POLITECNICO
DI TORINO

Master's Degree in Electronic Engineering

Master Thesis

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

Supervisor
Prof. Maurizio Martina

Candidate
Marco Andorno

Academic year 2018-2019

Abstract

RISC-V is a free and open source Instruction Set Architecture, which has sparked interest all over the community of computer architects, as it paves the way for a previously unseen era of extensible software and hardware design freedom. One of its main strength points is the vast modularity implemented in terms of different ISA extensions, which aim to cover a very broad range of applications. This allows designers to tailor the architecture according to their specific needs, without constraining them to support unnecessary instructions.

Being RISC-V a relatively new ISA, a limited number of cores is available at the moment, and in particular very few of them are open sourced. So the main motivation for this work is the contribution to this open source hardware community, by means of the design of an Out-of-Order RISC-V core as general purpose as possible.

The core is a 64-bit processor, supporting the G extension, which is a shorthand for the base integer (I), multiply and divide (M), floating point (F) and atomic (A) extensions. One goal of this project, which will be carried out alongside two colleagues, is to eventually include support also for the operating system, by implementing the yet unstandardized Privileged ISA, for the experimental vector extension (V) and possibly for a matrix extension to be defined from scratch. These last design choices are motivated by the lack of open source cores supporting them, and the great advantage that such vectorized computation can provide in a world where the popularity and the performance needs of artificial intelligence and machine learning are ever-growing.

Moreover, the choice of designing an out-of-order core arises mainly as all modern processors are of such kind, as it has been the best compromise to efficiently exploit instruction level parallelism for decades. The goal is to implement both instruction issue and execution to be performed Out-of-Order, because this allows the highest performance gain. This design choice, of course, comes with a series of implications that will need accurate analysis and bench- marking, possibly by keeping everything as parametric and modular as possible: branch prediction, instruction queue management, memory hierarchy and cache organization are just some examples.

The final outcome of this work will be an in-depth exploration of the design space offered by such complex architectures, to actually experience firsthand the main issues and tradeoffs designers must face and to be prepared to offer a significant contribution to the state of the art of processor design. Moreover, the common hope is for this project to serve as the basis for future in-house development of a complete RISC-V-based platform here at Politecnico di Torino. As mentioned above, the entire work will be open source and available on a GitHub repository.

Acknowledgements

Thanks everybody!

Contents

List of Tables

List of Figures

List of Acronyms

ALU Arithmetic-Logic Unit

ASIC Application Specific Integrated Circuit

BHT Branch History Table

CDB Common Data Bus

CPI Clocks Per Instruction

CPU Central Processing Unit

FPGA Field Programmable Gate Array

FPU Floating Point Unit

IC Instruction Count

ILP Instruction-Level Parallelism

IPC Instructions Per Clock

ISA Instruction Set Architecture

PC Program Counter

PHT Pattern History Table

RAW Read-After-Write

ROB Reorder Buffer

TAGE Tagged Geometric

VLIW Very Long Instruction Word

WAR Write-After-Read

WAW Write-After-Write

WOE Window Of Execution

Chapter 1

Introduction

Since their first development in the 1960s, *out-of-order* (also known as *dynamic scheduling*) microprocessors have become the main architectural paradigm used in high-performance CPUs, given their ability to hide pipeline latencies and allow for a faster program execution. Along with that, another key role in achieving high effective performance is played by the concept of *speculation* and in particular by branch prediction techniques, which improve the pipeline throughput by maintaining a constant instruction flow inside the processor.

Nowadays, almost every device of common use, from desktop computers, to laptops, to smartphones and tablets, contains some kind of out-of-order core which exploits such techniques to offer the computing power and pleasant user experience that the modern world demands. Of course, these architectural design choices come with the drawback of significant added hardware complexity, so there are still some very low power or very low cost microprocessors which do not employ them.

In order to deeply understand such complex architectures and explore the design choices that must be faced in order to achieve that final result, a very convenient way is to make use of an open-source Instruction Set Architecture (ISA), namely RISC-V, which in turn allows the design of open source hardware.

This is exactly the aim of this thesis work: to design a RISC-V core, featuring out-of-order execution and speculation to face the issues that such a project involves firsthand, and gain valuable experience in this field of computer architectures. Given its complexity, this work has been carried out by the candidate along with two other colleagues, each one developing a defined part of the core, to come up with the complete design. It is common hope for this project to also serve as the starting point for the future development of a RISC-V based platform at Politecnico di Torino, which could be used for a many different research purposes. For this reason, the entire design and its documentation will be available on a GitHub repository.

1.1 The RISC-V ISA

RISC-V started as a summer research project in 2010 at UC Berkeley by PhD candidates Andrew Waterman and Yunsup Lee and professors Krste Asanović and David Patterson, but soon developed into a fully featured ISA, presented several years later in Waterman's dissertation [1].

Why a new ISA from scratch?

Today the goal of RISC-V is to become a universal ISA [2], able to suit all kinds of processors, from small embedded ones to high-performance cores, from single issue in-order to superscalar out-of-order microarchitectures. Moreover, it is also designed to be implementation independent, in order to work on FPGAs, ASICs and even future technologies, and to be compatible with a large number of popular softwares and programming languages.

How RISC-V intends to achieve that is by leveraging its two main strengths: first of all it is a completely *open source* ISA, meaning that no single company has control over its development and future, and secondly it is *modular*, in the sense that the base instructions are frozen and will stay the same, while new extensions are available and will be developed to expand the capabilities of the ISA (see section 1.1.1).

RISC-V belongs to a non-profit foundation, composed by many different corporate members as well as other non-profits and academic institutions, which together aim at maintaining the stability of the ISA, evolving it when necessary and trying to make it ever more popular. For more information, refer to https://riscv.org/.

Add other advantages of the ISA (see Reader chapter 1)

1.1.1 Extensions

Most ISAs are *incremental*, meaning that, in order to ensure compatibility, every new processor must implement new ISA extensions as well as all the extensions introduced in the past, which leads to an accumulation of very rarely used instructions and a subsequent waste of hardware complexity and area. A clear example of this inflation is the growth of the number of instructions in the x86 ISA (figure 1.1).

One the other hand, as stated above, RISC-V is a *modular* ISA: a small number of base instructions (called RV32I, RV64I or RV128I for 32, 64 and 128-bit processors respectively) must be implemented by all instances of RISC-V processors and are guaranteed to never change in the future, while on top of that, designers can freely choose to include support or not for each of the other optional extensions, some of which have already been frozen, while others are still in development. Table 1.1 contains a list of available extensions at the time of writing.

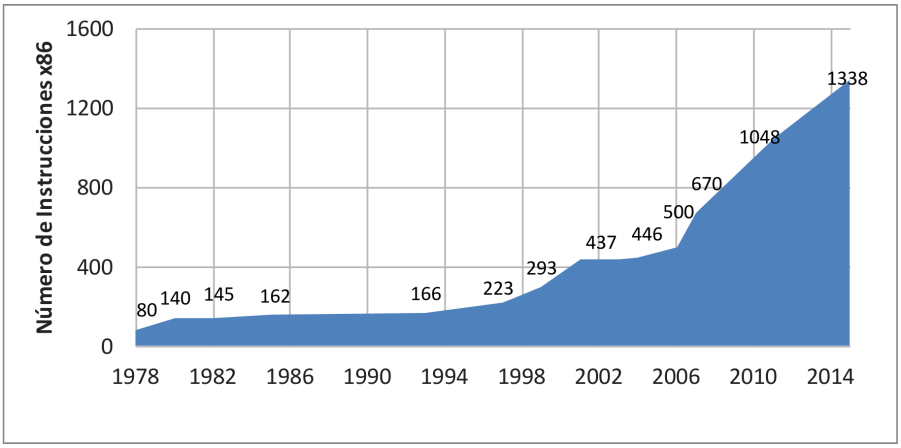

Figure 1.1. x86 instruction count over time [2, p. 3]

Name	Description
I	Base integer instruction set, including arithmetic and logic instructions, jump, branch and control transfer instructions and some miscellaneous general management ones.
M	Integer multiplication and division extension.
A	Atomic extension for atomic memory operations, for process synchronization.
F	Single-precision floating point extension.
D	Double-precision floating point extension.
G	Shorthand for all the previous ones. LEN5 supports the RV64G ISA.
Q	Quad-precision floating point extension.
L	Decimal floating point extension.
C	Compressed instructions extension.
B	Bit manipulation extension.
J	Dynamically translated languages extension.
T	Transactional memory extension.
P	Packed-SIMD extension.
V	Vector extension.
N	User-level interrupts extension.
H	Hypervisor extension.

