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, being based on modular and optional ISA extensions, that allow designers to tailor the architecture to their specific needs.

LEN5 is an an open source RISC-V core implementing the RV64G instruction set (integer, multiply and divide, floating point and atomic operations), based on Tomasulo's algorithm to schedule its out-of-order execution. This choice was made based on the fact that many modern microprocessor feature a dynamically scheduled pipeline, which offers the best performance and ILP exploitation at the cost, however, of high hardware complexity. This work in particular focuses on the frontend of LEN5, that is the part of the core responsible of generating addresses, predicting next directions and fetching instructions from memory to be issued to the execution stages.

The Instruction Fetch Unit (IFU) receives the current PC and interfaces with the instruction cache to read at the required address. Previously read cache lines are saved into line registers which reduce the total number of memory accesses, by allowing next sequential instructions to be immediately read from those registers. Moreover, this unit features a controller which handles both normal fetch operations and pipeline stalls in case of exceptional behavior, cache misses or busy issue queue.

Another significant part of the frontend is the Branch Prediction Unit (BPU), which predicts the direction of branch instructions in order to continue fetching immediately from the next predicted path of instructions. In order to accomplish that, it features a gshare branch predictor, which employs a subset of bits of the current PC and a global history of branch outcomes to perform a prediction. Furthermore, along with branch *direction*, also the branch *destination* address is predicted thanks to a Branch Target Buffer (BTB), which is a small direct mapped cache storing the next address after taken branches, to allow for zero latency branch instructions.

This work is intended as an exploration of such complex out-of-order architectures, to actually experience firsthand the main issues and tradeoffs designers must face and to contribute to the growing panorama of open source cores. 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, which could be relevant both in teaching and research. As mentioned before, the entire work will be open sourced and available in a GitHub repository.

Ringraziamenti

Ripensando al percorso che mi ha portato fino a questo punto, mi sento di ringraziare innanzitutto tutti i professori di cui ho avuto la possibilità di seguire i corsi per le conoscenze e le competenze che mi hanno aiutato ad acquisire e che saranno alla base della mia futura carriera. In particolare il professor Martina, per avermi dato la possibilità di lavorare a questo stimolante progetto e per aver ascoltato con pazienza i problemi con cui mi sono scontrato, sapendo sempre guidarmi verso la strada migliore.

In seguito, un grazie è dovuto ad Andrea, che dal cuore della Silicon Valley ha avuto lidea per questo progetto su RISC-V e che ha sempre saputo gestire landamento dei lavori in modo corretto.

Un grazie particolare lo rivolgo ai miei compagni di viaggio in questo progetto Matteo e Michele, così come a tutti gli altri miei colleghi di questi anni, perché tutte le volte che abbiamo consegnato un lavoro alle due di notte o abbiamo avuto paura insieme per gli esami il giorno dopo, alla fine ci siamo sempre divertiti e abbiamo vissuto momenti che ricorderemo per molto tempo.

Fuori dal mondo accademico, grazie ai miei genitori per il supporto incondizionato che mi hanno sempre offerto in ogni mia scelta, per i saggi consigli che mi hanno saputo dispensare e per avermi messo in condizione di poter portare a termine al meglio questo percorso, concentrandomi esclusivamente sui miei obiettivi.

Grazie a Noemi, per essere stata sempre al mio fianco e per aver gioito con me ad ogni successo, per avermi sempre messo di fronte la realtà in modo obiettivo quando mi sono trovato a prendere delle decisioni e per avermi sopportato nei momenti di massimo impegno.

Infine grazie a tutti i nonni, quelli che non ci sono più, ma che avrebbero voluto vedermi ancora una volta raggiungere un traguardo così importante e quelli che ci sono ancora e che costantemente mi fanno sentire il loro amore e il loro appoggio.

Marco

Contents

List of Tables

List of Figures

List of Acronyms

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 **ISA!** (**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 open source and 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 Asanovi and David Patterson, but soon developed into a fully featured **ISA!**, presented several years later in Waterman's dissertation [**?**].

Why a new ISA from scratch?

Today the goal of RISC-V is to become a universal **ISA!** [**?**], 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 **??**).

RISC-V belongs to a non-profit foundation, composed of 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 **ISA!**s 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 **??**).

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 **??** contains a list of available extensions at the time of writing.

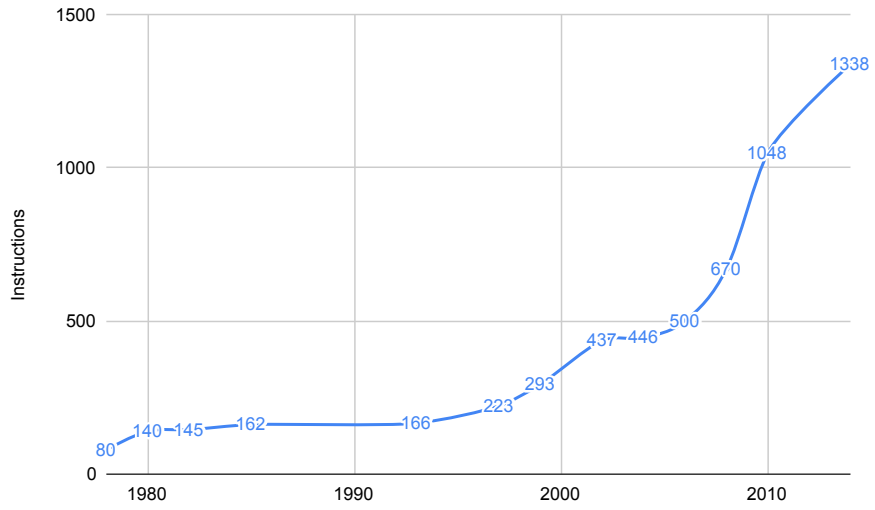


Figure 1.1: x86 instruction count over time. Data taken from [**?**, p. 3]

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 [**?**]

1.1.2 Comparison with other ISA!s

Arguably the two most popular **ISA!**s at the present time are Intel x86 and ARM, which are dominant in the personal computers/server and smartphones/tablets markets respectively. The first significant difference between them and RISC-V is that they are *proprietary* **ISA!**s, 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 **ISA!**s, which do not have a zero register, can be synthesized using RISC-V 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 **PC!** (**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 instead 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 [**?**, 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 [**?**, 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 ***IC!*** (**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 ***CPI!*** (**CPI!**)[1] can be derived. By factoring in these quantities, the final expression of CPU time is as follows [**?**, p. 53]:

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

Equation (**??**) 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 **??**) 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 ***IPC!*** (**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 **??**, taking a total of 15 clock cycles.

Figure 2.1: Subscalar processor[2]

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 ***ILP!*** (**ILP!**), which is a fundamental concept in developing techniques to enhance processor performance. For the same example of figure **??**, a pipelined processor could theoretically achieve a **CPI!** of 1, executing one instruction for each clock cycle (see figure **??**). Processors of this kind are called *scalar*.

