9.2 ROB control logic

The control logic of the *ROB* is quite similar to the one of the *issue queue*, except for an additional write port to save the result of an instruction from the *CDB* to the indicated entry. The possible operations are briefly described in the following paragraphs.

	#	v	instr	instr_pc	res_r	res_v	rd_idx	e_r	e_code
$t \rightarrow$	0	no	-	-	-	-	-	-	-
	1	no	-	-	-	-	-	-	-
$h \rightarrow$	2	yes	addi	0x...0dead0	yes	42	5	no	-
	3	yes	lb	0x...0dead4	no	-	31	no	-
	...	...	...	...	...	...	...	...	...
	m-1	yes	sh	0x...0beef0	yes	0x...	-	yes	0x7

Table 3.6: Example of the content of the *ROB*. Here, the `addi` instruction in entry 2 has completed its execution (i.e. it has its result ready) and reached the head of the *ROB*. Therefore, it is ready to be committed. The `lb` instruction in entry 3 doesn't have its loaded value yes, while the `sh` instruction in entry $m-1$ raised a *store page fault* exception during the *DTLB* access, so it is ready to commit without further checks. The exception handling logic will perform the necessary operations.

Insert a new instruction As described before when a new instruction is issued, it is sent to both the *RS* assigned by the *issue logic* (if any) and to the tail entry of the *ROB*. Of course, both the *RS* and the *ROB* must have an empty entry where the instruction can be allocated, otherwise, the *issue queue* is stalled. If the tail entry of the *ROB* is not empty (i.e. the *ROB* is full), the output *ready* signal to the *issue logic* is not asserted, and the *ready* from the *issue logic* to the *issue queue* won't be asserted too, stalling the issue of new instructions until the one in the head entry of the *ROB* commits.

Save result from the CDB An additional write port is connected to the *CDB* and uses its `rob_idx` field to address the destination *ROB* entry where the carried result or exception must be stored. The *valid* signal from the *CDB* triggers this action.

Pop a committed instruction Once a valid instruction reaches the head of the *ROB*, if it has its result ready[24], it is sent to the *commit logic*. As soon as the *commit logic* has performed all the necessary operations for the commit of the current instruction, it asserts its *ready* signal for the *ROB*, and the head instruction is popped. This action completes the execution process of an instruction in the execution pipeline. Starting from the next cycle, its result will be available in the associated *RF* or in the memory, unless an exception was raised or the instruction changes the execution flow or the status of the processor.

[24]When no result is produced by an instruction like *stores* or CSR ones, the result is considered ready so the commit can proceed.

3.10 Commit logic

The *commit logic* is the combinational control network that performs all the required actions on the committing instruction according to its type and possibly its result and operands. After an instruction has been processed by the *commit logic*, it can be popped from the *ROB*. If an exception was raised during the previous phases of execution, the exception code and its auxiliary information from the head entry of the *ROB* are processed by the internal *exception handling* logic briefly introduced in section 3.10.1.

Notice that the *commit logic* requires all the processor parts to be integrated properly before it can be completely defined and designed. For this reason, the code of *LEN5* doesn't include this component. A simple template is provided instead, to be completed as soon as the integration is made.

The *commit logic* contains a second *instruction* decoder that takes different actions based on the type of the currently committing instruction, which is the one from the head entry of the *ROB*. The *commit decoder* follows the same organization of the *issue decoder*. It generates an output *ready* signal for the *ROB* to communicate that the committing instruction has been processed and asserts the output *valid* signals for the integer and the floating-point *RF* and for the CSRs. When an instruction can be successfully committed, the output *ready* signal fro the *ROB* is raised. Besides, it generates the control signals that are used by the main control logic that together with the ones from the *issue logic* operate the required actions of the different parts of the processor. Also, it contains the logic required to handle exceptions that may have arisen during the fetch, the issue or the execution phases as described in the previous sections of this chapter.

The *commit logic* must communicate with the main control logic of the entire processor and possibly interact with the frontend and the memory system when instructions that modify the sequential flow of the program (i.e. update the PC) are committed. For this reason, the code of the *commit logic* has not been written in this first implementation of *LEN5*. However, its design is overall very similar to the one of the *issue logic*.

The following paragraphs describe how the main types of instructions are handled by the *commit logic*.

Arithmetic instructions Arithmetic instructions can adopt either the R-type or the I-type instruction format, depending on whether an *immediate* value is used in place of one of the source registers. Regardless of the type, they share the property of updating their destination register in the integer or the floating-point *RF*. Therefore, when an arithmetic instruction is decoded by the *issue decoder*, the *valid* signal for the corresponding *RF* is asserted, and the destination register index and the data are presented at the input of its write port. Also, the *ready* signal for the *ROB* is asserted so the head instruction can be popped as described

in section 3.9. As explained there, a *store* instruction is allowed to perform the *data cache* access only if no exception is registered in the *store buffer*.

Memory instructions *Load* instructions are no different from arithmetic instructions when they commit: they carry a result that has to be saved in their destination register. For this reason, they are handled by the *commit logic* exactly like arithmetic instructions. The only difference is the type of exceptions that might have been raised during execution. *Store* instructions, on the other hand, follow a slightly different procedure. As mentioned in section 3.7 they are not directly committed from the *ROB*. In fact, both the target physical memory address and the value to be stored are kept in the *store buffer*, which is compiled in order exactly like the *ROB*. A *store* can perform the *data cache* write request if and only if it is found both in the head entry of the *ROB* (like any other instruction) *and* in the head entry of the *store buffer*. Once the *data cache* communicates that the write operation has been successfully completed, the instruction can be popped from the *ROB*. The *commit logic* takes this acknowledgment signal from the *store buffer* and asserts its output *ready* for the *ROB* accordingly.

CSR instructions handling As mentioned before, CSR instructions are executed in program order at commit time to preserve the dependences of other instructions on the processor status. It is not possible to execute them during issue because it is not guaranteed that their operands are ready. When a CSR instruction commits, the *commit logic* performs the *RF* access to fetch its operand, and use the address of the target register to access the CSR data structure described in section 3.11. Since the *issue queue* is stalled when a CSR instruction is decoded, there is no need to flush it when the same instruction commits. Instead, when this happens the *commit logic* simply enables the *issue queue* again, so the already fetched instruction can be executed with the updated values in the CSRs. Another solution could be not to stall the issue, speculating that no instruction is going to read any CSR, and instead track if any of the in-flight instruction is actually using the CSR that is going to be updated. If so, the pipeline is flushed when the CSR instruction commits. However, this solution is quite expensive and would significantly complicate the control system of the execution pipeline. Also, CSR instructions are usually only a very small fraction of the entire program if compared to user space instructions. So, according to Amdahl's law, the improvement in performance given by this kind of speculation would be too small to justify the increased complexity. That would also be in contrast with the initial design principle of making the *common case* fast.

Special instructions Other instructions need additional actions to be performed at commit time. As an example, the `jal` and `jalr` instructions modify the PC. In this case, both the in-flight instructions, *issue queue* and the *fetch unit* must

be flushed before the new PC value can be loaded. This is also the case when instructions changing the privilege mode or the address space are committed. The *commit logic* and the *commit decoder* take all the required actions to communicate to the main control logic of the processor what is to be done in each of its parts. Notice that the memory system supports *abort* requests from the main control only for *load* instruction. Once a *store* request to the *data cache* has been accepted it cannot be aborted. This is totally fine because a *data cache* request from a *store* instruction is performed only if the store instruction reached the head of the *ROB* and of the *load buffer* with no registered exceptions. So only if it actually meets all the requirements to modify the memory and only if all previous instructions, including the once modifying the processor status and privilege mode, have already committed successfully.

3.10.1 Exception handling

The exception handling logic is responsible to take all the operations required to load the exception handling routine. This entire process is defined by the RISC-V specification, specifically the privileged ones [16]. Usually, the following steps must be performed:

1. The *cause* of the exception, encoded in the `except_code` filed of the *ROB*, is stored in the dedicated `mcause/scause` CSRs.
2. Additional information, like the faulting virtual address of a memory instruction available in the `res_value` field of the *ROB*, can be saved in the `mtval/stval` CSRs.
3. The PC of the instruction is stored in the `mepc/sepc` CSRs.
4. Finally, the PC of the exception handling routine associated to the raised exception is loaded from the `mtvec/stvec` CSRs and sent to the frontend. Notice that the support for this operation is not yet available in *LEN5*.
5. he `mscratch/sscratch` CSRs can be used by the exception handler to store temporary data.

Usually, when an exception occurs, the control is passed to the OS that operates in a privileged mode. For all the details it is suggested to read [8, chap 5].

3.11 Control Status Registers

In *LEN5*, the CSRs are organized as a register file. Each CSR has an associated address that is also used to check if the CSR instructions has the privileges to modify the target CSR. Usually, the operation on a given CSR must be atomic, and most of the time only *legal* values can be read or written. The RISC-V privileged specification [16] details every single aspect of the CSRs access. The implementation is straightforward, so it is not analyzed here.