Table 1.1. RISC-V ISA extensions [2]

1.1.2 Comparison with other ISAs

Arguably the two most popular ISAs at the present time are Intel x86 and ARM, which are dominant in the desktop/laptop computers and smartphones/tablets markets respectively. The first significant difference between them and RISC-V is that they are *proprietary* ISAs, which means that whoever wants to design a processor based on such instruction sets is obliged to the payment of the required royalties. On the other hand, RISC-V is free for everyone.

For what concerns the microarchitectural standpoint, another major difference resides in the organization of the internal registers. First of all, RISC-V has 32 of them, twice as much as ARM has, and four times as much as x86. A higher number of registers greatly simplifies assembly language programming and compiler writing. Moreover, the first of those registers, register `x0`, is hardwired to zero, which allows for a significant reduction in instruction count, as many instructions present in other ISAs, which do not have a zero register, can be synthesized using RV instructions with `x0` as an operand. As an example, RISC-V does not need a separate instruction in order to branch if the value of a register is zero: this operation can be obtained with the `beq` (branch if equal) instruction using `x0` as the second operand. The Program Counter (PC) in the RISC-V ISA is a separate register, and that prevents any instruction from being able to modify it and thus become a branch instruction, as is the case of the ARM ISA, reducing the complexity of the branch prediction hardware and avoiding the loss of one general purpose register.

By keeping simplicity in mind, RISC-V does not provide direct support for byte or half-word integer computation, which can be carried out using separate shift instructions, as they are not critical in terms of efficiency and energy consumption, as are for instance reduced-size memory accesses [2, p. 20]. In addition, multiplication and division are not present in the base ISA (they are comprised in the M extension), and that means that a full software stack can run even without them, which helps reduce the size of embedded chips where such operations are not needed.

Other instructions that the designers of RISC-V chose not to include are, among others, stack instructions, as the stack pointer is one of the general purpose registers and so is accessed as any other register, delayed load, as it is deemed as useless in modern deeply pipelined processors, and finally delayed branch and condition code instructions, which complicate the dependencies checking in out-of-order processors [2, p. 21].

It is quite clear that who conceived the RISC-V ISA adopted a philosophy of keeping it simple and that *less is more*, by targeted choices made by learning from the work achieved in the previous decades.

Chapter 2

State-of-the-art processor architectures

The performance of a processor is defined by the the time it takes to execute a program. This time span, called *CPU time*, can be expressed as:

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Clock cycles}}{\text{Program}} \cdot T_{ck}$$

where T_{ck} is the clock period.

The first term can be decomposed further by computing the total number of instructions inside a program, called *Instruction Count* (IC), which is known given the assembly code of the program. From this figure and the total number of clock cycles, the average number of *Clocks Per Instruction* (CPI)[1] can be derived. By factoring in these quantities, the final expression of CPU time is as follows [3, p. 53]:

$$\text{CPU time} = \text{IC} \cdot \text{CPI} \cdot T_{ck} \tag{2.1}$$

Equation (2.1) shows that the processor performance is directly and equally dependent on three factors:

- Clock period, which depends mainly on the implementation technology and the microarchitectural choices (e.g. pipeline depth).
- IC, which is determined for the most part by the ISA (see section 1.1.2) and compiler technology.
- CPI, which is dependant on both the ISA and the architecture.

[1] Sometimes, also the inverse figure can be used, that is *Instructions Per Clock* (IPC).

The goal is then to minimize each of these terms, but it is evident that none of these parameters can be modified without affecting the others, as many design choices influence many of them.

2.1 Instruction-level parallelism

Earliest processors executed instructions one at a time, fetching a new one only after the previous has finished, leading to a number of clock cycles per instruction greater than one, and in particular equal to the number of stages an instruction must get through. These processors, where CPI > 1, are called *subscalar*. To illustrate the situation, in the example of the classic 5-stage RISC pipeline (fetch, decode, execute, memory access, write back), a subscalar processor would execute three consecutive instructions as shown in figure 2.1, taking a total of 15 clock cycles.

Figure 2.1. Subscalar processor

Starting from the mid 80s, processor architects introduced *pipelining* to improve performance by overlapping the execution of different instructions. This overlap means that at any given point in time there can be multiple instructions running in different stages of the processor, that is *in parallel*, hence the term *Instruction-Level Parallelism* (ILP), which is a fundamental concept in developing techniques to enhance processor performance. For the same example of figure 2.1, a pipelined processor could theoretically achieve a CPI of 1, executing one instruction for each clock cycle (see figure 2.2). Processors of this kind are called *scalar*.

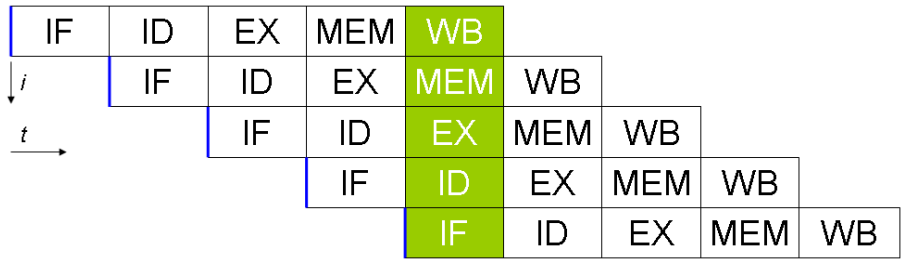

Figure 2.2. Scalar processor

In practice however, data and control dependencies between successive instructions could cause hazards and force the pipeline to stall, causing CPI to rise once again at values greater than one. There are mainly three types of hazards that can take place in a pipelined processor:

- **Structural hazards** arise when a hardware block is needed by two or more instructions at the same point in time. For instance, if a processor features only one memory block for both instructions and data, then two different instructions executing in the fetch and memory access stages could generate a structural hazard when trying to read from memory. Such hazards can either be easily solved (e.g. separate instruction and data memory in this example) or are known and accepted by the designers, given the limited hardware available.

- **Data hazards** in a simple pipelined processor occur when there is a *data dependence* between instructions, that is one instruction needs to read a value that provided by a previous instruction. For example, in

```
add     x1,x2,x3
sub     x4,x5,x1
```

  the `sub` instruction needs the value of register `x1` in the decode stage, but the previous `add` has not yet reached the write back stage and a data hazard is generated.

- **Control hazards** arise in the case of conditional flow changing instructions, such as branches, that prevent following instructions to be fetched until the new direction is resolved.

The real CPI a pipelined processor can achieve is then given by the sum of the ideal CPI and all the delays introduced by pipeline stalls caused by hazards [3, p. 168]:

$$\begin{aligned} \text{CPI} &= \text{Ideal CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls} \\ &= 1 + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls} > 1 \end{aligned} \tag{2.2}$$

Those hazards become more frequent and more expensive to manage the more pipeline stages are introduced and that is a clear example of a tradeoff between two factors of the performance equation (2.1): a deeper pipeline shortens the critical path and thus reduces the clock period, but at the same time it increases the CPI. That is the reason why designers at some point had to find other architectural solutions to improve performance.

2.1.1 Multiple-issue processors

A processor featuring a single execution pipeline can only achieve a theoretical CPI of 1, but by duplicating the pipeline to include multiple execution units more than one instruction per clock cycle could be delivered. That is the idea that lies behind *multiple-issue* processors, that exploit ILP by executing independent instructions on separate execution pipelines.