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

[1]Taken from https://commons.wikimedia.org/wiki/File:Nopipeline.png under the license Attribution-Share Alike 3.0 Unported

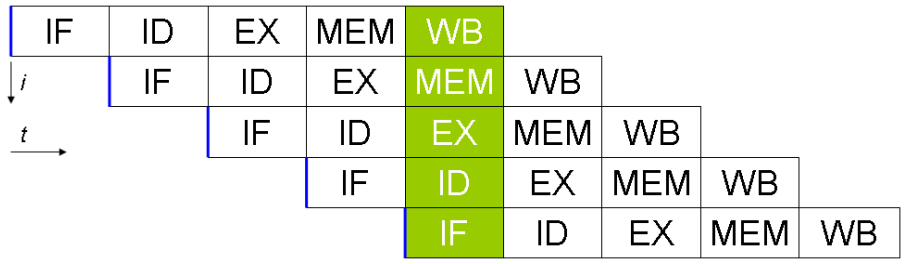


Figure 2.2: Scalar processor[3]

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 [**?**, 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 (**??**): 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.

[3]Taken from https://commons.wikimedia.org/wiki/File:Fivestagespipeline.png under the license Attribution-Share Alike 3.0 Unported

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 **??** shows the execution scheme for a multiple issue processor.

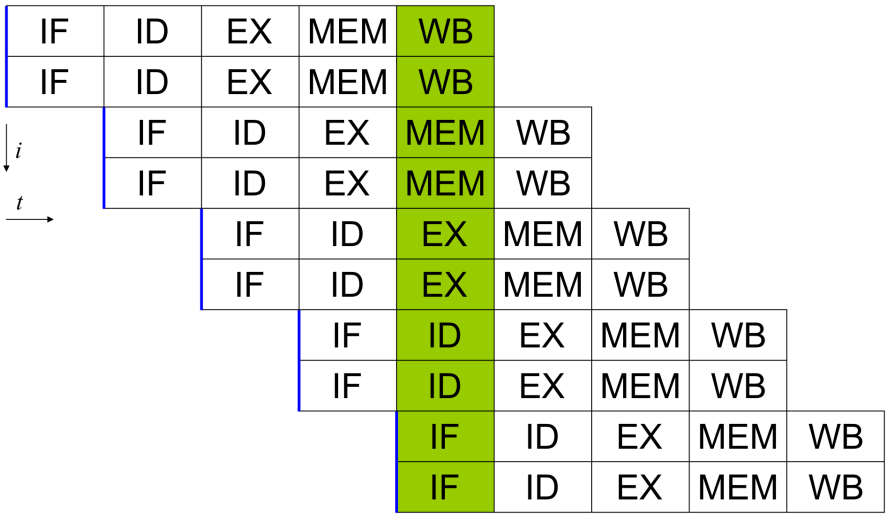


Figure 2.3: Multiple-issue processor with two pipelines[4]

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 themselves efficient only for specific applications presenting a high level of data parallelism [**?**, p. 168], mainly because the compiler software needs a

[4]Taken from https://commons.wikimedia.org/wiki/File:Superscalarpipeline.png under the license Attribution-Share Alike 3.0 Unported

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 **WOE!** (**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 **IPC!** (**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 [**?**, 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 **??**, 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[5]. 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:

- **RAW! (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
```

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

More formally, **RAW!** hazards occur if:

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

- **WAR! (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:

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

- **WAW! (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 [**?**] 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 **??**. The first one is the *instruction status* table, that keeps track of which 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 **??**, based on the one included in the MIPS 10000. The *rename table* keeps track of the mapping between architectural and physical registers, to maintain correct value references, while the *register free list* contains the names of all available physical registers to be used for renaming.

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Figure 2.4: Scoreboard structure

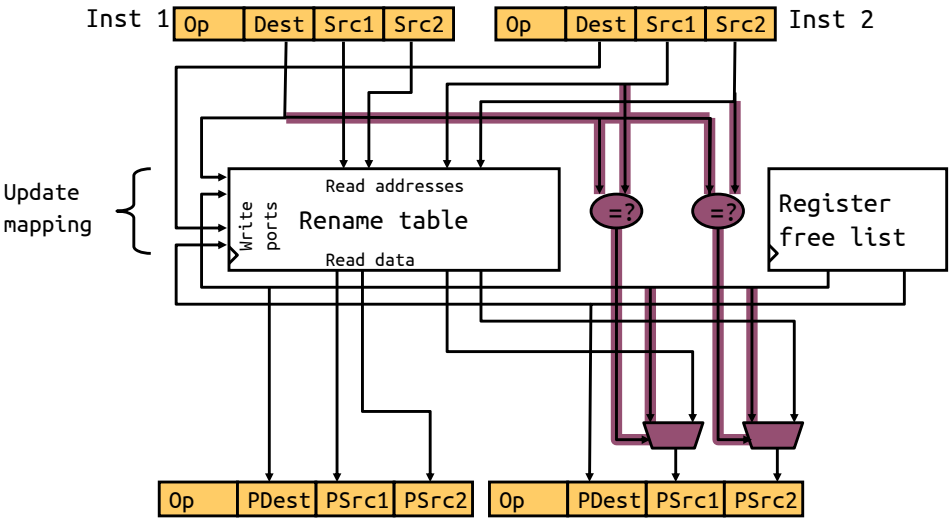


Figure 2.5: Renaming unit

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 **FPU!** (**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 **??**.

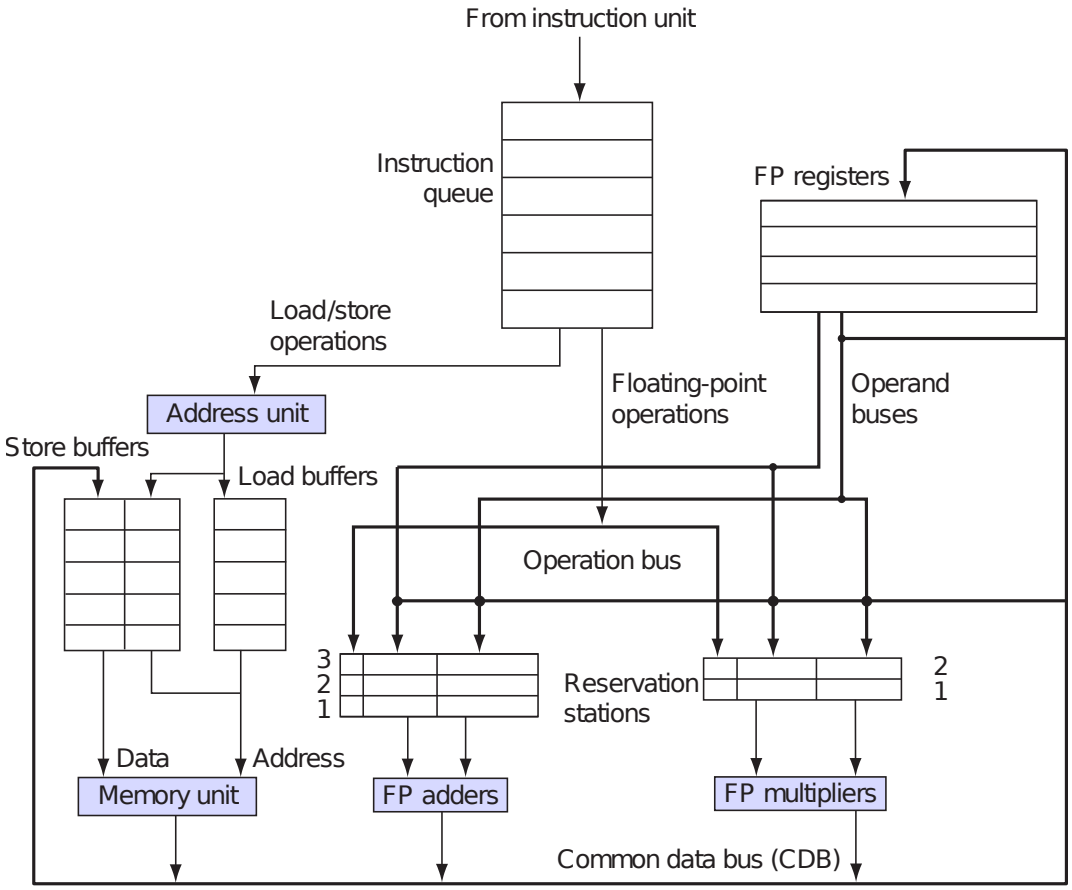


Figure 2.6: **FPU!** example architecture using Tomasulo's algorithm [**?**, p. 198]

Each reservation station contains several fields providing similar information with respect to the 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 **CDB!** (**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 [**?**, 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 **ISA!**s 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 [**?**, p. 208]:

- Dynamic branch prediction, to fetch and issue instructions before the outcome of a branch is determined (refer to section **??** 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 ROB!

In both scoreboarding and Tomasulo's approach, instruction commit can be handled using a dedicated hardware structure called **ROB!** (**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 **??**, highlighting which parts of the pipeline are in-order and which out-of-order.

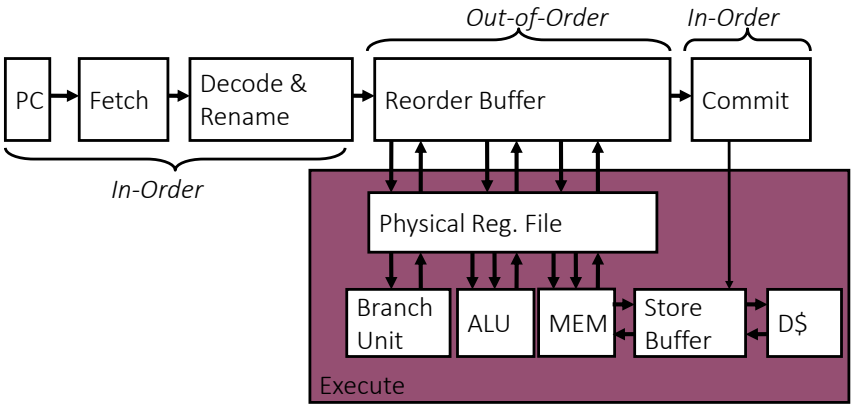


Figure 2.7: An out-of-order pipeline featuring register renaming and **ROB!**

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 **??** 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.

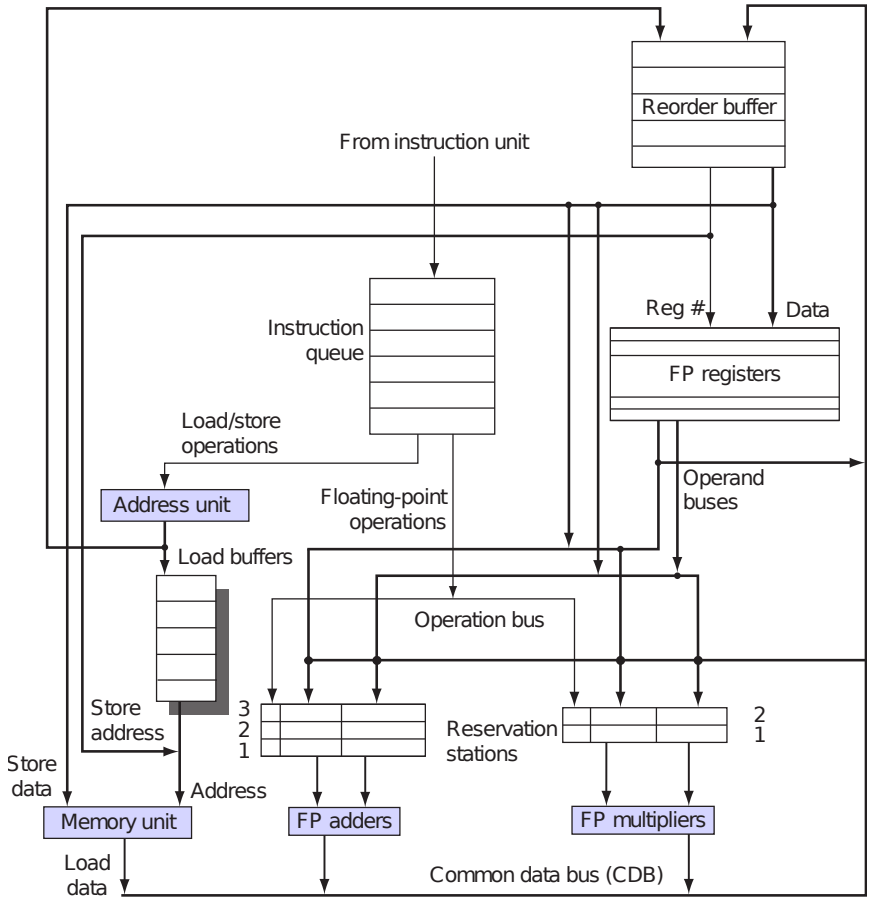


Figure 2.8: **FPU!** example architecture using Tomasulo's algorithm and **ROB!** [**?**, p. 210]

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 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 **??** provides all the important differences at a glance.