Some CSRs are used to evaluate the performance of the processor in real-time during execution. These register are essentially counters incremented when specific events take place. In particular, there are a free running *clock cycle counter* (`ucycle`), a *real time counter* (`utime`) and a *retired instruction counter* (`uinstret`). Those are specific to the *user space*, but the same registers might be instantiated also for *supervisor* and *machine* modes. The content of the performance counters in each cycle is strictly implementation-dependent, and their function cannot be completely defined, especially for OoO and multi-core or Simultaneus Multi-Threading (SMT) cores. In this first implementation of *LEN5*, the user space performance counter are provided, but their control signals are not yet connected.

Chapter 4

Testing and synthesis

In this section, the testing methodology will be discussed, and some of the synthesis results will be reported.

4.1 Functional verification

Proper testing of a microprocessor requires a quite complex environment. A testing *space* must be defined, aiming at covering all the possible situations the processor can encounter during its execution. During functional simulation with the compiled net-list, the testbench should be able to track the *coverage*. Notice that simulating all the possible combinations of instructions is usually practically impossible. So, the trend in this field is to make use of *formal verification* too. Besides, many sequences of instruction should be defined to stress the critical section of instruction execution and flow control. To do this, the entire processor must be integrated, including the *functional units*, that can be also emulated in the testbench. The outcome of a certain code should be then compared to the output of a software model of the processor. After functional verification, some sort of benchmarking should be performed to evaluate the performance of the processor and compared them to the existing solution. Eventually, the critical sections can be optimized. In general, testing is the step of the entire design process that takes longer to be completed, especially when dealing with very complex systems.

Unfortunately, the three main parts of *LEN5* have not been integrated yet. Moreover, some components that operate on the whole processor and not only on the execution pipeline (like the *commit logic* and the exception handling) have not yet been fully defined and described in System Verilog[1]. For this reason, it wasn't possible to simulate the execution pipeline comprehensively. In some cases, some of the missing hardware was described in a test-focused manner, meaning that

[1] A discussion of what remains to be done can be found in section 5.2.

only the logic required to carry on the defined test scenario was actually described. These dummy components were not inserted in the *LEN5* code since most of them have to be most certainly completely redesigned in a proper and comprehensive way. The next section describes how the components of *LEN5* were tested to work as expected.

4.1.1 Testing methodology

Because of the missing integration of the other parts and of the actual *functional units*, the execution pipeline could be tested properly. Instead, each of its internal components was tested on its own, forcing its inputs to assume all the critical values and also random one to see if the control logic was able to detect a forbidden state and raise an exception. To this purpose, many System Verilog *assertions* were introduced in the code. Then, a testbench dedicated to each of the units from chapter 3 was defined and used to generate the input vectors. In most of the case, the main output signals were printed on an output simulation log checked automatically by some simple Python script or verified manually when necessary. In the most critical cases, the waves of the simulator were analyzed to check if the content of a data structure or the control signals to update it matched the expected ones in each cycle. In many cases, some timing diagrams[2] were realized during the initial design, and used to check the waves produced by the simulation[3].

Very simple data structures like the *issue queue* were simply tested simulating the state of the connected units and producing ad hoc instructions or data to be inserted in the structure.

Due to the lack of the *functional units*, only the *RS* associated with the integer ALU was tested, emulating the ALU with a behavioral component in the testbench. Remember that the same arithmetic *RS* is instantiated for each of the arithmetic *functional units* in the execution pipeline (integer multiplier, integer divider, and floating-point unit). So testing one of them should be quite significant also for the other ones in most of the cases. The correctness or type of the result is, in fact, independent on the *RS*, while what's important is the flow control of the execution and the handshake mechanism with the *functional unit*, the *issue logic*, and the *CDB*, that are the same in every *RS*.

Complex control logic like the *issue logic* was manually verified to work as expected only in a limited number of cases since many instructions still have to be supported by the *commit logic*, especially those regarding privileged execution modes. A comprehensive verification of the *issue logic* and decoder can be easily

[2]Not included in this document.

[3]No wave is reported here, since it would be of little interest for the reader, given its general complexity and the knowledge of the code that is required to really understand it.

performed once the core is entirely assembled. At that point, any sequence of supported instructions can be simply fed into the frontend, while the output can be checked tracking the content of the register and of the memory.

For all these reasons, the functional verification was performed in a heuristic way, and the fact that each component works properly o its own doesn't mean that the entire pipeline or processor will, because not all the possibilities were covered and some bug might still be present.

While this means leaving some work to be done by those that will further develop *LEN5*, it is also important to remember that the initial aim wasn't to deliver a fully functional and usable speculative, OS compliant OoOE processor, since this would require much more time and resources than what three students can provide during a Master's thesis project.

4.2 Synthesis results

The synthesis was performed using the *UMC* 65 nm low-leakage library [14]. In particular, *typical* gate parameters were employed. The target clock period was set to 0 ns to let the compiler do all the possible optimizations to increase performance.

Assembling and synthesizing the entire execution pipeline when some of its parts are missing would result in mostly insignificant figures of merit. Therefore, once again each component was synthesized on its own, or at most with some of the surrounding ones (like the *load-store unit*), to extract some *relative* information and identify the critical sections of the design. For each synthesized unit, the total area, including both combinational and non-combinational elements, the non-combinational (NC) area and the cycle time (or propagation delay for combinational blocks) were extracted.

Some system-wide parameters can still be obtained by the individual ones if some assumption is made.

- The *area* of the currently available hardware in the execution hardware can be obtained as the sum of the areas of each component.
- The global *minimum cycle time* can be obtained as the maximum *slack time* (absolute value) among all the units.

Where *slack time* is the difference between the beginning of the clock arrival time, that is the clock cycle period plus the clock skew, and the data arrival time. A negative slack time means that the target clock cycle is not met. Since the target clock cycle time was set to 0 ns, here the value of the slack time is actually the cycle time obtained by the synthesis compiler for a given sequential component changed in sign.

Notice that both these results, reported in section 4.2.11, *won't* accurate. For what concerns the area, a more accurate logic optimization can be achieved by the

compiler when the entire design is flattened and processed at once. So the result might represent a worst-case scenario, which is therefore not totally meaningless. However, on the other hand, the fan-out of each output port of a unit is set to a standard value during synthesis. When the unit is integrated into the entire system, the fan-out can be different. A significant example is the *CDB*: its data is broadcast to all the *RSs*, so its fan-out is possibly much higher than the one assumed during synthesis. As a consequence, there isn't a defined relation between these results and the ones obtained properly synthesizing the entire processor.

Similar considerations might be done regarding timing. The global critical path might represent a best-case scenario since not all the units have input or output register, so when they are connected the number of logic levels a given signal must traverse might be bigger than it is in the stand-alone synthesis.

So, all the numbers reported in the following section are to be considered *preliminary* results, that might not significantly represent the actual figures of merit of the execution pipeline or *LEN5* in general. They are meant to highlight the critical section of the processor, and not to provide an absolute rating of its performance.

Before proceeding to the actual results, some of the general assumptions regarding the synthesis must be pointed out and explained.

Input and output delays Synopsys lets the user choose an *input* and an *output* external delays to be associated to each register in the design[4]. The input delay is added to the propagation delay of the logic between two registers, practically hiding the set-up time of the register. The output delay is added to the clock-to-output delay. So both these parameters enter in the definition of the well-known definition of *critical path* (and consequently of the cycle period):

$$T_{cycle} = T_{c \to q} + T_{skew} + T_{pd} + T_{su}$$

For this reason, it was decided to set those synthesis parameters to 0 ns. When needed, they can simply be added to the reported timing results, since their contribute is linearly combined to the other ones. Of course, the same is true for the system-wide parameters defined above and reported at the end of this section.

Clock uncertainty This is another parameter that is usually set in Synopsys[5] to take into account that the generated clock doesn't have an ideal, constant period. Instead, it is subject to a certain variation. Since this parameter is strictly dependent on the implementation of the clock generation network, that, of course, goes beyond the purposes of this work, it was again considered null (0 ns). As for

[4]This is accomplished by the `set_input_delay` and `set_output_delay` commands.

[5]Using the `set_clock_uncertainty` command.

the input and output delays, the actual parameter can be simply added to a given critical path in a second time.

Output load As mentioned before, synthesizing each component on its own might lead to inaccurate results because the fan-out of the output ports assumed during synthesis might not be representative of the actual one when the entire system is integrated and synthesized. Unlike the previous ones, this parameter has consequences both on the area and the timing results, since a bigger load imply bigger or multiple drivers to handle the required logical effort. However, since this *preliminary* synthesis results do not aim at providing realistic figures of merit, but only at highlighting what are the critical sections of the design, the output capacity was simply set to the input capacitance of a buffer with *drive strength* equal to 4 according to the definition in the library documentation[6] [14].

Other synthesis settings To get the *preliminary* reference results, no *retiming* or Synopsys advanced compile mode (like *compile ultra*) was used. The system-level parameters from the configuration packages are reported in the section dedicated to each component. Since the number of entries in the *ROB* has some consequences also on the other units, its value is considered set to 16 if not differently stated.

4.2.1 Arbiters

Both the two types of arbiters described in section 3.1.3 should be quite simple, consisting only in a few logic gate levels. The fair arbiter also contains a Flip-Flop (FF), so it is expected to be bigger and maybe a bit slower due to the technological parameters of the sequential element. Still, the impact of choosing one over the other is totally negligible in the entire system, and this choice should be made based only on global considerations on instruction dependences or global execution latency.