Instructions that can be issued independently to the different pipelines are selected among a so called *basic block*, that is a sequence of instructions comprised between single entry and exit points (i.e. with no branches or jumps in between). Recalling the examples of the previous section, figure 2.3 shows the execution scheme for a multiple issue processor.

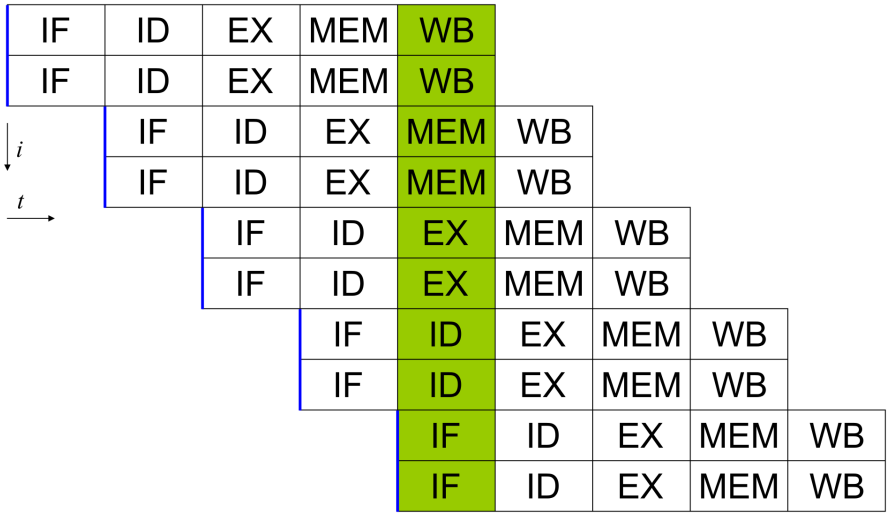

Figure 2.3. Multiple-issue processor with two pipelines

Two main different approaches exist to multiple issue processors:

- **Very Long Instruction Word (VLIW)** processors, also known as *static* multiple-issue, rely on software to discover ILP chances at compile time, thus avoiding increased hardware complexity. The compiler groups instructions that can be executed in parallel in a single long packet-like instruction (hence the name VLIW), that is then split and issued to the different execution units at run time. Despite many efforts, however, such static techniques reveal efficient only for specific applications presenting a high level of data parallelism [3, p. 168], mainly because the compiler software needs a perfect knowledge of the underlying architecture in order to efficiently exploit ILP.

- **Superscalar** processors, also known as *dynamic* multiple-issue, on the other

hand, rely on dedicated hardware to exploit ILP at run time. Instructions belonging to a basic block are inserted into a Window Of Execution (WOE), from where instruction that can run in parallel thanks to no data dependence are selected and issued to the respective following pipeline stages. This dynamic approach has been shown to work better than a static one, at the cost of a significant hardware complexity overhead.

Multiple-issue processors can achieve a CPI lower than 1 (usually expressed at this point as Instructions Per Clock (IPC), greater than 1) thanks to duplicate hardware units that also lower the impact of structural hazards, but they are nonetheless subject to data and control hazards. Instructions belonging to the same basic block are very likely to depend upon one another, as they are part of the same piece of program, and as such the amount of ILP in contiguous instructions of a basic block is usually very small, leading to a low usage of the additional pipelines, and that is the reason why allowing multiple issues is not very useful by itself, but is almost always paired with the techniques analyzed in the next section.

2.2 Dynamic scheduling

All the processors seen in the previous sections adopted a so called *static scheduling* of the pipeline, meaning that instructions are issued and executed along the pipe strictly in program order. To really extract the benefits of ILP, however, all modern high-end processor employ a *dynamically scheduled* pipeline, that can execute instructions out of order with respect to the assembled program. As an example consider the following code:

```
add    x1,x2,x3
sub    x5,x1,x4
mul    x12,x18,x19
```

In a classic 5-stage statically scheduled pipeline, instructions are executed in-order, and that means that the `mul` instruction cannot begin execution until the data dependence between `add` and `sub` is resolved by stalling the pipeline, as the execution takes place in program order. By using dynamic scheduling, on the other hand, if there are no structural hazards (and we can safely assume that that is the case, as the multiplier is likely to be a separate block from the ALU), the `mul` can be executed and maybe even completed before the `sub`. Instructions are then still issued in-order to the execution stage from the window of execution, but they can begin and complete execution out-of-order.

Dynamic scheduling is almost always used in conjunction with superscalar processors, because the advantages given by the out-of-order execution and the availability of multiple functional units go hand in hand. This combination offers several strengths compared to static scheduling or VLIW processors [3, p. 192]. For instance,

it allows compiled code to run in an efficient way on different microarchitectures, as the pipeline can manage itself and exploit ILP without needing the help of the compiler. Moreover, it can handle cases where dependencies cannot be found at compile time, such as memory operations or dynamic branches. But the most important advantage of all is that an out-of-order processor is able to mask the effect of unpredictable delays in the pipeline by executing later instructions without stalling. Remember that cache misses can easily take hundreds of clock cycles to resolve, which would turn into hundreds of wasted cycles in an in-order processor, but are instead taken advantage of to carry out unrelated tasks in an out-of-order one.

In order to do so, the WOE acts as a buffer between the fetch stages (called *frontend*) and the execution and commit stages (called *backend*), that hopefully always contains enough instructions to ensure a constant flow to the functional units, even when earlier instructions are waiting for some event. This is obviously possible only if the frontend is able to maintain a high enough bandwidth of fetched instructions to the WOE.

2.2.1 Dependencies and hazards

Out-of-order processors are subject to all the dependencies listed in section 2.1, but due to the reordering of instructions, other hazards can arise from so called *name dependencies*. In this context, a useful taxonomy to categorize such hazards is defined[2]. Let $D(i)$ be the *domain* and $R(i)$ be the *range* of instruction i, meaning respectively the registers or memory locations read and written by instruction i, and consider two instructions i and j, with j following i in the program order. Then, there are three possible kinds of data hazards:

- **Read-After-Write (RAW)** hazards are the only true data hazards arising from a data dependence and occur, as seen previously, when instruction j is trying to read a piece of data before i writes it, leading to a wrong value read by j, as in the following example:

```
add     x1,x2,x3
sub     x4,x5,x1
```

  More formally, RAW hazards occur if:

$$R(i) \cap D(j) \neq \varnothing$$

[2]Structural and control hazards are not considered here, as they are the same as in an in-order processor.

- **Write-After-Read (WAR)** hazards arise from the name dependence called *anti-dependence*, that occurs when instruction j writes the same location that i reads, causing i to read the wrong value if j is executed first, as in:

```
add    x1,x2,x3
mul    x2,x5,x6
```

  More formally, WAR hazards occur if:

$$D(i) \cap R(j) \neq \varnothing$$

- **Write-After-Write (WAW)** hazards arise from the name dependence called *output dependence*, that occurs when instructions i and j write their outputs on the same storage locations, leaving the final wrong value written by i, if j is executed first, like in the following:

```
add    x1,x2,x3
mul    x1,x5,x6
```

  More formally, WAW hazards occur if:

$$R(i) \cap R(j) \neq \varnothing$$

It is hopefully clear that WAR and WAW hazards occur only in dynamically scheduled processors where instruction order can be rearranged and that the dependencies that cause them are called name dependencies, because it is only a matter of storage location used and not an issue with the correct outcome of the program. In the examples above, if the `mul` instruction could (temporarily) write its output on a different register, until the `add` completes, then the semantics of the program would respected and the hazards resolved without stalling. That is the idea that lies behind *register renaming*, which is the technique used in out-of-order processors at the decode stage to detect and solve WAR and WAW hazards by converting the architectural registers that instructions refer to to different physical registers hidden to the programmer and compiler.

2.2.2 Scheduling techniques

Out-of-order execution needs dedicated hardware to select instructions inside the WOE and detect and prevent hazards. For this purpose, several schemes and algorithms exist, among which are *scoreboarding* and *Tomasulo's algorithm* that are described in the following.