Name	**Issue structure**	**Hazard detection**	**Schedul-ing**	**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 [**?**, p. 219]

Chapter 3

Branch prediction techniques

As stated throughout the previous chapter, as well as in equation (**??**), 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 [**?**]:

- 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 [**?**], 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 **PC!**s) 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 the body and so produces a backward branch if true. This can however introduce some delay needed to compute whether the target of the branch is higher or lower than the current **PC!**.
- Delayed branch [**?**]: 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 executed 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 **??**.

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 satisfactory 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 **??**. An event source, that is the

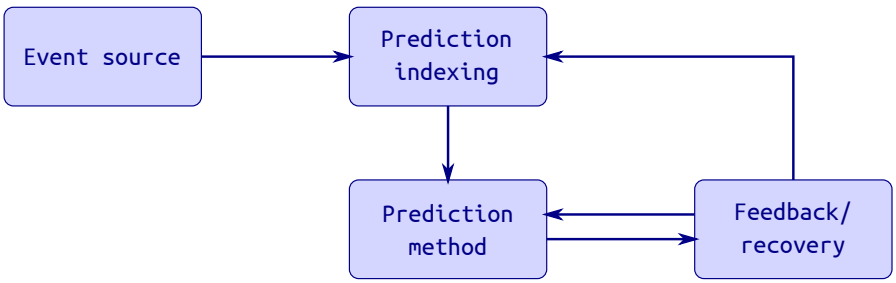


Figure 3.1: General dynamic branch predictor scheme

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 the 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 **BHT!** (**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 **??** shows this scheme.

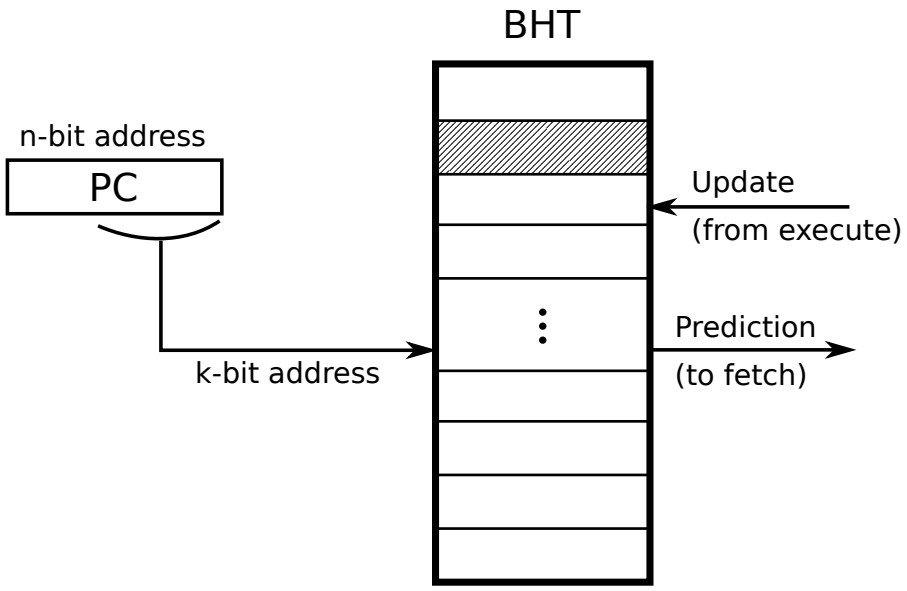


Figure 3.2: One-level branch predictor[1]

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

[1]Edited from https://commons.wikimedia.org/wiki/File:Two-level_branch_prediction.svg under the license Attribution-Share Alike 3.0 Unported

only after two consecutive mispredictions. These 2-bit counters are nothing more

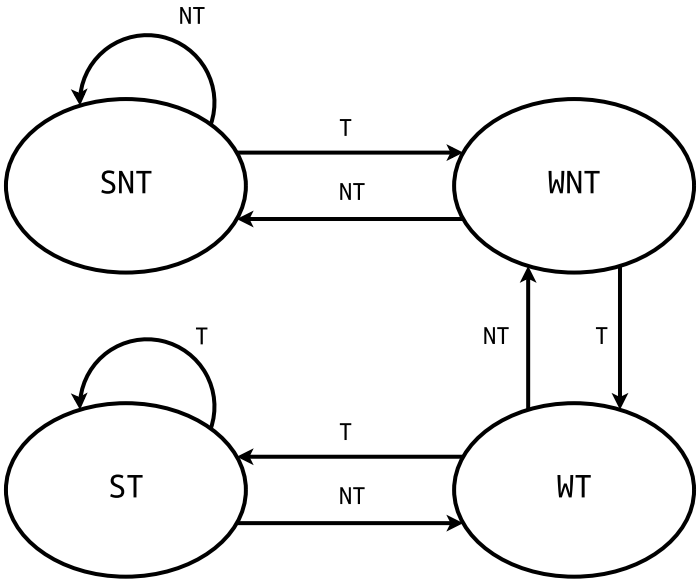


Figure 3.3: 2-bit counter FSM

than simple state machines, that work as illustrated in **??**. **??** shows the list of the counter states and their description.

State	**Counter value**	**Prediction**
Strongly not taken (SNT)	00	Not taken
Weakly not taken (WNT)	01	Not taken
Weakly taken (WT)	10	Taken
Strongly taken (ST)	11	Taken

Table 3.1: 2-bit counter state description

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 actual 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, which means that these three branches are deterministically correlated.

A design that exploits such correlations was first proposed in [**?**] and it is called *two-level predictor*, shown in figure **??**. It features an k-bit **BHT!** shift register storing the outcome of the last k executed branches (first level), pointing to a **PHT!** (**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 successfully