The synthesis results for both the types of 2-way arbiters are shown in table 4.1, and confirm the expectations.

Arbiter type	**Area (NC area)** [µm^2]	**Latency** [ns]
Priority	7.56 (0)	0.07
Fair	35.64 (12.24)	0.18

Table 4.1: *2-way arbiters* synthesis results.

[6]The corresponding gate in the UMC 65 nm technology library is the `BUFM4R`. The input capacity of its only input `A` was used as output load capacitance for the output ports.

4.2.2 Entry selectors

Both the entry selector were synthesized with 8 and 16 inputs since those are the values used for the number of entries in the *RSs*, as described in section 4.2.7. Notice that being combinational components, the non-combinational area of the selectors is always zero.

Priority encoder

The priority encoder, whose results are shown in table 4.2, is a very simple and fast structure, and shouldn't introduce much latency in the communication between the *RSs* and the associated *functional unit*. However its propagation delay increases almost linearly with the number of inputs, so if used in very large data structures it could enter the critical path of the entire system.

Input n.	**Area** [μm^2]	**Delay** [ns]
8	31	0.18
16	66	0.33

Table 4.2: Priority encoder selector synthesis results with 8 and 16 inputs.

Age based selector

The age based selectors were synthesized with 4-bit age inputs (up to 15), so compatible with a 16 entries *ROB*. The results are shown in table 4.3.

Input n.	**Area** [μm^2]	**Delay** [ns]
8	233	3.45
16	519	8.05

Table 4.3: Age-based selector synthesis results with 8 and 16 inputs. The age is encoded on 4 bits.

As shown, the age-based selector is a very, very slow component, and its delay grows more than linearly with the number of input ages. Originally it was introduced to give higher priority to older instructions during execution. Notice that as mentioned in section 3.1.4, the age-based selector was described in a behavioral way in System Verilog. Still, a structural version was designed too. Despite the possibility to manually optimize this component, it is very unlikely for it to be competitive with a priority encoder: it would almost certainly enter the critical path, given the presence of multiple adders and comparators. Even if in theory it could slightly reduce the execution latency of instructions, its huge delay would nullify any sort of advantage. For this reason, by default is **not** instantiated by default.

4.2.3 Issue queue

The issue queue is a quite simple FIFO buffer, so it is not expected to be a critical component in the execution pipeline. It was synthesized both with 8 and 16 entries. The resulting area and timing values are reported in table 4.4.

Entry n.	**Area (NC area)** [µm^2]	**Cycle time** [ns]
8	17 554 (14 509)	0.51
16	38 332 (23 472)	0.60

Table 4.4: *Issue queue* synthesis results with 8 and 16 entries.

As shown, most of the area is occupied by sequential logic: the *issue queue* data structure itself and the head and tail counters. The quite simple control logic does not represent a bottleneck, and the cycle time is only slightly increased doubling the number of entries probably due to the more complex addressing logic. This is supported by quite a substantial increase in the combinational area when the number of entries is set to 16.

In general, the size of the *issue queue* can be set to meet the system requirements without impacting the overall performance of the core.

4.2.4 Issue logic

The issue logic is one of the slowest combinational components in the entire execution pipeline and probably in the entire processor, given the presence of the *issue decoder*. It was synthesized assuming 7 *RSs* instantiated in the execution pipeline, as described in section 3.6. The synthesis results in table 4.5 show a quite high delay. In particular, its critical path includes the instruction decoder and the operands fetch logic.

Area [µm^2]	**Delay** [ns]
1353	1.96

Table 4.5: *Issue logic* synthesis results.

As suggested in section 5.3.1 the decoder organization might need a revision in future versions. Also, remember that the operand fetch also requires the *register status* and the *RF* or the *ROB* to be accessed. The *RF* access is carried out in parallel to the rest of the instruction decoding, since the `rs1` and `rs2` register indexes are read from the instruction regardless of its format or decoding process. The *ROB* access can be performed once the *ROB* entries providing the operands are fetched from the *register status* data structure. Assuming that the read ports of the *register status* and of the *ROB* have a latency equal to their cycle time (from

table 4.6 and table 4.14), the resulting total latency of the operand fetch operation is:

$$T_{op.fetch} = T_{reg.stat.} + T_{ROB} \tag{4.1}$$

The operand fetch from a 16-entry *ROB* would take 1.22 ns, from a 32-entry *ROB* it takes 1.25 ns and from a 64-entry *ROB* it takes 1.38 ns.

This operation goes in parallel with the instruction decoding, and when this is completed, the *issue logic* selects the right operand value from the correct source. So, the operand fetch from the *RF* and from the *ROB* through the *register status* is performed **in parallel** to the instruction decoding. The reported critical path regards exactly the operand selection in the *issue logic*. Since this critical path is higher than the fetch operation, the latter is completely hidden by the latency of the decoding logic, so the *ROB* and the *register status* are not critical components of the execution pipeline.

4.2.5 Register status

The *register status* is a quite simple structure. Despite having 32 entries, their content is quite small: a status bit and a *ROB* index. Therefore, it's complexity is expected to be limited, while its delay should be in the same order of magnitude of the other simple data structures, like the *register files* or the *issue queue*. The synthesis results in table 4.6 confirm these expectations. Notice how the area is mostly occupied by sequential logic (i.e. the data structure itself).

Area (NC) [µm^2]	**Cycle time** [ns]
4605 (2191)	0.60

Table 4.6: *Register status* synthesis results.

4.2.6 Register files

The only difference between these two components is that the first register (0) of the integer register file is hard-wired to 0. In both of them, two-thirds of the area is occupied by sequential logic, while the remaining by the access ports, naming MUXs and decoders.

Both the area and timing reports are shown in table 4.7.

Register file	**Area (NC area)** [µm^2]	**Cycle time** [ns]
Integer	32 585 (21 572)	0.35
Floating-point	32 752 (22 448)	0.29

Table 4.7: *Register files* synthesis results.

The integer *RF* shows a lower non-combinational area due to the lack of one of the registers. However, the combinational area and the slack time are slightly higher due to the additional check on the input register indexes.

4.2.7 Reservation stations

The *RSs* were synthesized without instantiating the *age based selectors*, for the reasons explained in section 3.1.4 and section 4.2.2. The results were obtained for both an 8-entry and a 16-entry version of both types of *RS*.

Arithmetic reservation station

The synthesis results of the arithmetic *RS* are reported in table 4.8. They were obtained assuming a generic *functional unit* whose possible operations are encoded on 4 bits and whose generated exceptions are encoded on 2 bits. Since these parameters change only the size of each entry, they shouldn't affect the timing performance of the unit.

Entry n.	**Area (NC area)** [μm^2]	**Cycle time** [ns]
8	26 762 (16 629)	0.59
16	52 804 (33 389)	0.75

Table 4.8: Arithmetic *RS* synthesis results with 8 and 16 entries.

As shown, despite the sequential elements dominate the area of a *RS*, the more complex control logic and the selectors have a greater impact than in simpler in-order structures like the *issue queue* (section 4.2.3): 39 % compared to 17 % in an 8-entry *issue queue*. On the other hand, the delay is not increased as much, because the control logic is mostly parallel as a consequence of the organization discussed in section 3.1.2. When doubling the number of entries to 16, the area doubles too, indicating a linear dependency. The increase in the delay is higher probably due to the delay of the bigger priority encoders used as selectors (see section 4.2.2).

Branch unit reservation station

Similar results are expected from the synthesis of the *branch unit RS*. This time, the selection of the *branch* instruction to be executed is performed in-order by some counters instead of the priority encoders used in arithmetic *RS*. The additional sequential components might lead to a larger component, while the lack of priority encoders should reduce the critical path, especially with a higher number of entries. The obtained results are reported in table 4.9, and confirm the expectations.

The previous results classify the *RSs* as non-critical components in the execution pipeline of *LEN5*. This concurs to defining some of the advantages of Tomasulo's

Entry n.	**Area (NC area)** [µm^2]	**Cycle time** [ns]
8	36 380 (20 176)	0.55
16	68 931 (45 682)	0.63

Table 4.9: *Branch unit RS* synthesis results with 8 and 16 entries.

approach discussed in chapter 2. *RSs* allow OoOE while distributing most of the instruction execution logic, avoiding centralized complex data structures that are typical in different approaches like scoreboarding.

Doubling the number of entries doubles the area while increasing the cycle time by 15 %. Notice that increasing the depth of a *RS* over 8-16 is very unlikely to bring any advantage, especially if only one *functional unit* is associated to that *RS*: it is quite rare to issue as many instructions of the same time as required to fill a *RS* before the maximum *ROB* capacity is reached.

4.2.8 Load-store unit

The *load buffer* and the *store buffer* are two of the most complex components of the execution pipeline of *LEN5*. Their data structures are not much different from the one of an arithmetic *RS*, with the only difference being the size and fields of each entry. The control logic is slightly more complex to support the additional operations required to execute *loads* and *stores*. Still, each operation is processed in parallel, so even if a larger combinational area is expected, it shouldn't impact on the delay. The real critical section for what concerns timing performance is the forwarding logic in the *store buffer*.