Scoreboarding

Scoreboarding is a centralized scheduling technique first introduced in the CDC 6600 in the 60s [4] and still widely used today. The algorithm provides the following stages for each instruction after the decoding:

- **Issue**: instructions stall in this stage until there are no structural hazards and all the output dependencies with previously issued instructions are resolved, to avoid WAW hazards.
- **Read operands**: instructions can proceed when their operands are available, resolving RAW hazards, in an out-of-order fashion.
- **Execute**: operands are passed to the functional units that perform the requested operations.
- **Write result**: the write back operation is stalled until all earlier instructions that are anti-dependent have read the previous value, resolving WAR hazards.

Each of these stages can take an arbitrary number of cycles, thus, in order to control the progress of all instructions, a set of three data structures is used as shown in figure 2.4. The first one is the *instruction status* table, that keeps track of which

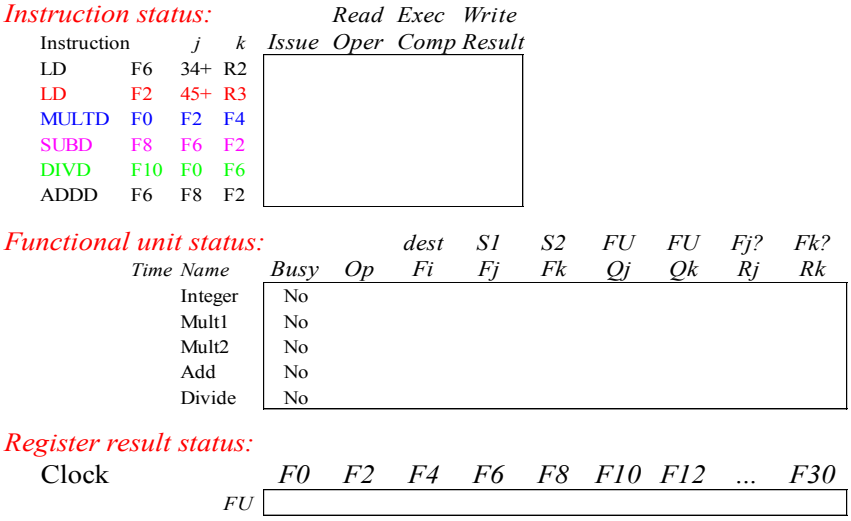

Figure 2.4. Scoreboard structure

of the four stages each instruction is currently in. Then, there is the *functional unit status* table, which has nine fields for each functional unit, indicating if that unit is busy, what operation it has to perform, the destination register, the source operands registers, the functional units that will produce the operands and two

flags indicating when those operands are ready. Finally, the *register result status* table indicates for each register which functional unit will write its result to it.

The original scoreboarding algorithm did not include register renaming and so WAW and WAR hazards could potentially cause the pipeline to stall in the issue and write result stages respectively. For this reason, register renaming can still be implemented, but it must be carried out in the issue stage, by a dedicated renaming unit, like the one shown in figure 2.5, based on the one included in the MIPS 10000. The *rename table* keeps track of the mapping between architectural and physical

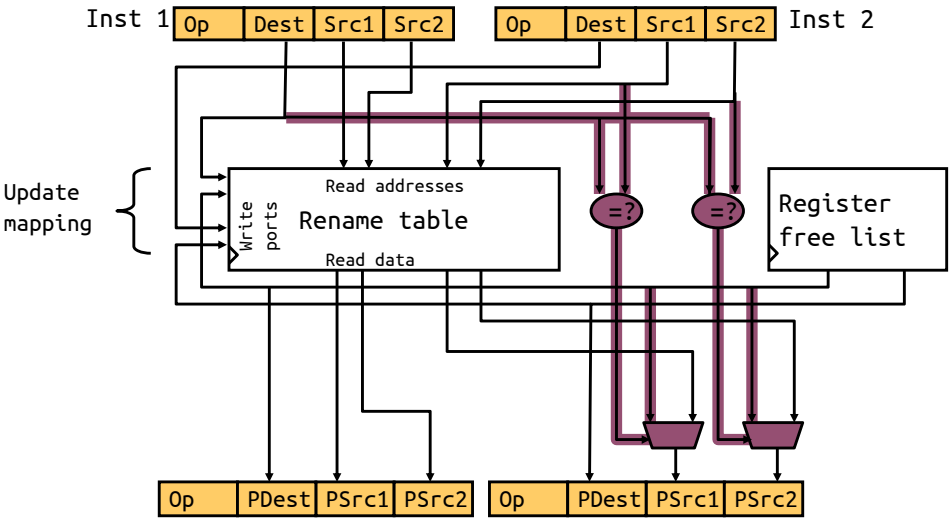

Figure 2.5. Renaming unit

registers, to maintain correct value references, while the *register free list* contains the names of all available physical registers to be used for renaming. As this unit renames two instructions in parallel, it has to check for RAW hazards between them, and in case there is one, rename the second instruction with the newly assigned physical registers to the other one.

Using this scheme, also known as *explicit* register renaming, WAW and WAR hazards are completely avoided as early as an instruction is decoded and issued, meaning that no further checks must be performed in the later stages of the algorithm.

Tomasulo's algorithm

Invented by Robert Tomasulo for the IBM 360/91 Floating Point Unit (FPU), this algorithm offers a different approach to dynamic scheduling, by adopting a *distributed* control instead of a centralized one, as present in scoreboarding. This idea

is based around the concept of *reservation stations*, which are buffers placed in front of each functional unit, including load and store units, to store instruction operands. A generic architecture based on Tomasulo's approach is shown in figure 2.6.

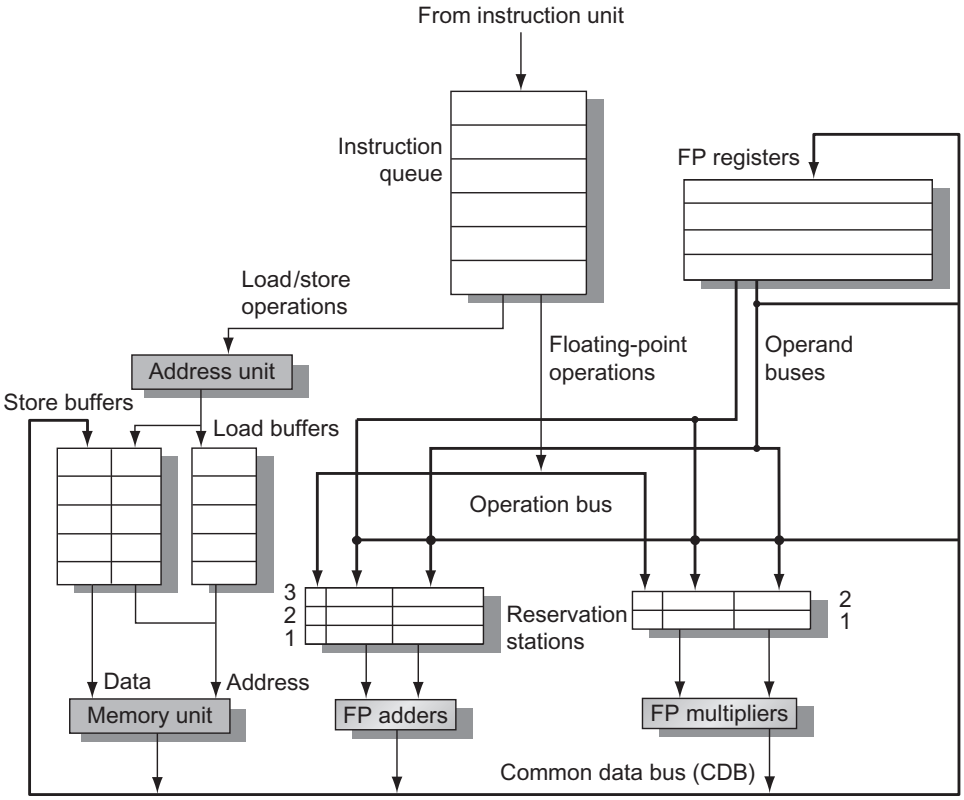

Figure 2.6. FPU example architecture using Tomasulo's algorithm [3, p. 198]

Each reservation station contains several fields providing similar information with respect to functional unit status data structure used in scoreboarding: which operation to perform, the reservation stations from which the source operands will come, the value of the operands and the busy status. The key difference with scoreboarding is, however, that the results of the functional units are broadcast to the register file as well as to all the reservation stations through a Common Data Bus (CDB), while the scoreboarding technique only writes results to registers. This, in turn, provides the great advantage of allowing *implicit* register renaming at each reservation station, because register names are discarded when an instruction is issued to a reservation station, as operands will come from another reservation station of from the CDB as soon as they become available. Moreover, if multiple instructions are waiting on the same result, they can all be started simultaneously when such result arrives because of the presence of multiple reservation stations,

while on the other hand, using scoreboarding, they would wait in turn for the register file bus to be free, possibly wasting clock cycles [3, p. 201].