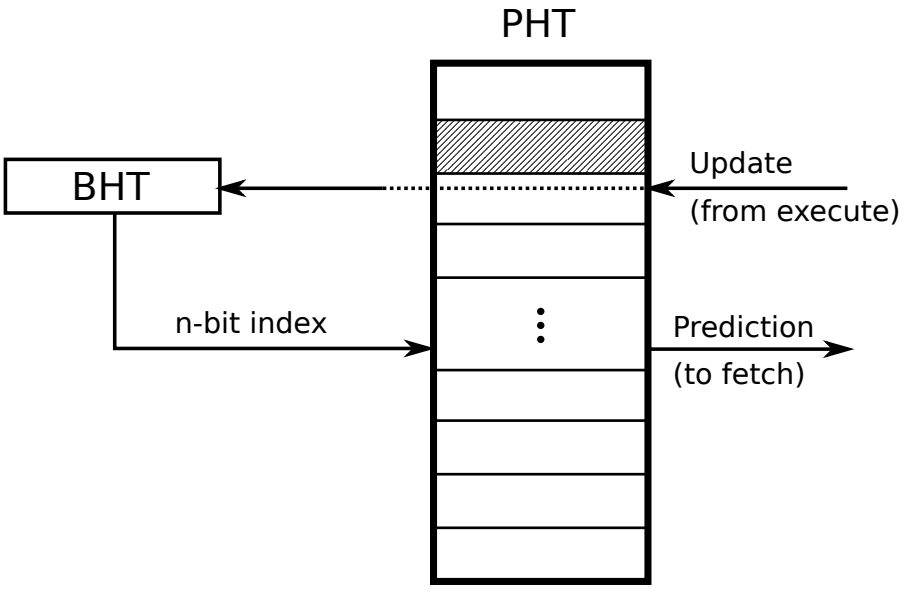


Figure 3.4: Two-level branch predictor[2]

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 [**?**] that exploit either one or both the local and global information (figure **??**), by storing, for instance, multiple **BHT!**s indexed by the branch address.

[2]Taken from https://commons.wikimedia.org/wiki/File:Two-level_branch_prediction.svg under the license Attribution-Share Alike 3.0 Unported

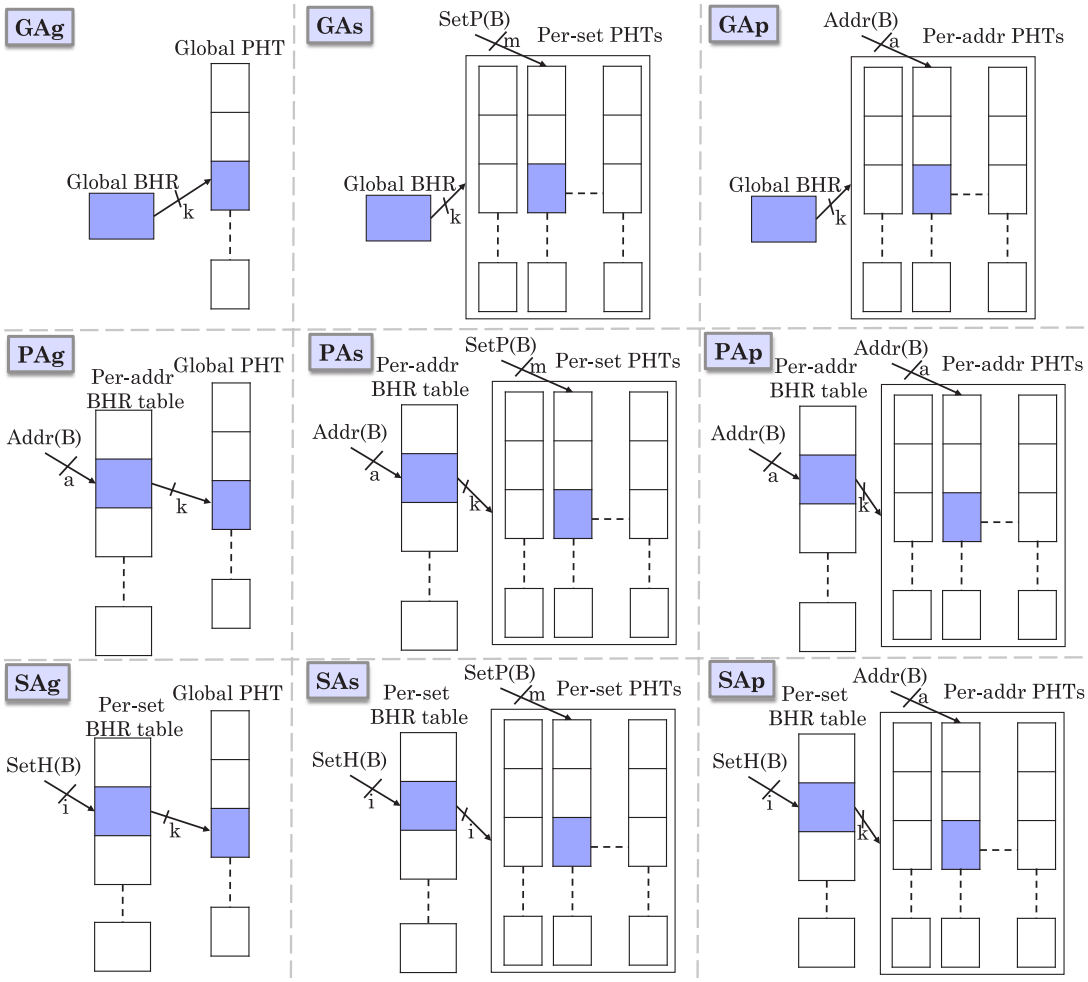


Figure 3.5: Two-level predictor variations [**?**]

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 on 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 TAGE! predictor

TAGE! (**TAGE!**) predictors [**?**] use a series of global predictors indexed with histories of different length, like in the scheme shown in figure **??**. The base predictor

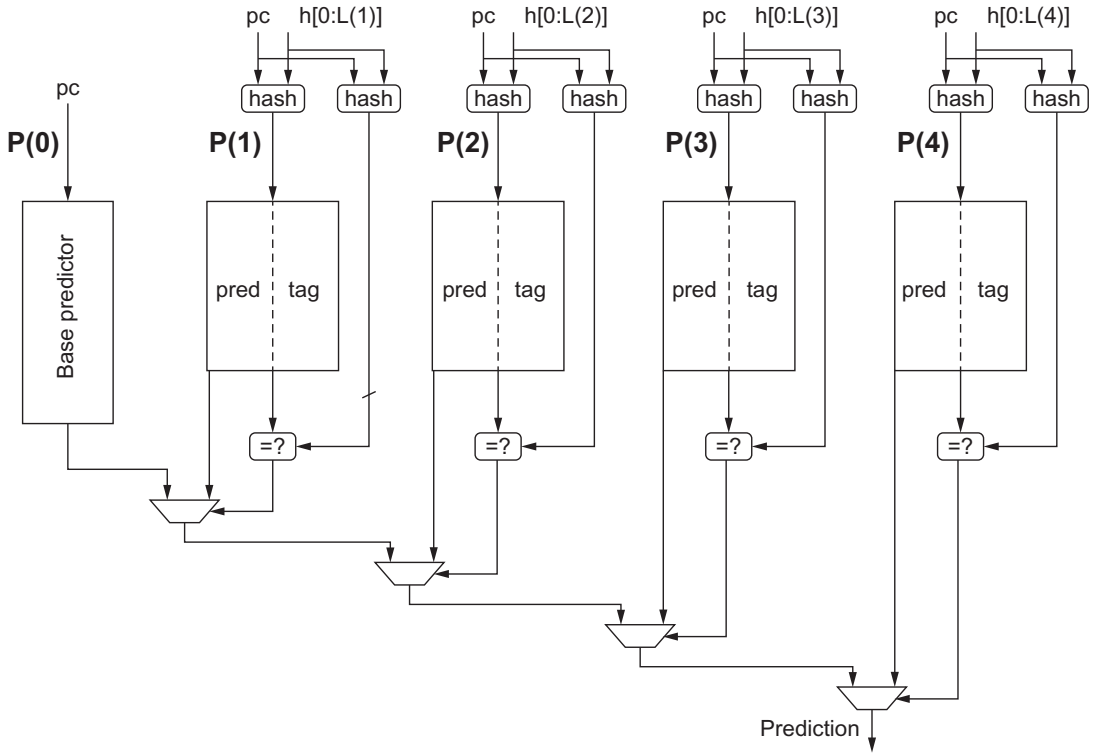


Figure 3.6: **TAGE!** predictor [**?**, p. 188]

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 **BHT!**s.

This predictor uses *tagging* to avoid aliasing by 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 [**?**, 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* [**?**], a single-layer artificial neuron, whose structure is shown in figure **??**. The perceptron receives a certain number of inputs $x_1 \dots x_n$, that in the case

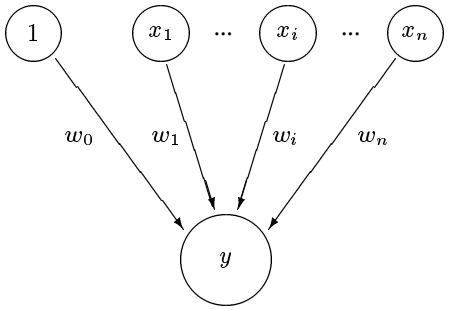


Figure 3.7: Perceptron

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 **??** 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.

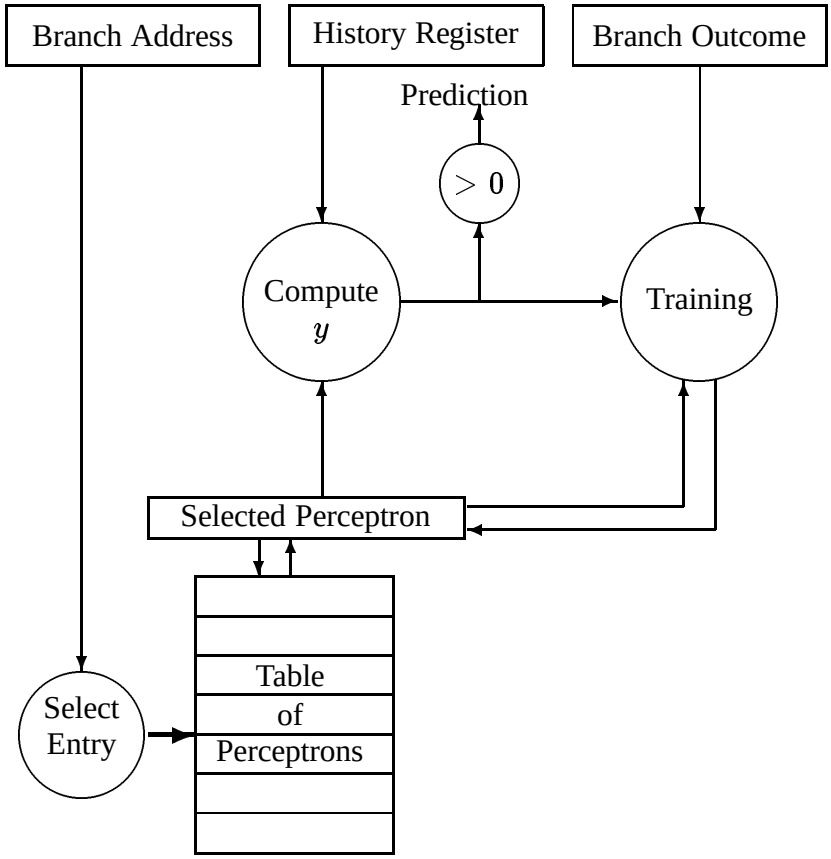


Figure 3.8: Perceptron-based branch predictor

Research [**?**] 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.

Chapter 4

LEN5 frontend

4.1 General block diagram

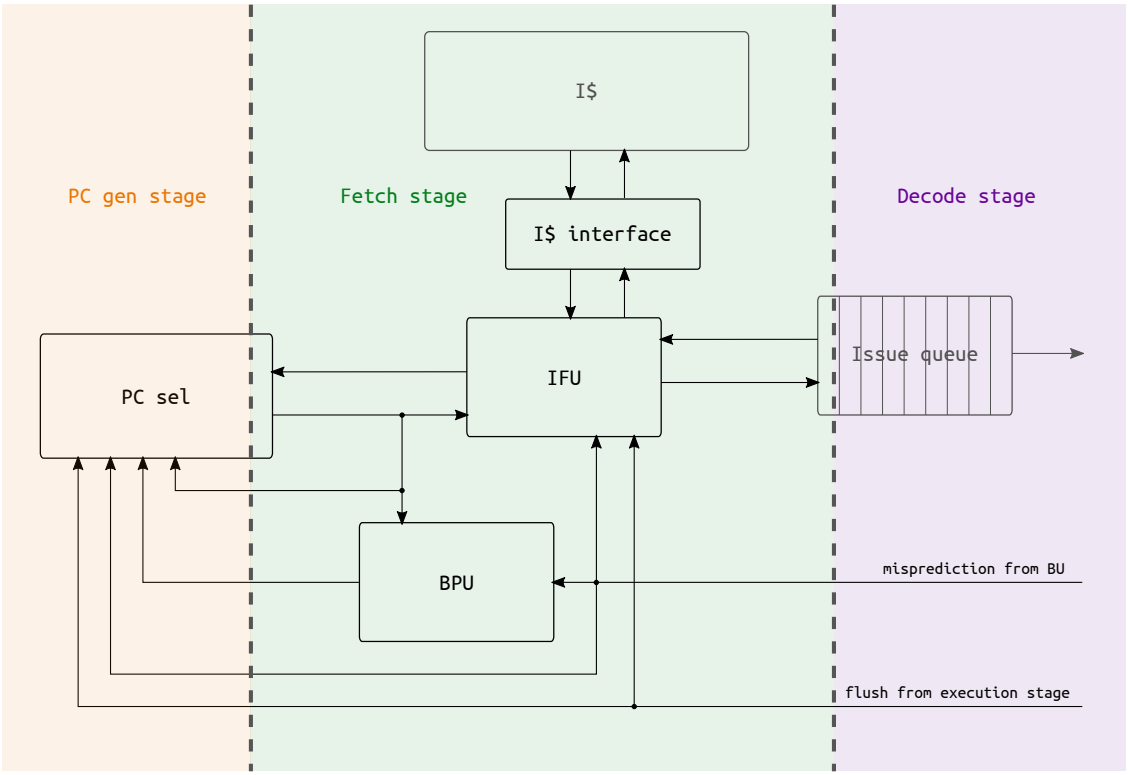


Figure 4.1: LEN5 frontend

?? shows a top-level block diagram of the LEN5 frontend, with the modules that were developed and that will be described in the following sections shown in solid black color. Gray blocks are instead the ones the frontend interfaces with, that are

the instruction cache, or equivalently the memory management unit, and the issue queue.

The frontend is composed of two pipeline stages, namely the ***PC!** generation* and the *fetch* stages. **??** also shows the *decode* stage, where the issue queue is found. The latter is basically a **FIFO!** that serves as a buffer interface between the frontend and the backend of the processor by storing a queue of instructions to be issued to the later stages of the pipeline.

4.1.1 PC! gen stage