Both the buffers were synthesized separately to distinguish the critical sections of each of them. At the end of this section, a synthesis of all the *load-store unit*, including the *virtual address adder* and all the arbiters and MUXs instantiated to handle shared unit units (the *virtual address adder*, the *DTLB*, and the *data cache*).

Load buffer

The *load buffer* is shouldn't perform much different from an arithmetic *RS*. Still, as mentioned above it is expected to have a larger area. The obtained results are shown in table 4.10.

Entry n.	**Area (NC area)** [µm^2]	**Cycle time** [ns]
8	49 588 (21 456)	0.79
16	92 514 (41 404)	0.97

Table 4.10: *Load buffer* synthesis results with 8 and 16 entries.

The area of an 8-entry *load buffer* is similar to the one of a 16-entry arithmetic reservation station, with this being due primarily to the combinational logic, that occupies the 57 % of the total area. The several selectors are the main reason for this. The cycle time is higher than the one of a *RS*, probably because of the more complex control logic, that requires additional sequential checks.

When doubling the number of entries, the area increases linearly, while the cycle time is increased by 22 %. Notice that having more than 16 entries in the *load buffer*, like in every other *RS*, is not probably worth it. The *ROB*, that contains more than one type of instructions and has limited capacity too, would fill before as many instructions of the same type as the depth of a *RS* are issued. Only a comprehensive simulation od the entire processor or an accurate code analysis can suggest the best depth of these units.

Store buffer

As mentioned at the beginning of this section, the forwarding logic in the *store buffer* is probably one of the bottlenecks of the entire execution pipeline, since many sums and comparisons must be performed to reconstruct the relative order of *load* and *store* instructions. Under any other aspect, the *store buffer* is not different from the *load buffer*, apart from using two counters instead of two selectors to insert and execute the *data cache*. All the other operations are carried out OoO. Once again, the results of the synthesis are shown in table 4.11.

Entry n.	**Area (NC area)** [µm^2]	**Cycle time** [ns]
8	49 173 (20 311)	0.92
16	102 521 (42 093)	1.11

Table 4.11: *Store buffer* synthesis results with 8 and 16 entries.

As expected, the combinational critical path of the *store buffer* goes through through the forwarding logic, producing a cycle time similar to the one of a *load buffer* with double the entries. This suggests an implementation with a *load buffer* bigger than the *store buffer*. However, this configuration may or may not bring advantages to the system: thanks to forwarding, the time a *load* spends in the *load buffer* waiting for its value is probably shorter than the time a *store* spends in the *store* buffer, since *stores* must be executed in order and must be kept in the buffer until the *data cache* access is complete. This means that the average occupancy of the *load buffer* is probably lower. On the other hand, the ratio between issued *loads* and *stores* is usually higher than one. So the choice depends on the programs executed. Still, if some advantage is found, this solution comes at no cost in terms of performance[7].

[7] the area and the power consumption will, of course, be higher.

Doubling the size of the *store buffer* leads to a cycle time 21 % higher, which might be unacceptable in some cases. On the other hand, a bigger *store buffer* means more space for the *cache level zero* mechanism, allowing a higher hit ratio and possibly less *DTLB* and *data cache* accesses.

Entire load-store unit

The forwarding logic in the *store buffer* is purely combinational. For this reason, also the (small) logic issuing and processing the forwarding operation from the *load buffer* should be included in the critical path of the entire *load-store unit*. Besides, the *virtual address adder* and all the arbiters must be considered. So, the entire *load-store unit* was synthesized to evaluate its actual performance in four different size configurations of the two buffers to evaluate the scaling of this unit. The results are shown in table 4.12.

LB entry n.	**SB entry n.**	**Area (NC area)** [μm^2]	**Cycle time** [ns]
8	8	95 646 (43 546)	1.44
16	8	134 628 (62 128)	1.56
8	16	140 951 (63 166)	1.63
16	16	179 685 (83 893)	1.89

Table 4.12: *Load-store unit* synthesis results with 8 and 16 entries in all the possible configurations.

In general, the area of the entire *load-store unit* is reported to be comparable with the sum of the same-sized *load buffer* and *store buffer* area despite the few additional components. This can be explained by the additional level of optimization that the Synopsys compiler can operate on the entire flatten *load-store unit* compared to the single buffers. For what concerns the critical path (and so the cycle time), it is incremented by 56 % while still including the forwarding logic, meaning that the *load buffer* contributes for about one-third in term of delay to the store-to-load forwarding process. Besides, the arbiters and the MUXs for the shader units, and the *virtual address adder* do not represent a bottleneck in the *load-store unit*. A simpler version of *LEN5*, that would probably have simpler decoders too, could, therefore, implement no forwarding logic and achieve better performance overall. However, this would strongly impact the efficiency of memory operations, so it really depends on the use case.

As it is clear from fig. 4.1 a bigger *store buffer* impacts more on performance than a bigger *load buffer* does. Instead, the area scales quite linearly with the total size of the buffers, both for sequential and combinational elements.

One last observation is that Synopsys used a *parallel prefix* architecture to implement the *virtual address adder*. This usually offers very good performance while keeping the area under control. This choice contributes to keeping this component

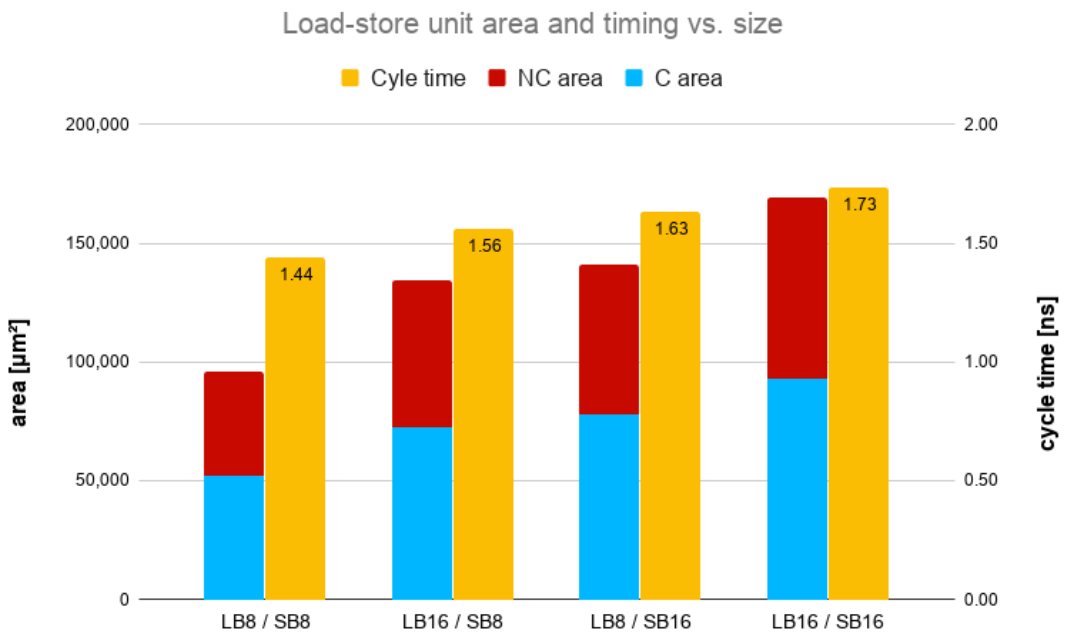


Figure 4.1: *Load-store unit* area and timing chart. The data is from table 4.12.

out of the critical path of the *load-store unit*.

4.2.9 Common Data Bus

Despite being a very simple component, the *CDB* is the component that mostly falls into the observations made at the beginning of section 4.2 about the accuracy of these synthesis results. In fact, as described in section 3.8, it is connected to *all* the entries of *all* the *RSs*. This is necessary to fetch those source operands that are not available at issue time. For this reason, its output capacity is quite high. Assuming that all the seven *RSs* have 8 entries (including the *branch unit RS*, the *load buffer* and the *store buffer*) and considering the destination *ROB* entry too, the data carried by the *CDB* must be broadcast to 57 registers, so its buffers might limit its timing performance. On the other hand, the *CDB arbiter* shouldn't be too slow since it only uses a decoder, an encoder, some MUXs and a very simple masking logic. The results of the synthesis of the *CDB* assuming seven *RSs* (i.e. seven couples of *valid-ready* signals) are reported in table 4.13. Also, a second synthesis was performed assuming 32 *RSs* (and therefore 32 *functional units*) to see if the encoding/decoding logic might represent an issue in more complex execution pipelines.

As it is clear from those results, the *CDB* is far from being a bottleneck in the system, but the increased fan-out in the fully integrated execution pipeline might

***RS* n.**	**Area (NC)** [µm^2]	**Cycle time** [ns]
7	1542 (60)	0.51
32	7284 (314)	0.92

Table 4.13: *CDB* synthesis results with seven *RSs*.

lower its performance and/or increase its area. Also with a very high number of *RSs*, the performance of the *CDB* is still acceptable, especially considering that such a complex pipeline would also require more complex *issue logic* and *ROB*, which are more likely to enter the pipeline critical path than the *CDB* is. From the two cases, the increment in the area is directly proportional to the number of *RSs*. This is in accordance with the higher number of registers required and the increased complexity of the encoding logic (see section 4.2.2).