In the end, the steps that each instruction must get through are similar to the one in scoreboarding, but the actions performed are different:

- **Issue**: instructions are fetched in-order from the issue queue stall in this stage until there is a matching reservation station available (no structural hazards) and then are issued to the reservation station with implicit renaming.
- **Execute**: the CDB is monitored until all operands are available (avoid RAW hazards), at which point the functional unit executes the instruction.
- **Write result**: as soon as an operation completes, the result is written on the CDB and from there to the register file and reservation stations.

Tomasulo's algorithm is today used in many high-performance processors and it has been chosen also for the design of LEN5.

2.3 Hardware-based speculation

For typical ISAs around 10–20% of instructions are branches, meaning that an average basic block will not contain more than 5 to 10 instructions. This is obviously a significant constraint, as the amount of ILP that can be exploited in such a small set of instructions without incurring in control dependencies is quite limited. From these reason the idea of *hardware-based speculation* was born, based on three principles [3, p. 208]:

- Dynamic branch prediction, to fetch and issue instructions before the outcome of a branch is determined (refer to section 3.2 for details).
- Speculative execution, to allow the execution of such instructions even if their control dependencies are not resolved yet.
- Dynamic scheduling, to schedule instructions crossing the boundary of a single basic block.

This represents an important improvement over mere dynamic scheduling and even branch prediction alone, because this way instructions are executed as if the guesses on the taken direction were always correct, leading to a *data flow execution* of the program, where operations are executed as soon as their operands are available, irrespective of control flow.

Of course, some precautions must be taken in order to handle the situation when the speculated flow was predicted incorrectly, as to restore the original state of the processor and proceed down the right path. For this reason, an additional stage after the write result must be inserted in order to decouple the production of a

result by a functional unit and the actual irreversible update of the register file and data memory, that can take place only when an instruction is no longer speculative. This last step is called *instruction commit* and must always be performed in-order.

2.3.1 Reorder Buffer

In both scoreboarding and Tomasulo's approach, instruction commit can be handled using a dedicated hardware structure called Reorder Buffer (ROB). As the name implies, the ROB acts as a buffer between the functional unit outputs and the register file and memory, storing speculative results until the speculation is resolved and thus effectively increasing the number of registers available, similarly to reservation stations. A general scheme of an architecture using the ROB is shown in figure 2.7, highlighting which parts of the pipeline are in-order and which out-of-order.

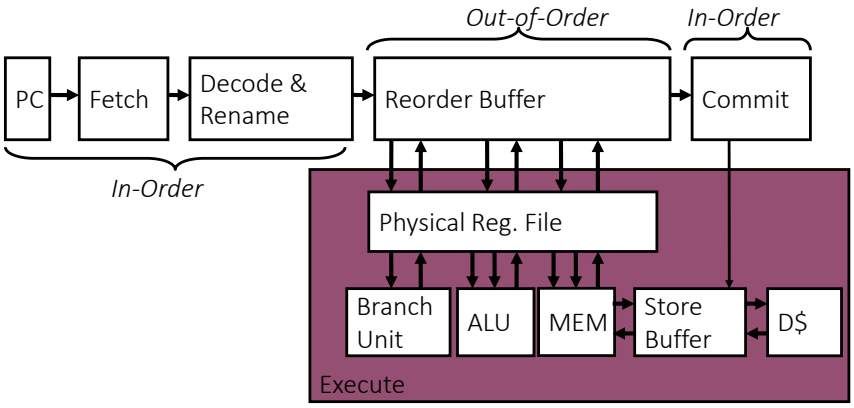

Figure 2.7. An out-of-order pipeline featuring register renaming and ROB

ROB with Tomasulo's algorithm

As stated above, the ROB extends the number of available registers and thus provides renaming on its own by substituting the register file before instruction commit. Moreover, alongside the reservation stations and the CDB, the ROB serves as another source of operands, as figure 2.8 shows. Finally, for its intrinsic nature, the ROB also serves almost the same purpose of the store buffer as a reservation station before the data memory.

In terms of algorithm steps, what changes is that during execution the results are broadcast on the CDB and to the ROB instead of to the register file. In addition, during the added step of instruction commit, an instruction is removed when it reaches the head of the ROB (which acts as a circular buffer) and, if it is a branch

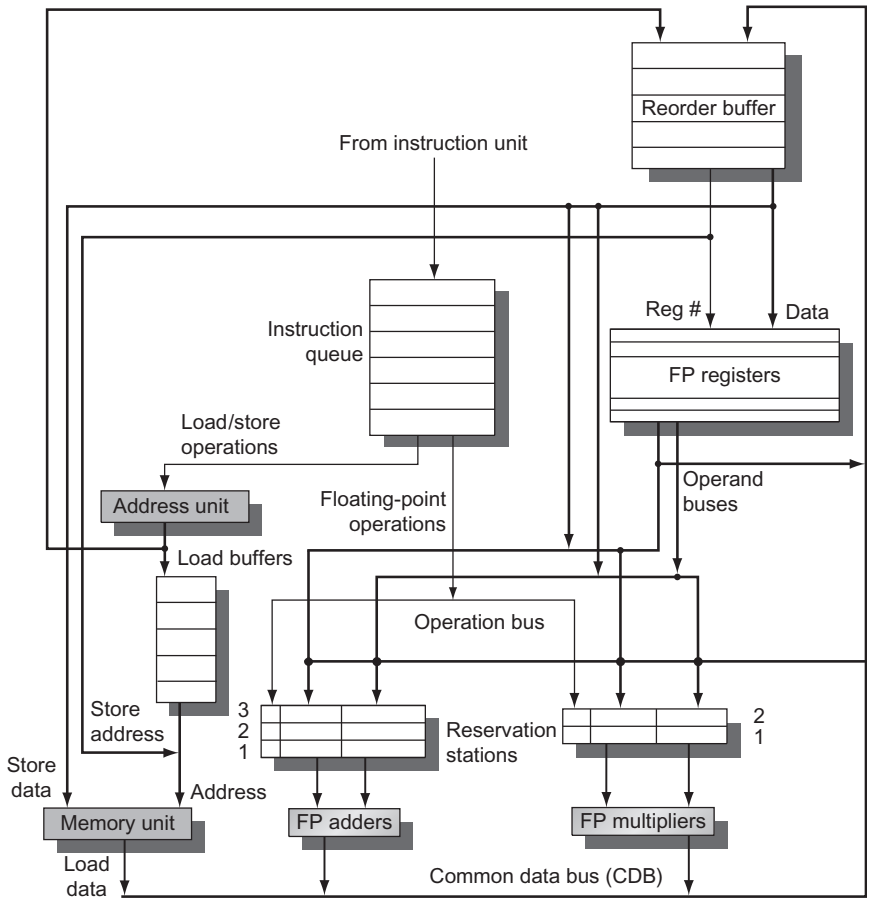

Figure 2.8. FPU example architecture using Tomasulo's algorithm and ROB [3, p. 210]

with a wrong prediction to reach the commit stage, then the ROB gets flushed and execution resumes at the correct target of the branch.