In the **PC!** generation (**PC!** gen) stage, the next **PC!** is selected among a number of different options by the **PC!** sel block, using a predefined priority, and is then written on the output register of the stage. This register also serves as the pipeline register between the two stages, and that is why **??** shows a dashed gray line crossing the **PC!** sel block.

The selection of the new **PC!** is carried out by a network of combinational logic, so that this stage always takes exactly one clock cycle.

4.1.2 Fetch stage

In the fetch stage, the **PC!** is used by the **IFU!** (**IFU!**) to select and possibly read from memory the next instruction to be pushed to the issue queue. At the same time, the **BPU!** (**BPU!**) uses the current address to predict the next direction in case of branch and passes such information back to the **PC!** gen stage. Memory accesses are performed through the instruction cache interface which manages the control signals to the instruction cache.

The latency of this stage is at least two clock cycles (see **??**); in a normal steady state the **IFU!** can provide a throughput of one instruction each clock cycle to the issue queue, but in case of cache miss the number of cycles to resolve the stall can grow significantly, so the latency cannot be determined in advance. The issue queue is there exactly to provide some elasticity to the pipeline, by buffering already fetched instructions, masking at least in part this unpredictable latency.

4.1.3 Handshake signals

The communication between each stage is always bidirectional, because in case of a stall, caused for instance by a cache miss, by a full issue queue or by some other exceptional behavior down the pipeline, the **PC!** generation process must be interrupted along with the fetch. In order to do so, a handshake process handles the communication between each stage as well as between the instruction cache interface and the actual cache.

This handshake mechanism is based on the AXI valid/ready protocol described below, even if it is not compliant with all the AXI specifications.

In each communication the source of data generates a *valid* signal to indicate that the information is available, while the destination generates a *ready* signal to indicate that it can accept such information [**?**, p. A3-41]. The handshake takes place and the information is successfully exchanged only at the rising clock edge when both valid and ready are asserted. For example, in **??**, the handshake happens at the third rising edge of the clock.

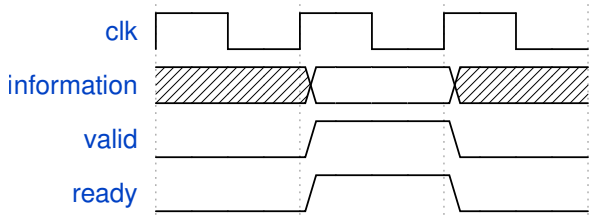
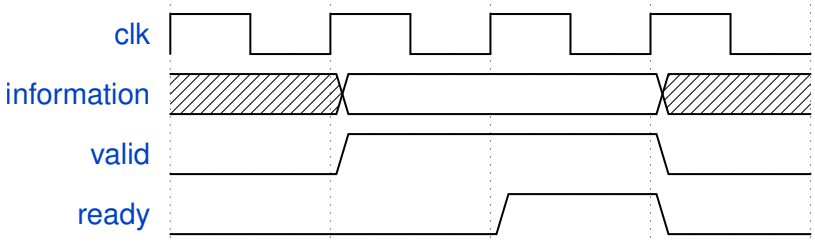