4.2.10 Reorder Buffer

As mentioned before, the *ROB* is a quite simple component. It is compiled in-order using a head and a tail pointer, with two additional read ports to fetch operands and another write one to store the result carried on the *CDB*. On the other hand, it must provide enough space for all the in-flight instructions. Actually, it shouldn't be necessary to set the depth of the *ROB* to the absolute maximum number of instructions that can exist in the entire execution pipeline in a given cycle since this situation is hardly ever reached. In other words, the number of *ROB* entries doesn't have to be equal to the sum of the size of all the *RSs*. For the synthesis, the *ROB* was instantiated in three different sizes: 16, 32 and 64. Remember that a bigger *ROB* doesn't impact on the area other units inside the execution pipeline except for the few additional FFs that are required to store its indexes. Still, those are never processed by any combinational network except for the comparator that checks the commit conditions in the *store buffer* (see section 3.7.2). Most of the time they are only exchanged between units, without entering their critical path. So the *ROB* depth shouldn't matter at all also for the timing results in the previous sections, and the variation in the area results should be negligible as well.

The synthesis results for the *ROB* are reported in table 4.14.

Entry n.	**Area (NC)** [µm^2]	**Cycle time** [ns]
16	46 228 (24 580)	0.65
32	89 438 (54 152)	0.68
64	177 626 (99 107)	0.78

Table 4.14: *ROB* synthesis results with 16, 32 and 64 entries.

As shown, the *ROB* is smaller and faster than an equal-size *RS*, so it does not represent a critical unit in the execution pipeline, regarding neither area nor

timing performance. Its size can, therefore, be chosen according to the number of instructions that a specific configuration of the rest of the *LEN5* execution pipeline can handle at the same time (size of the other buffer, latency of the *functional units*, etc.).

4.2.11 Execution pipeline area and performance

As explained at the beginning of section 4.2, the system-wide parameters of the execution pipeline can be extracted from the ones reported in sections 4.2.1 to 4.2.10. However, due to the lack of the *commit logic*, the complete CSRs and the main control logic, these numbers are to be considered *preliminary ones*.

Also, some assumption is required:

1. As explained in section 4.2.4, the latency of a *register status* access is added to the one of the *ROB* access since these two operations must be performed one after the other. This operation is performed in parallel to the instruction decoding. For this reason, the *register status* and *ROB* access operations during issue are considered as a single operation whose latency is obtained from eq. (4.1) depending on the size of the *ROB*.

2. The *commit logic* is considered to have a lower impact of the *issue logic* for what concerns performance. The *commit decoder* is almost certainly smaller than the issue one, and the rest of the *commit logic*, is also quite simple, as described in section 3.10. For what concerns the area of the *commit logic*, it is considered to be about the same as the *issue logic*: $1353\,\mu m^2$.

3. Four of the five *RSs* (ALU, MULT, DIV, FPU) are considered to be equal to the arithmetic one from section 4.2.7, while the last one is the one associated with the *branch unit*, from section 4.2.7.

4. Since the number of CSRs depends on the implemented features, the area of the CSR data structure is assumed to be four times the one of a *RF* ($130\,\mu m^2$), because of the larger number of registers and the additional control logic (only *legal* values are accepted most of the time, as explained in section 3.11). Because the CSR register file doesn't contain complex combinational logic, its critical path is assumed to be similar to the one of the other *RF*, and therefore not significant for the system-wide performance estimation.

For what concerns the other system parameters, each *RS* is supposed to have 8 entries, including the *load buffer* and the *store buffer*. The *ROB* is assumed to have 32 entries and the *issue queue* 16.

It is important to remind that **no functional unit is included** in the following estimations. The intent of this synthesis process and the entire project was to focus on the architecture implementing the extended Tomasulo's algorithm. The

functional units can be chosen and inserted easily in the pipeline depending on the application requirements. Also, any arithmetic operator can be pipelined until it doesn't represent a bottleneck in the execution pipeline. In fact, the proposed architecture is completely independent of the type, latency and pipeline depth of the *functional units* employed. There are many examples of high-performance arithmetic operators that could be used in *LEN5*, and many of them are available under open-source licenses too. So once the entire processor is integrated, those units can be simply instantiated together with a dedicated *RS*, ensuring that they are fast enough to exploit the full capabilities of this execution pipeline.

As a consequence of the previous observation, the presence of the different *functional units* is, in theory, affecting only the *area* of the processor.

Area The estimation of the area of the entire execution pipeline is obtained simply as the sum of all its units, with the assumptions reported above.

$$A_{ex.} = A_{iss.q.} + A_{iss.l.} + 2 \cdot A_{reg.stat.} + A_{int.RF} + A_{f.p.RF} + 4 \cdot A_{ar.RS} + A_{b.u.RS} +$$

$$+ A_{l.s.u.} + A_{CDB} + A_{ROB} + A_{com.l.} + A_{CSR} \simeq 0.45\,\mathrm{mm}^2 \tag{4.2}$$

The 57 % of this area ($0.26\,\mathrm{mm}^2$) is due to sequential elements. The weight on the total area of each unit is shown in fig. 4.2. The reservation

As it is possible to see from the results in sections 4.2.1 to 4.2.10, in most cases the area of a unit grows *linearly* with the size of the buffers. Since most of the largest structures in the execution pipeline contain at least one buffer, also the increase in the total area should be linear with the total number of registers in the buffers.

Timing The estimation of the timing performance is obtained as the maximum critical path or cycle time among all the units. This is without any doubt the delay of the instruction decoding during issue: 1.96 ns. This means that the *estimated* maximum operating frequency of the execution pipeline using the *UMC 65nm* technology is:

$$f_{clk} < \frac{1}{T_{max}} = \frac{1}{t_{iss.l.}} \simeq 510\,\mathrm{MHz} \tag{4.3}$$

Notice that all the key components of the execution pipeline of *LEN5* are *faster* than the *issue logic*. So if the optimizations suggested in section 5.3.1 are applied (pipeline, better organization of the *issue decoder*) and the delay of the issue stage is removed from the critical path, the bottleneck will be represented by the *load-store unit* (see table 4.12) whose minimum cycle time is 1.44 ns. In the case the maximum clock frequency with the buffer sizes listed before would be:

$$f_{clk} < \frac{1}{T_{l.s.u.}} \simeq 694\,\mathrm{MHz}$$

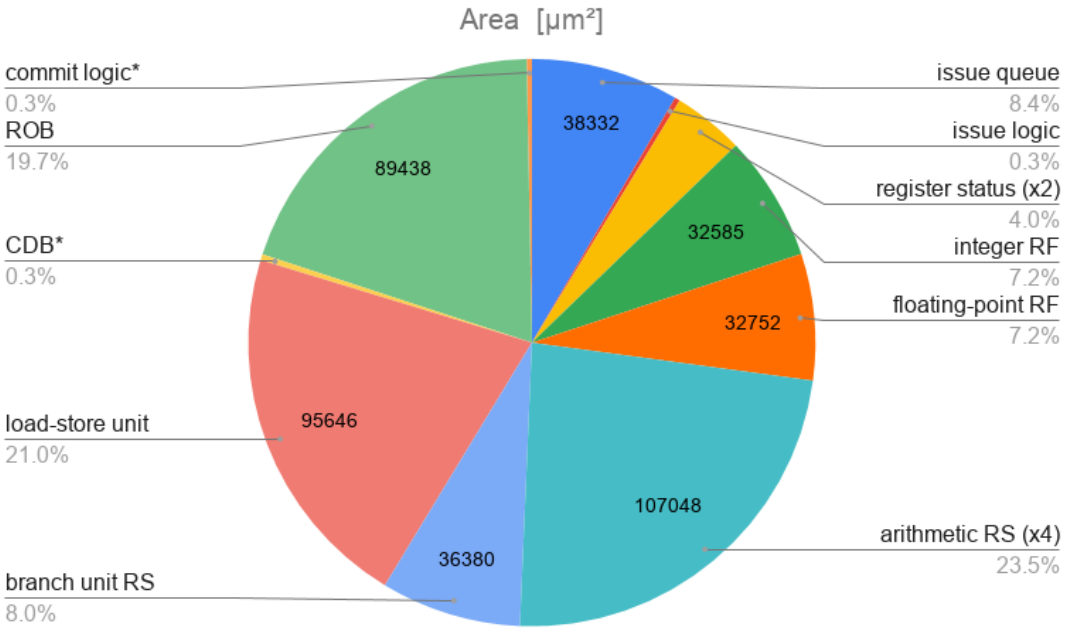


Figure 4.2: Area composition of the execution pipeline with 16-entry *issue queue*, 8-entry *RSs* (including *load buffer* and *store buffer*) and 32-entry *ROB*. The starred units use predicted or probably inaccurate values for the reason explained earlier in this section.

That is an improvement of 36 % over the result from eq. (4.3). Of course, changing the sizes of the *load buffer* and the *store buffer* slows the system down, but it may lead to better overall results decreasing the average number of stalls in the pipeline due to some buffer being full. On the other hand, the latency introduced by the *register status* and *ROB* accesses during the issue phase doesn't represent a bottleneck, being the sum of those operations in the worse conditions (64-entry *ROB*) lower than the cycle time of the *load-store unit* in the best conditions (8-entry *load buffer* and *store buffer*).