2.4 Summary of ILP techniques

To summarize this overview of multiple-issue processors and techniques to exploit ILP, table 2.1 provides all the important differences at a glance.

Name	Issue structure	Hazard detection	Scheduling	Relevant features	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Embedded MIPS and ARM cores
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Out-of-order execution, but no speculation	None
Superscalar (speculative)	Dynamic	Hardware	Dynamic	Out-of-order execution and speculation	Intel Core i3, i5, i7, AMD Ryzen
VLIW	Static	Software	Static	Hazards detected by the compiler	Signal processors like the TI C6x

Table 2.1. Summary table of multiple-issue approaches [3, p. 219]

Ariane e BOOM??

Chapter 3

Branch prediction techniques

As stated throughout the previous chapter, as well as in equation (2.2), in order to avoid stalls caused by control hazards and to ensure a steady flow of instructions to issue to the execution stages, one of the most important features of almost any modern high-performance processor is branch prediction. As seen for the multiple-issue paradigm, branch prediction too can be implemented at compile time or at run time, leading to two separate families of techniques, known as *static* and *dynamic* branch prediction.

The overall performance of a certain branch prediction technique can be essentially traced back to two factors:

- **Accuracy**: that is the percentage of correctly predicted branch instructions. This figure depends only on the type of branch predictor used.
- **Misprediction penalty**: the CPU time lost in executing wrong path instructions in case of an incorrect prediction. This parameter is determined by the architecture of the processor, and not by the branch predictor.

3.1 Static branch prediction

In static branch prediction, the action to be taken for each branch instruction is determined solely by the compiler and is then fixed at execution time. A number of static techniques exists [6]:

- Predict always taken (not taken): the accuracy of this method depends greatly on the *program sensitivity* to the number of taken (not taken) branches, so results may vary according to the algorithm, the programmer and the compiler.
- Predict branches with certain opcodes as taken or not taken: as in [6], this technique gives better results than the previous one but only if the predictions are tailored to the benchmark algorithm.

- Predict backward branches (to lower PCs) as taken and forward branches as not taken: this strategy exploits the fact that in loops that iterate a large number of times, the condition is checked at the end of teh body and so produces a backward branch if true. This can however introduce some delay in computing if the target of the branch is higher or lower than the current PC.

- Delayed branch [7]: this is not actually a prediction technique, but a way to reduce the branch penalty, with the compiler scheduling (usually) one instruction that would be execute regardless of the branch outcome in a so called *delay slot* which masks the delay of a single cycle stall in simple pipelines. In more complex out-of-order pipelines, however, this can have harmful side effects, as mentioned in section 1.1.2.

Static prediction techniques have the advantage of adding no hardware complexity whatsoever by relying only on software technology, but on the other hand struggle to achieve satisfying accuracy. For this reason, nowadays, they are used only in application specific processors or as an assist to more complex and performing dynamic techniques.

3.2 Dynamic branch prediction

Dynamic strategies rely on the other hand on dedicated hardware to make predictions based on the actual run-time past behavior of branches. The general scheme of a dynamic branch predictor is shown in figure 3.1. An event source, that is

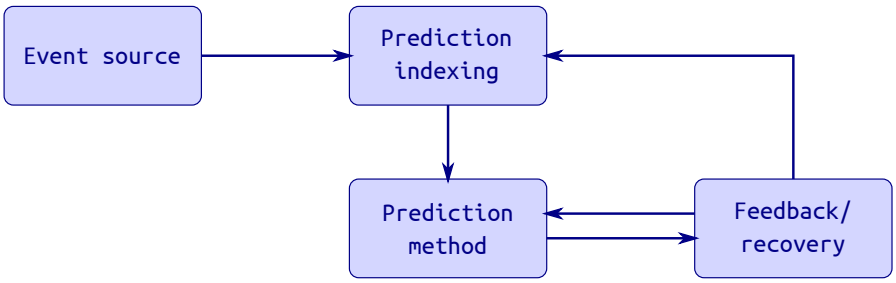

Figure 3.1. General dynamic branch predictor scheme

the actual branch instruction, indexes a table with information about the past behavior of that branch (*local history*) or of all previous branches (*global history*). The information read from that table is used to make prediction and finally, when the real outcome of the branch is resolved, the tables and the prediction method are updated, taking countermeasures in case of misprediction. It is effectively a feedback control system.

3.2.1 Basic predictor structures

The most basic dynamic predictor consists of a table of 2^k entries (flip-flops), called Branch History Table (BHT), addressed using k bits from the branch PC, that stores a single bit at each location to predict if the branch will be taken (value 1) or not taken (value 0). When the branch outcome is resolved, the table is updated so that it always predicts the direction that the branch took the last time. Figure 3.2 shows this scheme.

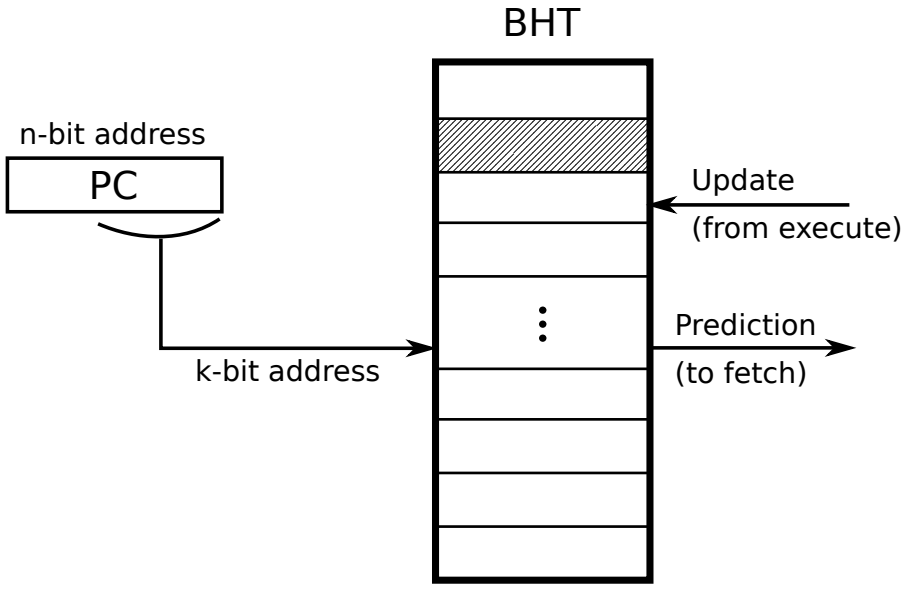

Figure 3.2. One-level branch predictor

This one-bit model is however quite weak, especially with nested loops, as the inner loop is mispredicted twice, at the last iteration when exiting and at the next first iteration when entering again.

This predictor can be improved by introducing some hysteresis in the system using two bits instead of one. This way, the BHT is composed by 2-bit entries that work as saturating counters, with their value incremented saturating at 3 every time the branch is taken and decremented saturating at 0 when the branch is not taken. The most significant bit of the counter provides the prediction, that changes only after two consecutive mispredictions.

These designs, known as *one-level* or *bimodal* predictors, could be extended to n-bit saturating counters, but solutions with more than two bits are rarely employed, because the size of the table is the limiting factor, given that, by using only a subset of PC bits, multiple branches could index the same BHT entry, thus producing *aliasing* issues.

3.2.2 Two-level branch predictors

An improvement over the simple predictors of the previous section comes from the concept of *correlation* between branches. Consider for example the following code:

```
if (x)          // branch 1
    a = 0;
if (y)          // branch 2
    b = 0;
if (a != b)     // branch 3
    ...
```

If the first two branches are taken, then the third one will be not taken for sure: these three branches are deterministically correlated.