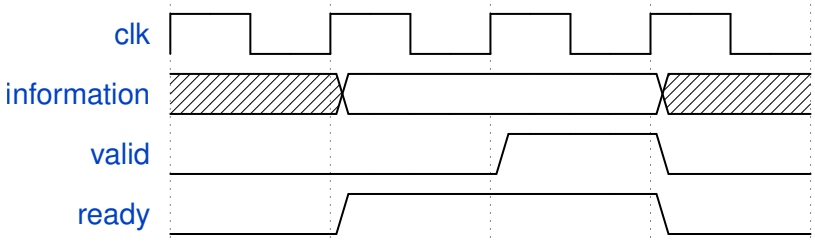


Figure 4.2: AXI handshake protocol

When a source has information available (**??**), it must assert valid and then wait until the corresponding ready is produced. It cannot wait for the ready before asserting valid. On the other hand (**??**), a destination is allowed to wait for its



(a) Valid before ready



(b) Ready before valid

Figure 4.3: Possible handshake timings

valid before asserting ready and it can also deassert ready before a corresponding valid arrives, which is necessary if the destination becomes busy for other reasons.

4.2 PC! gen stage

The selection of the next **PC!** is based on the following list of priorities, from highest to lowest:

1. **Exception**[1]: if an exception occurs, the **PC!** gen stage will receive the next starting address as the base address present in the vector table provided by the CSR unit.

2. **Misprediction**: if a resolved branch is discovered to have been mispredicted, then the **PC!** gen stage resumes execution from the correct target if the branch was actually taken, or from the next sequential address from the branch **PC!** if it was actually not taken.

3. **Branch prediction**: if the **BPU!** predicts a taken branch for the current **PC!** then it provides this stage with the predicted target address (see **??**), which will be fetched at the next cycle, thus allowing for zero penalty branches when predicted correctly.

4. **Default assignment**: if none of the conditions before occur, then the next **PC!** is selected as usual as the next sequential address, which corresponds to the current **PC!**+4 for word-aligned 32-bit instructions.

?? shows the diagram of this stage. This and all the following diagrams in this document are color coded so that input signals are in blue and have the suffix `_i`, output signals are in red and have the suffix `_o`, internal signals in black and bit widths are in gray.

The heart of the **PC!** gen stage is the `pc_priority_enc` block, which is an encoder that takes as inputs all the status signals indicating a behavior different from the default and all the corresponding potential next **PC!**s. In behavioral SystemVerilog (**??**) it is described as an if-then-else chain, which gets synthesized as a list of cascading multiplexers implementing the desired priority[2].

[1] Or interrupt. The two terms can be used almost interchangeably in RISC-V.

[2] As opposed to the description using a case statement, which leads to a single parallel mux, with no priority encoded.

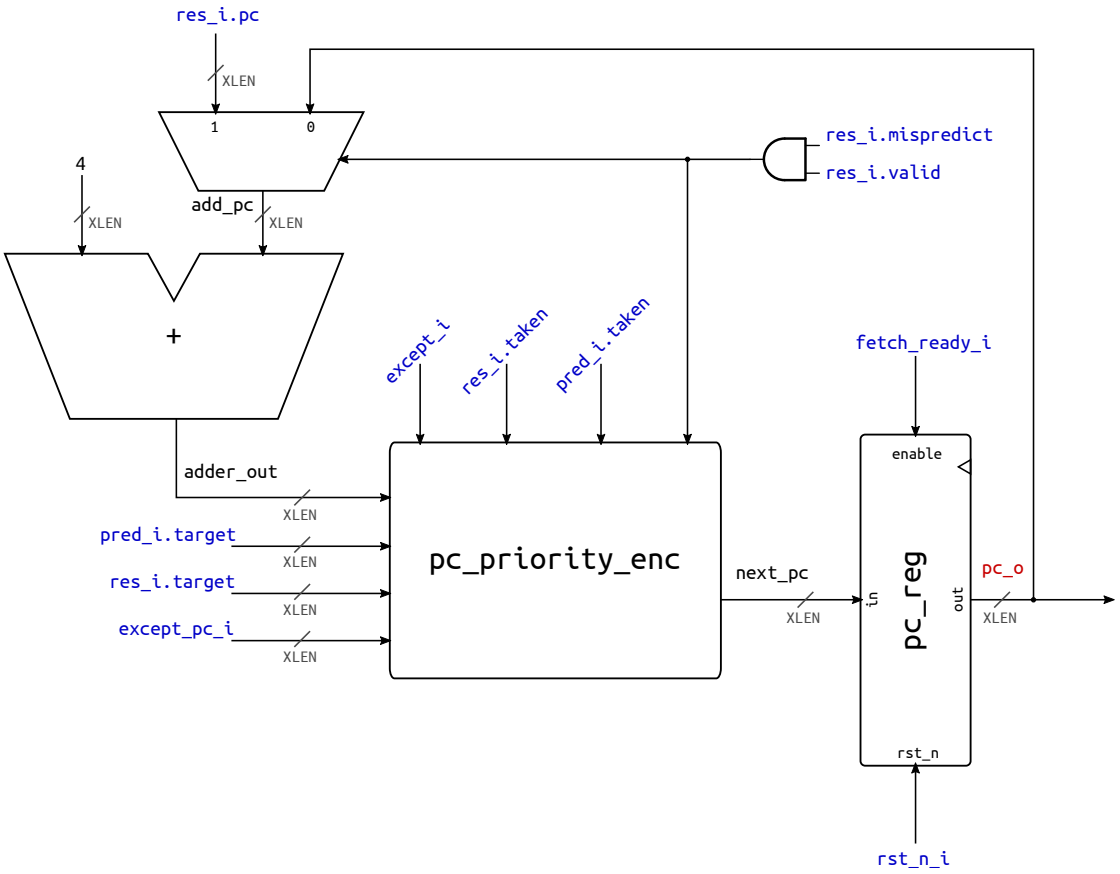


Figure 4.4: **PC!** gen stage diagram