Figure 4.3 reports the cycle time (or the latency for combinational components) of each unit in the execution pipeline. Also, it shows the absolute increment in timing performance of some units when their size is doubled: from 8 to 16 in the *RSs*, in the *load buffer*, and in the *store buffer*, while from 32 to 64 in the *ROB*. Once again, the worst-case scenario for the operands fetch has been represented by the sum of the critical paths of the *register status* and the *ROB*.

As shown, the *load-store unit* is strongly affected by the number of entries in the *load buffer* and the *store buffer* because of the forwarding logic, as mentioned in section 4.2.8.

Once again, these are *preliminary results*. They might differ from the ones

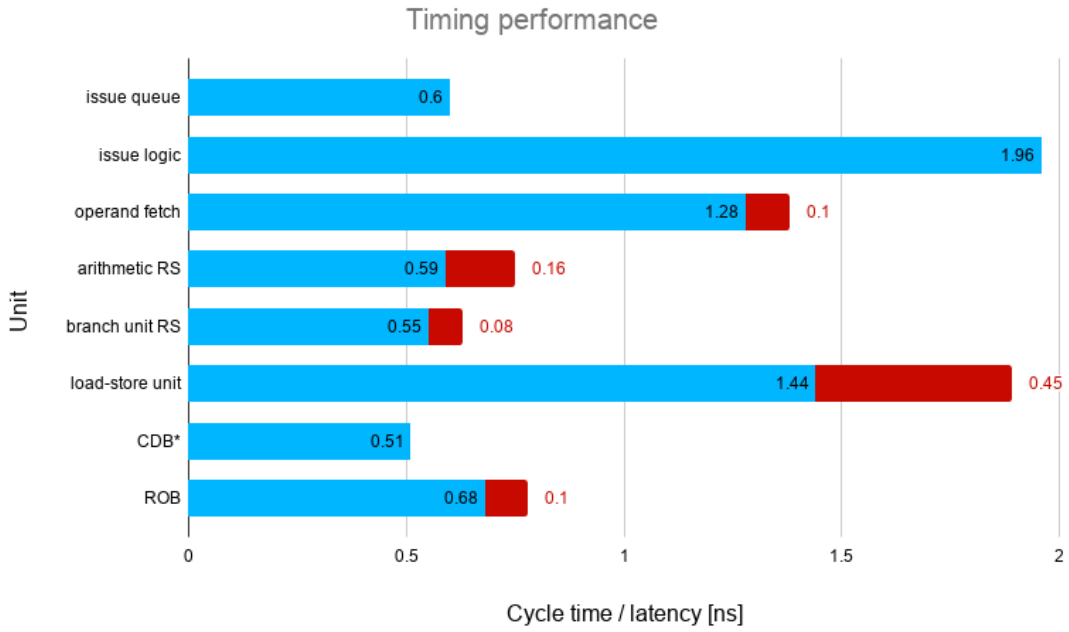


Figure 4.3: Comparison among the cycle times or latencies of each component in the execution pipeline, and absolute increment when the buffer size is doubled.

obtained by the synthesis of the entire system, especially if also the frontend and the memory system are integrated.

Chapter 5

Conclusions and further improvements

5.1 Conclusion

The design of the backend of *LEN5* exposed most of the critical issues in microprocessor design. The proposed architecture succeeds in addressing most of them by providing a modestly versatile, customizable and modular structure dedicated to a wide range of applications. These features meet the design principles the RISC-V ISA is based on, making it possible to extend the instruction set on demand. The extended Tomasulo's algorithm manages to hide most of the major sources of latency in the instruction execution process by inherently implementing *register renaming* and OoOE. At the same time it fully supports *speculation* and *precise exceptions*, two key features that are hardly ever missing in modern consumer and high-performance microprocessors. The resulting architecture, together with the system-wide AXI-like handshake protocol, simplifies the insertion of multiple and application-specific *functional units*, making it possible to adopt *LEN5* in high-performance applications, with dedicated SIMD or vector and matrix accelerators. The implemented *load-store unit* supports VM and employs many tricks to speed-up the execution of memory instructions while reducing the number of *cache accesses* by caching and forwarding values between *store* and *load* instructions whenever possible and caching already committed *store* instructions.

The synthesis results discussed in section 4.2.11 indicate that the key components of the execution pipeline architecture shouldn't represent a critical bottleneck in the entire processor. The *RSs*, the *CDB*, and the *ROB* successfully support all the advantages of OoOE while maintaining the delay quite small thanks to the simple and distributed execution control logic. The *load-store unit* is the slowest component apart from the *issue logic*, but this is not a limitation if all the processor parts are put together. The real bottlenecks in the pipeline are represented by

those components whose presence and implementation are mostly independent on the execution algorithm, with the *issue logic* being the most evident one.

Being this a first version of a very complex system developed in a few months, it is not a fully functional and usable core. Unfortunately, some additional work is required system-wide to integrate the frontend, the memory system and the execution pipeline of *LEN5*. The next sections describe what remains to do and hint some possible improvements of the backend that should further increase the performance of the core.

5.2 What is missing

As mentioned during chapter 3, the execution pipeline lacks some of the logic that is necessary to fully complete the instruction execution. Without overstating, almost each of the components of the backend of a processor might represent a Master's thesis project on its own to be completely optimized. The amount of knowledge to be added to what is known from academic courses is huge, and the series of implications hidden in a single statement in a computer architecture book is longer than expected most of the time.

In *LEN5*, most of the missing hardware was at least partially designed. However, some of it requires all the three parts to be integrated before it can be completely defined and described in the *LEN5* code. Namely, the units that must be added to the execution pipeline and to *LEN5* in general, are reported in the list below.

- If the M, F and D ISA extensions are to be supported, the *issue decoder* must be integrated with the definitions and controls for those instructions[1]. These extensions are already supported by the rest of the pipeline except for the *commit logic*.

- The *commit logic* with its *commit decoder* and *exception/interrupt handling* logic.

- The core top-level control system, that interacts with the frontend and the memory system besides the execution pipeline.

- Most of the CSR that are necessary to implement privileged execution.

- The *functional units*: ALU, MULT, DIV, FPU.

- Possibly, the interfaces with I/O devices and the main memory.

[1] All instructions from the base instruction set are already supported.

When all these units are ready and the three parts are integrated, the figure of merit of *LEN5* can be extracted, and further improvements can be applied to its critical section. Besides, benchmarking can be done to evaluate *LEN5* performance in comparison to existing solutions.

The next paragraphs briefly introduce some additional details about some of the components listed above.

Commit logic As described in section 3.10, this unit is responsible for the commit of instructions and the execution of those that cannot be executed earlier[2]. Most important, it handles exceptions or interrupts and generates most of the control signal for the top-level control logic of the entire processor. Some instructions change the flow of execution by changing the value of the PC, by entering a privileged execution mode, by accessing the CSRs and by setting some condition for memory or thread synchronization[3]. When those instructions are executed at commit, the other parts of the processor might have to be *flushed* or *stalled*. The *commit logic* generates the necessary control signals for the top-level control logic to take the required actions.

Top-level control This is the main control logic of the entire processor. It coordinates the operation of the frontend, the execution pipeline, and the memory system. Until a single-core and single-thread implementation are desired, the top-level control logic should be quite simple: each unit of the processor can be flushed using the dedicated control signal, while stalling can be achieved very simply by masking the input *ready* signals of a unit. If more than one core is instantiated, the top-level control logic must take care of all the synchronization and coherency issues too.

CSRs Not all the CSRs defined by the RISC-V specification are actually required. Their inclusion depends on the choices made during the implementation. Some CSRs are reserved to be used by the specific implementation in totally custom ways. Some of the CSRs have already been inserted in the *LEN5* code, but they have not been tested since the hardware to support CSR instruction is not ready yes as described in the previous paragraphs.

Functional Units The design of each *functional unit* for the execution pipeline was left as one of the last steps during the definition of the roles in this project.

[2] `fence`, `fence.i`, `fence.vma`, `wfi`, `mret`, `sret`, `ebreak`, `ecall`, `jal`, `jalr` and all the CSR instructions.

[3] This last class of instructions doesn't have any effect in *LEN5*, where a single core is present and only one thread is executed at a time.

Unfortunately, none of us was actually able to complete its part in time to proceed to the *functional unit* implementation. In addition to this, this is a well known and mature field, and many good arithmetic units can be actually found as open-source projects on the internet, even complex FPUs. For this reason, they had a secondary role in the design of *LEN5*. The main focus was on building the main core infrastructure.

5.3 Further improvements

This section is dedicated to hint some possible improvement of this preliminary implementation of the execution pipeline of *LEN5*. In general, each of its units offers many opportunities for improvement, since there was absolutely no time to optimize all of them during the past months. As stated in section 5.2, the design of a microprocessor is one of the most challenging projects that an electronic engineering student (and a graduated engineer too) may face. Despite being attractive, it also requires a huge amount of time, skill and knowledge to obtain results that can really contribute to defining the state fo the art in this field.

Here follow some of the possible improvements that might be applied to the execution pipeline of *LEN5* to improve its current performance and capabilities.

5.3.1 Overall optimization

As mentioned above, the focus during this first implementation was not on the optimization of the internal structure of each component. Therefore, there's still a lot of room for improvements in the system performance.