A design that exploits such correlations was first proposed in [8] and it is called *two-level predictor*, shown in figure 3.3. It features an k-bit BHT shift register storing the outcome of the last k executed branches (first level), pointing to Pattern History Table (PHT) (second level) which stores 2^k 2-bit counters, one for each BHT pattern combination. In the example above, the two-level predictor would

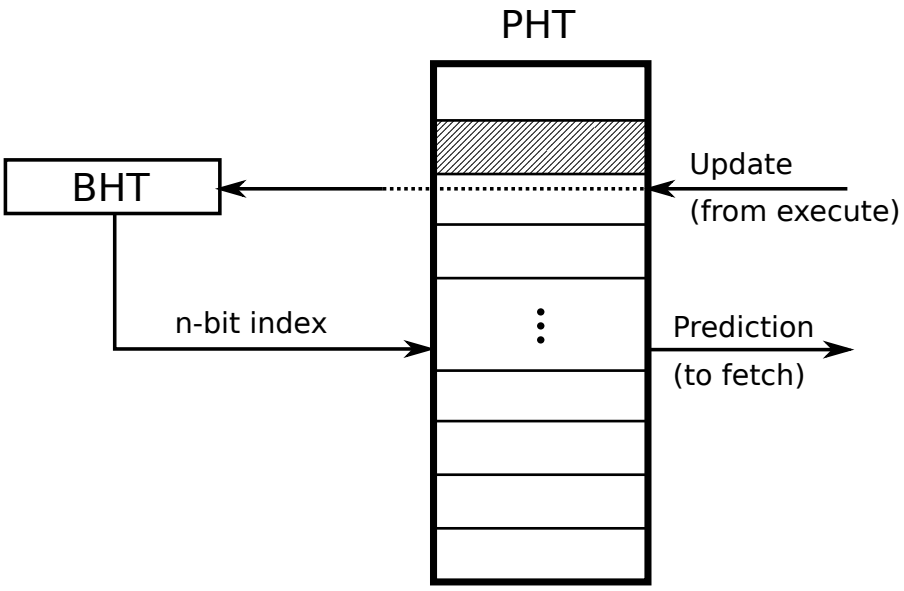

Figure 3.3. Two-level branch predictor

successfully predict as not taken the third branch if the global history stored in the BHT indicated that the previous two branches were taken.

This scheme, however, has lost the *local* information about the current branch instruction, relying only on the global history of branches for the prediction. Thus, nine variants were proposed [9] that exploit either one or both the local and global information (figure 3.4), by storing, for instance, multiple BHTs indexed by the

branch address.

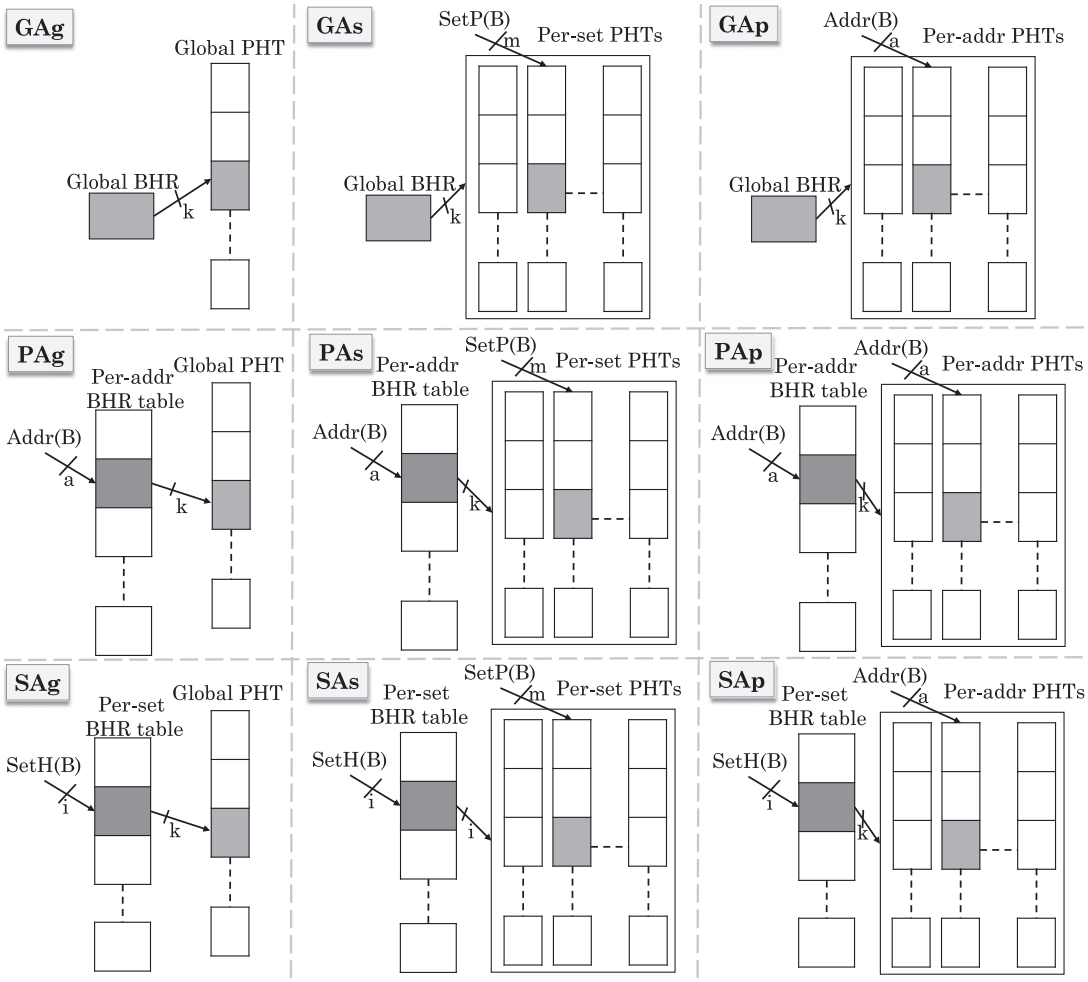

Figure 3.4. Two-level predictor variations [5]

3.3 State-of-the-art branch predictors

Building on the schemes described in the previous sections developed in the 90s, nowadays modern high-performance processors use very advanced design for branch predictors, that even occupy a significant area in the chip die. Two main classes of state-of-the-art branch predictors are used today: *TAGE-based* predictors and *perceptron-based* predictors.

3.3.1 Tagged Geometric predictor

Tagged Geometric (TAGE) predictors [10] use a series of global predictors indexed with histories of different length, like the scheme shown in figure 3.5. The base pre-

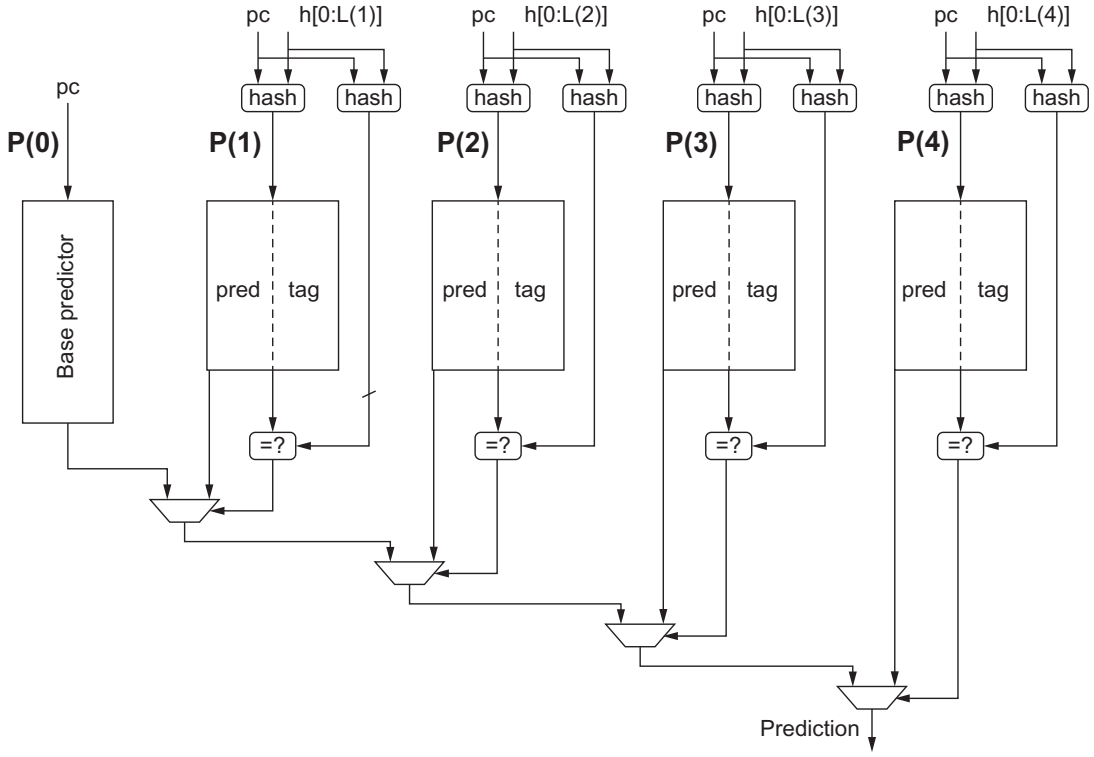

Figure 3.5. TAGE predictor [3, p. 188]