In particular, the issue phase represents the most evident bottleneck of the entire pipeline, as observed in section 4.2.4. Optimizing the *issue decoder* organization should increase performance as much as it is allowed by the next slowest component in the pipeline: the *load-store unit*. The operands fetch, on the other hand, is performed in parallel with the instruction decoding, so it doesn't affect the overall cycle time. If simplifying the *issue decoder* is not enough, an idea is to add a pipeline register to the *issue logic* while paying attention to detect and resolve dependencies: a given instruction entering the pipelined issue logic must take into account all the possible dependency from the previous one.

For what concerns the *load-store unit*, its optimization is probably a bit trickier. As a starting point, the forwarding logic could be simplified by adopting some more general constraints for a value to be forwarded. This might reduce the depth of the logic performing those checks before forwarding a value from the *store buffer* to the *load buffer*.

5.3.2 Multiple Issue

Multiple issue is certainly one of the most effective techniques to improve ILP. By construction, an execution pipeline architecture implementing Tomasulo's algorithm is well prone to host multiple execution units for each type of instruction. To exploit the increased ILP that they offer, more than one instruction should be issued in each cycle, as described in chapter 2. This requires the frontend, the *issue queue*, and the *issue logic* to produce multiple instructions in a single cycle. Any modern commercial processor implements multiple issue to a certain degree. In *LEN5*, it is enough to read more the one instruction from a line of the *instruction cache* and 'teach' the *issue logic* to detect and resolve hazards between instructions being issued concurrently. Otherwise, instead of a processor with multiple issue, the result will be a processor with multiple issues. Resolving dependences between concurrently issued instructions means assigning the destination *ROB* entry index-based also on the other issuing instructions besides on the information from the register status.

In addition to this, the higher IPC must be supported also while writing the result. This may be done by instantiating more than one *CDB*. However, this would increase the complexity of the parallel access ports in the *RSs*. A solution is dividing the *RSs* in different virtual *paths*. Each *RS* can access only one *CDB*. This requires instructions showing some dependence on a previous one to be issued on the same path, so it can correctly fetch the operands. It also requires additional write ports in the *ROB*, but this is less likely to represent a problem since the *ROB* is not a complex structure.

5.3.3 Unaligned memory address

The support for unaligned memory accesses is becoming less and less used in general-purpose processors. However, in some application-specific domain, having the possibility to access any byte or set of bytes in memory without having to introduce dedicated software routines might lead to significant performance improvements. Also, sometimes some legacy code must be executed. In both cases, a solution to introduce hardware support for unaligned memory operation in *LEN5* is to issue two different *loads* or *stores* when an unaligned instruction of the same time is detected at issue time. In the case of a *load*, the two instructions operate on two different portions of a single destination register, meaning that the *RF* must support per-byte write enables.

5.3.4 Full support for *m-mode* and *s-mode*

Even if *LEN5* fully supports VM, not all the requirements for the support of a proper OS are provided in this first version. Privilege modes require special instructions to be supported by the *commit logic*, and additional memory protection mechanisms and I/O handling features must be added to both the execution pipeline and the memory system.

5.3.5 Other ISA Extensions

As mentioned before, one of the main advantages of the *LEN5* execution pipeline is the possibility to easily instantiate additional execution units, with the only requirement being to support the index-based, AXI-like handshake protocol described in detail in section 3.6.3. This offers full support for most of the frozen or draft RISC-V extensions. RISC-V also permits *fully custom* instructions to be executed, meaning that dedicated accelerators can be inserted in the execution pipeline, like matrix and vector operators with a dedicated Direct Memory Access Controller (DMAC) and *array functional units*. Also, complex arithmetic functions could be supported in hardware in high-performance scientific applications. All in all, this places virtually no boundaries on the evolution of this simple processor, with this being an absolute killer requirement in university and research.

Appendix A

Code

A.1 General code organization

The code of the execution pipeline of *LEN5* is organised following about the same organization of chapter 3. The components can be found in the directories listed below (relative to the repository root directory).

- `src/expipe/`: contains the main components divided in subdirectories dedicated to each unit.

- `src/util/`: contains some common components like the 2-way arbiters and the entry selectors.

The `include/mmm_pkg.sv` file, that is the top level configuration package, contains some system level parameters, some of which set some constants for execution pipeline, as the depth or each data structure (the *ROB*, the *issue queue* and the *RSs*). In addition, the following ones are used in the execution pipeline:

- `include/expipe_pkg.sv` contains the configuration switches and most of the type definitions and local parameters that are used in the execution pipeline modules.

- `include/control_pkg.sv` contains the instruction definitions for the issue and commit decode logic, in accordance with the RISC-V specification [17] and [16]. If an extension is to be added to *LEN5*, its instruction definitions must be inserted here. Then, the `issue_decoder`, `commit_decoder` and the `misa` CSR in `csr.sv` must be modified accordingly.

All the code has been carefully commented, explaining what is being described and recalling some of the concepts from chapter 2 and the details from chapter 3.

A.1.1 Switches

Two switches are defined in the `expipe_pkg.sv` file.

`ENABLE_AGE_BASED_SELECTOR` As mentioned in section 3.1.4, if this macro is not defined (i.e. commented), a simple priority encoder is instantiated in each RS to select the next instruction to be executed among the valid ones. Otherwise, the RSs will instantiate an execution selector that chooses the next instruction to be executed based on its age, that is the number of instructions of the same type that have been inserted in the RS after it. The oldest instruction has the higher priority. A dedicated counter is inserted in each entry of the RS. The selector is described in a behavioural way, and may enter the critical path slowing down the processor. On the other hand, it could reduce the latency after which an instruction is completed and committed. This might result in slightly higher IPC since the *ROB* is less likely to fill completely. Due to the higher power consumption of this second solution, the switch is disabled by default.

`ENABLE_STORE_PRIO_2WAY_ARBITER` If this macro is defined, priority encoders are used to select the *store buffer* over the *load buffer* when both make a request to a shared unit (the *virtual address adder*, the *data cache* or the *DTLB*). Otherwise, the "fair" arbiter described in section 3.1.3 is instantiated in the *load-store unit*. Since store-to-load forwarding is implemented (see section 3.7), giving priority to *store* instructions can reduce the number of misses during the forwarding attempts, reducing the number of *data cache* and *DTLB* accesses as a consequence. The only downside is a slightly higher latency in the execution of load instructions, but this may be compensated by the forwarding mechanism. By default, the switch is enabled so that priority encoders prioritizing *store* instructions are used.

A.2 Data structure control

In section 3.1.2 the use of Moore FSM was discussed in relation to the complex data structures that are found in the execution pipeline of the core. As stated there, the Moore FSM control design paradigm was discarded in favour of a data-driven control mechanism, where the state is represented by the status of the data structure entries. This status is strictly updated **synchronously**, and the outputs are generated as a function of this status information. Here, the coding paradigm employed to describe the control logic of these component is explained. Once again, let us consider the *load buffer* as an example. It communicates using the *valid-ready* handshake mechanism with the *issue logic*, the *virtual address adder*, the *TLB*, the *data cache* and the *CDB*. The *load buffer* can send a **request** to a unit or process its **answer**. In both cases, the handshake is related to the operation control of the

component. For this reason, handshake and operation control are most of the time included in the same control logic block.

The following paragraphs describe how requests and answers control logic is described.

Requests A request to a unit is implemented asserting the *valid* signal to that unit when the selected entry is valid and ready for that specific operation. All these informations are read from the status bit data structure, so they don't depend on the component inputs. In other words, the output *valid* signals of the component are updated based on synchronous information. As an example, this happens when one entry of the *load buffer* has fetched the operand `rs1` that is required to compute the virtual address of the source memory location in the *virtual address adder*. At this point, the status bits of the entry are checked by a dedicated selector. If they meet the requirements, the *valid* signal to the *virtual address adder* is asserted while the operands of the selected entry are presented at its input. If the ready signal of the *virtual address adder* is asserted a request to modify its status bits is generated and will take effect at the next clock edge, when the entry will be moved to a 'busy' state until the *virtual address adder* will produce the result (i.e. will answer the *load buffer* request). Under this perspective, the *load buffer* can be seen as a FSM where **the state is encoded in the status bit of its data structure**. As a matter of fact the synchronous data structure is what brakes the combinational path between the input and outputs of the *load buffer*. This is true for both data and handshake control signals. This FSM evolution is done directly programming the status information of each entry of the queue. No mealy connection is created, and the only difference with a traditional Moore FSM resides in the fact that each state is not explicitly encoded. Notice that doing this would require an encoding expressive enough to include every possible combinations of status bits in the structure. As a result, most of the status bits would be doubled if the traditional *data path* and *control unit* was adopted. However, the downside of this alternative descriptions of the units is that designing and debugging is much harder than with explicit Moore FSMs. For this reason some of the checks that control the status update, which corresponds to the state progression of an FSM, are redundant. The most evident example of redundancy is that the valid bit of the entry is checked both by the selector and by the control logic before the request is sent to the destination unit. This is shown in the example code in listing A.1.

```
// REQUEST TO THE VIRTUAL ADDRESS ADDER
// The selected entry must be valid (other checks are
   performed by the selector)
if (lb_data[vadder_req_idx].valid && vadder_idx_valid) begin
    vadder_valid_o  = 1'b1;
```

```
    if (vadder_ready_i) lb_vadder_req = 1'b1; // The adder
       accepted the request, and the entry is marked as busy
end
```

Listing A.1: Control logic description example for a *request* to the *virtual address adder*.