dictor can be as simple as a basic bimodal predictor, while the others are variable-length two-level predictors that combine local and global branch information by hashing part of the branch PC with the BHTs.

This predictor uses *tagging* to avoid aliasing saving a subset of the bits of the PC not used for indexing in a dedicated field in the PHT. All the predictors are accessed simultaneously and if more than one two-level predictors have a match between the branch address and the tag, then the prediction coming from the one with the longest global history is selected. If no two-level predictor hits, then the base predictor is used as a fallback.

Variants of this TAGE predictor have been shown to win annual branch prediction competitions without needing too much memory size [3, p. 189] and are present in many high-end CPUs.

3.3.2 Perceptron predictor

These kinds of predictor take a completely different approach to the problem with respect to previously analyzed designs. The idea is based around the concept of the *perceptron* [11], a single-layer artificial neuron, whose structure is shown in figure 3.6. The perceptron receives a certain number of inputs $x_1 \dots x_n$, that in the case

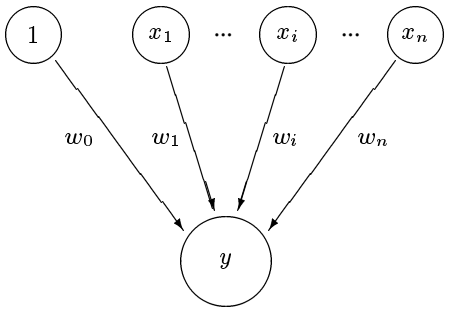

Figure 3.6. Perceptron

of branch prediction correspond to the entries of the global history register (the previous outcomes), and computes the output y as a weighted sum of its inputs:

$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

If the output turns out to be non-negative, then the branch is predicted as taken, otherwise as not taken.

The weights express the degree of correlation between the current and previous branches, specifically weight w_i indicates how much the current branch is biased toward the result of the last-but-i branch. The input corresponding to weight w_0 is always 1, to indicate the intrinsic bias of the current branch (if it is more likely to be taken or not regardless of previous history). These weights are updated with a training algorithm that is executed every time a new branch resolution arrives.

The structure of the complete perceptron predictor is shown in figure 3.7 and features a table of perceptrons indexed by the branch address, a block that computes the prediction starting from the selected perceptron and the global history and a training unit dedicated to updating the weights upon a new actual branch result.

Research [11] has shown that this new approach offers complementary strengths to the previous ones and so that an optimal solution is developing a hybrid predictor between the two. At the present time, high-end processors for both desktop and mobile make use of some kind of perceptron-based prediction network, such as the AMD Ryzen and Samsung Exynos families.

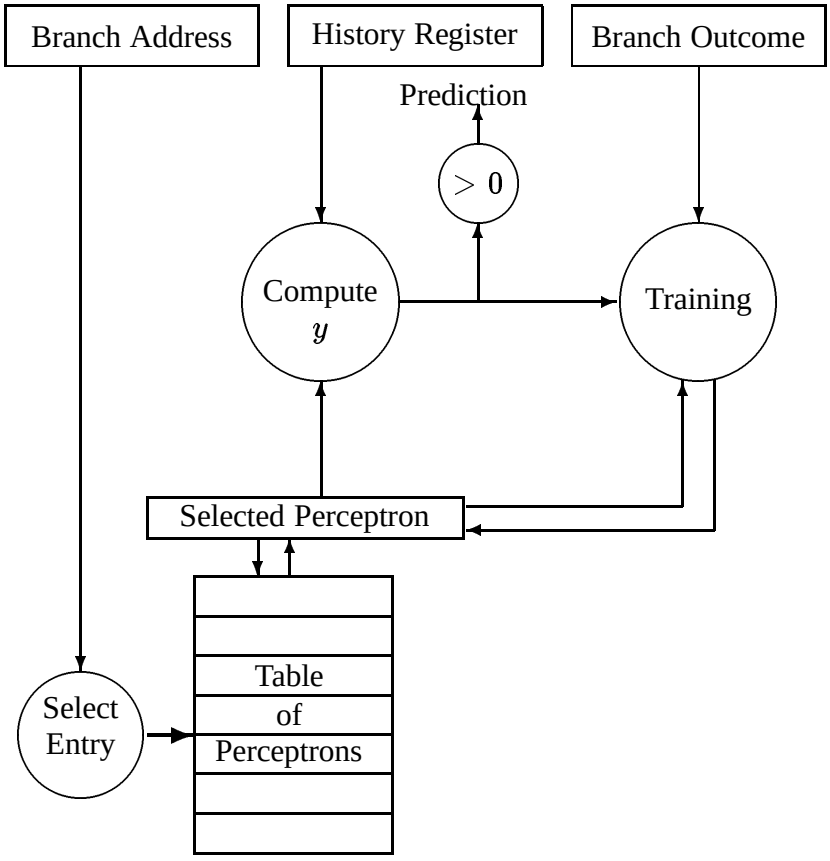

Figure 3.7. Perceptron-based branch predictor

Chapter 4

Proposed design

4.1 General scheme

4.2 PC gen stage

4.3 Fetch stage

4.3.1 Branch Prediction Unit (BPU)

4.3.2 Instruction cache interface

4.3.3 Fetch unit

4.4 Execution stage

4.4.1 Branch unit

Chapter 5

Results

5.1 Simulation

5.1.1 Fetch unit

5.1.2 BPU

5.2 BPU benchmarking

5.3 Synthesis results

Chapter 6

Concluding remarks

6.1 Future work

Bibliography

[1] Waterman A., *Design of the RISC-V Instruction Set Architecture*, PhD diss., Electrical Engineering and Computer Sciences, University of California at Berkeley, 2016, UCB/EECS-2016-1.

[2] Patterson D., Waterman A., *The RISC-V Reader: An Open Architecture Atlas*, First edition, Strawberry Canyon, 2017.

[3] Hennessy J., Patterson D., *Computer Architecture: A Quantitative Approach*, Sixth edition, Morgan Kaufmann, 2017.

[4] Thornton J., "Parallel operation in the control data 6600", *Proceedings of the fall joint computer conference, part II: very high speed computer systems*, vol. 26, pp. 33–40, 1965.

[5] Mittal S., "A Survey of Techniques for Dynamic Branch Prediction", *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, 2019.

[6] Smith J., "A study of branch prediction strategies", *25 Years of the International Symposia on Computer Architecture*, pp. 202–215, 1998.

[7] Gross T., Hennessy J., "Optimizing delayed branches", *ACM SIGMICRO Newsletter*, vol. 13, pp. 114–120, 1982.

[8] Yeh T., Patt Y., "Two-level adaptive training branch prediction", *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, 1991.

[9] Yeh T., Patt Y., "A comparison of dynamic branch predictors that use two levels of branch history", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, 1993.

[10] Seznec A., Michaud P., "A case for (partially)-tagged geometric history length predictors", *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

[11] Jimenez D., Lin C., "Dynamic branch prediction with perceptrons", *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, 2001.