Answers In the chosen example, this is the case where a unit asserts its *valid* signal for the *load buffer*, meaning it has valid output data to be stored in the *load buffer*. If the *load buffer* can accept the incoming request, its *ready* signal to that unit must be asserted. A fist observation is that the availability of the *load buffer* to answer a request from a unit is independent on the information contained in that specific request. Instead, it depends only on the status of the *load buffer*. As an example, a new *load* that is issued from the *issue queue* can be accepted by the *load buffer* if at least one empty entry exists in the *load buffer* data structure. Therefore, the *ready* signal for the *issue logic* is simply given by the opposite of the *valid* bit of the entry where the new instruction will be allocated. In this way the selected new entry is overwritten only if the previous contained instruction has already completed its execution, that means the entry no longer contains useful information. In other cases, the *load buffer* must always be ready to accept incoming requests, as when accepting the virtual address computed by the *virtual address adder*. In both cases the *ready* signal does not depend directly on the input signals of the *load buffer*, meaning that the *ready* generation is not a Mealy process. This is true for every single data structure in the execution pipeline: *issue queue*, *RSs*, *load buffer*, *store buffer* and *ROB*. The input *valid* signal from the source unit determines whether the associated operation must be performed or not, such as inserting a new instruction in the assigned entry of the *load buffer* in the previous example. A piece of code describing such behaviour is shown in listing A.2.

```
// INSERT NEW INSTRUCTION
// Insert a new instruction in the queue if the selected
   entry is empty (i.e. not valid) and the decoder is
   sending a valid instruction to the load buffer
if (!lb_data[new_idx].valid && new_idx_valid) begin
    issue_logic_ready_o = 1'b1;
    if (issue_logic_valid_i) lb_insert = 1'b1;
end
```

Listing A.2: Control logic description example of the insertion of a new instrruction.

In both cases, a second synchronous process takes care of updating the data structure according to the output produced by the control logic. Using the *load*

buffer example once again, the operation control signal produced by the control logic in listing A.1 and listing A.2 is used by the synchronous update logic in listing A.3 to move the entry to a *busy* state when the virtual address computation request has been sent to the *virtual address adder*.

```
// REQUEST TO THE VIRTUAL ADDRESS ADDER UPDATE
if (lb_vadder_req) begin
    lb_data[vadder_req_idx].busy    <= 1'b1;
end
...
// ANSWER FROM THE VIRTUAL ADDRESS ADDER UPDATE
if (lb_vadder_ans) begin
    // Exception handling
    case(vadder_except_i)
        VADDER_ALIGN_EXCEPT: begin
            lb_data[vadder_idx_i].busy          <= 1'b0;
            lb_data[vadder_idx_i].except_raised <= 1'b1;
            lb_data[vadder_idx_i].except_code   <=
               E_LD_ADDR_MISALIGNED;
            lb_data[vadder_idx_i].ld_value      <=
               vadder_vaddr_i;
            lb_data[vadder_idx_i].completed     <= 1'b1;
        end
        ...
```

Listing A.3: Status update after the virtual address computation request and answer.

A.3 Assertions

One of the main advantages of System Verilog over older hardware description languages is the possibility to verify at runtime some user-defined properties that a certain module should have. These properties are called *assertions* and can be set to end the simulation with an error when violated or to throw some warning message to the user. In the execution pipeline of *LEN5* assertions are widely used to verify key properties of the control logic of the different data structures. As an example, an assertion in the *RSs* checks whether an instruction is executed (i.e. sent to the functional unit) before having its operands ready. If this happens, a warning is thrown to the simulation console or log.

Assertion are also exploited to communicate the user when if a certain data structure reached its maximum capacity, possibly slowing down the pipeline. As an example, a warning is thrown each time the *issue queue* is full. In this way the user can decide to increase its depth or modify the execution pipeline accordingly.

listing A.4 reports the code of an assertion from the *issue queue*. The ‘`ifndef` compiler directive uses the environmental variable `SYNTHESIS` usually set by the synthesis tool to prevent it from trying to synthesize the assertion construct, that would obviously result in an error.

```
//----------------------\\
//----- ASSERTION -----\\
//----------------------\\
`ifndef SYNTHESIS
always @(negedge clk_i) begin
    // Notice when the issue queue is full
    assert (fifo_full !== 'b1) else $warning("The issue
       queue is full. You might want to increase its size.");
end
`endif
```

Listing A.4: Assertion example in the *issue queue*.

Bibliography

[1] Ricardo Alves, Alberto Ros, David Black-Schaffer, and Stefanos Kaxiras. "Filter Caching for Free: The Untapped Potential of the Store-buffer". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. New York, NY, USA: ACM, 2019, pp. 436–448. ISBN: 978-1-4503-6669-4. DOI: 10.1145/3307650.3322269 (cit. on pp. 73, 76, 83, 86).

[2] *AMBA AXI and ACE Protocol Specification*. ARM IHI 0022D. ARM. Oct. 2011 (cit. on p. 25).

[3] Marco Andorno. *Design of the frontend for LEN5, a RISC-V Out-of-Order processor*. Dec. 2019 (cit. on pp. 1–3, 17, 39, 59).

[4] RISC-V Foundation. *RISC-V: The Free and Open RISC Instruction Set Architecture*. 2019. URL: https://riscv.org/ (visited on Nov. 14, 2019) (cit. on p. 2).

[5] The Apache Software Foundation. *Apache License*. Version 2.0. Jan. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0 (visited on Nov. 27, 2019) (cit. on p. 1).

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 9780128119051 (cit. on pp. i, 18).

[7] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. ISSN. Elsevier Science, 2017. ISBN: 9780128122761 (cit. on p. 5).

[8] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN: 9780999249109 (cit. on pp. 11, 42, 60, 66, 94).

[9] David A. Patterson. "Latency Lags Bandwith". In: *Commun. ACM* 47.10 (Oct. 2004), pp. 71–75. ISSN: 0001-0782. DOI: 10.1145/1022594.1022596 (cit. on p. 6).

[10] Matteo Perotti. *Design of an OS compliant memory system for LEN5, a RISC-V Out-of-Order processor*. Dec. 2019 (cit. on pp. 1, 3, 42, 71, 82).

[11] André Seznec and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction". In: *J. Instruction-Level Parallelism* 8 (2006) (cit. on p. 8).

[12] SolderPad. *Solderpad Hardware Licence*. Version 2.0. 2018. URL: https://solderpad.org/licenses/SHL-2.0/ (visited on Nov. 27, 2019) (cit. on p. 1).

[13] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 25–33. DOI: 10.1147/rd.111.0025 (cit. on p. 12).

[14] UMC, ed. *UMK65LSCLLMVBBR_B. UMC 65nm Low-K Multi-Voltage Low Leakage RVT Tapless Standard Cell Library Databook*. Version B03. UMC. Jan. 2014 (cit. on pp. 99, 101).

[15] David W. Wall. "Limits of Instruction-level Parallelism". In: *SIGARCH Comput. Archit. News* 19.2 (Apr. 1991), pp. 176–188. ISSN: 0163-5964. DOI: 10.1145/106975.106991 (cit. on p. 7).

[16] Andrew Waterman and Krste Asanovic, eds. *The RISC-V Instruction Set Manual. Volume II: Privileged Architecture*. Version 20190608-Priv-MSU- Ratified. RISC-V Foundation. June 2019 (cit. on pp. 42, 66, 71, 94, 95, 121).

[17] Andrew Waterman and Krste Asanovic, eds. *The RISC-V Instruction Set Manual. Volume I: User-Level ISA*. Version 20190608-Base-Ratified. RISC-V Foundation. Mar. 2019 (cit. on pp. 45, 49, 53, 55, 60, 121).

Acronyms

Generic

AXI Advanced eXtensible Interface

CAM Content Access Memory

FF Flip-Flop

FIFO First In First Out

FSM Finite State Machine

HPC High Performance Computing

LSB Least Significant Bit

MSB Most Significant Bit

MUX multiplexer

RAW Read After Write

WAR Write After Read

WAW Write After Write

Computer architecture

ASID Address Space Identifier

BTB Branch Target Address

CPU Central Processing Unit

CSR Control Status Register

DMAC Direct Memory Access Controller

FPU Floating-Point Unit

GPU Graphics Processing Unit

ILP Instruction Level Parallelism

IPC Instructions Per Cycle

ISA Instruction Set Architecture

MMU Memory Management Unit

OoO Out of Order

OoOE Out of Order Execution

OS Operating System

PC Program Counter

PPN Physical Page Number

RISC Reduced Instruction Set Computer

SIMD Single Instruction Multiple Data

SMT Simultaneus Multi-Threading

VM Virtual Memory

VPN Virtual Page Number

LEN5 components

ALU Arithmetic and Logic Unit

CDB Common Data Bus

DTLB Data Translation Lookaside Buffer

FPU Floating-Point Unit

MMU Memory Management Unit

ROB Reorder Buffer

RF Register File

RS Reservation Station

TLB Translation Lookaside Buffer