```
always_comb begin: pc_priority_enc
  if (except_i) begin
    next_pc = except_pc_i;
  end else if (res_i.valid && res_i.mispredict) begin
    if (res_i.taken) begin
      next_pc = res_i.target;
    end else begin
      next_pc = adder_out;
    end
  end else if (pred_i.taken) begin
    next_pc = pred_i.target;
  end else begin
    next_pc = adder_out;
  end
end: pc_priority_enc
```

Listing 4.1: `pc_priority_enc` description

In order to save resources, a single adder is used to generate both the next sequential address and the next **PC!** after a mispredicted not taken branch. A multiplexer driven by the misprediction signals is used to select the right operand.

The final chosen next **PC!** is fed into the `pc_reg` output register for the later stages. The enable of this register is controlled by the signal `fetch_ready` which comes from the fetch stage and disables the **PC!** generation if a stall occurs. This is part of the handshake mechanism described in **??**, even if there is no *valid* signal from the **PC!** gen stage, as it is redundant due to the fact that a valid new **PC!** is always present at the output register.

4.3 Instruction cache interface

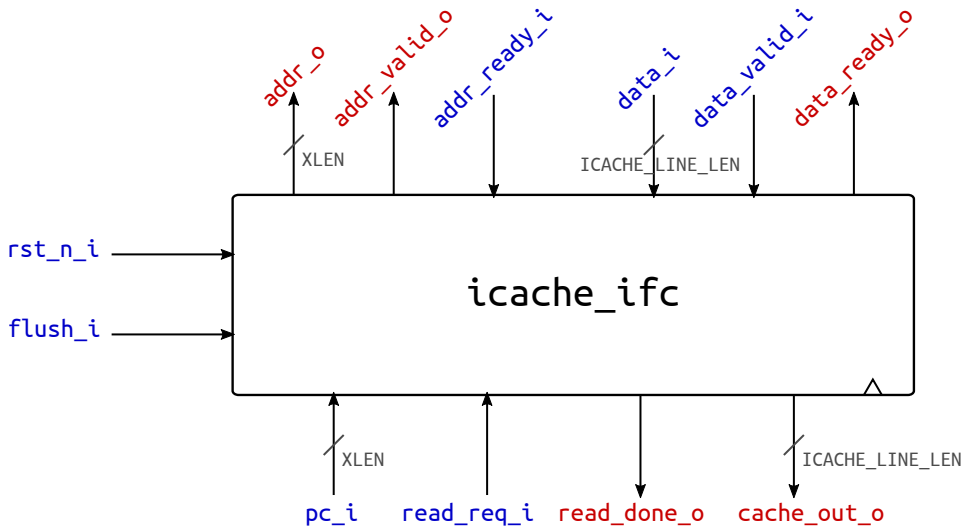


Figure 4.5: Instruction cache interface module ports

The instruction cache interface (**??**) is responsible for translating the fetch requests coming from the **IFU!** into compliant valid/ready handshake signals for both address and data to the instruction cache. This unit basically provides two main benefits. First, it simplifies the control of the **IFU!**, by delegating the handshake process. Second, and more important, it provides an additional separation layer between the core frontend and the instruction cache with modularity in mind so that, should the cache block be modified, only this interface unit needs to be updated, while the signals coming from the **IFU!** module would remain unchanged.

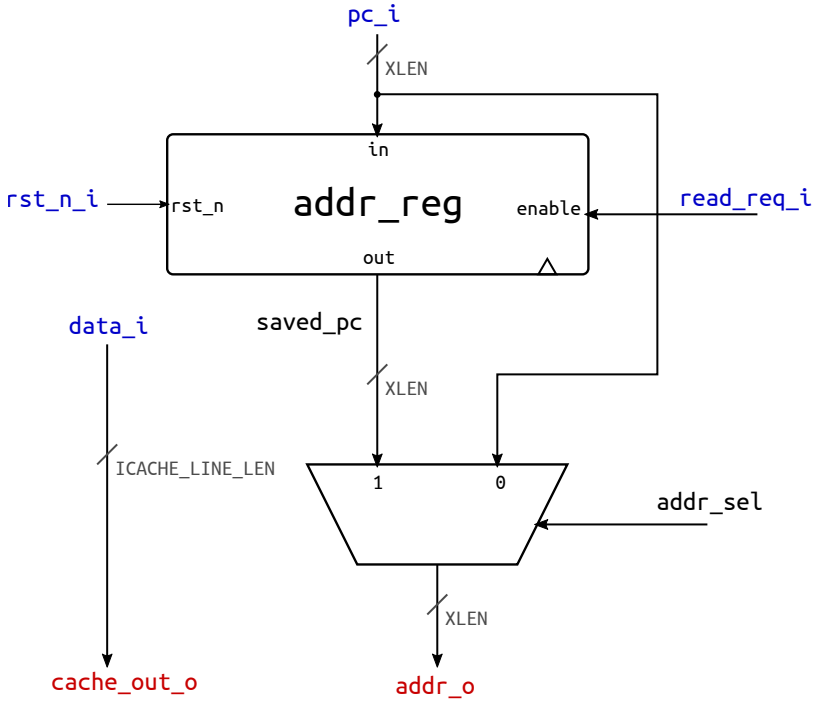


Figure 4.6: Instruction cache interface datapath

4.3.1 Datapath

For what concerns data signals, the line received from the instruction cache is directly connected with the `cache_out` output, as it is the cache itself that is responsible for keeping its output valid until the handshake occurs. On the other hand, no assumption can be done on the timing of the input address other than the fact that it will be valid in the same cycle when `read_req` is asserted. For this reason, given the possibility for the **PC!** to change at the following cycle, the requested address must be retained inside the interface by means of a register, as shown in **??**. The multiplexer selects the input address if the cache is ready to receive the address in the cycle the request is sent, otherwise it selects the registered address in the following cycles.

4.3.2 Control FSM! and timing

The control unit of this block is a simple Mealy **FSM!** (**??**) that after reset waits for a read request from the **IFU!**, then checks if the instruction cache is ready to receive an address and finally waits for a valid cache line to be read.

Read requests can be accepted both in the idle `WAIT_REQ` state and in the `WAIT_DATA` state, in order to ensure maximum throughput by initiating the next memory access in the same clock cycle as the data handshake of the previous one.

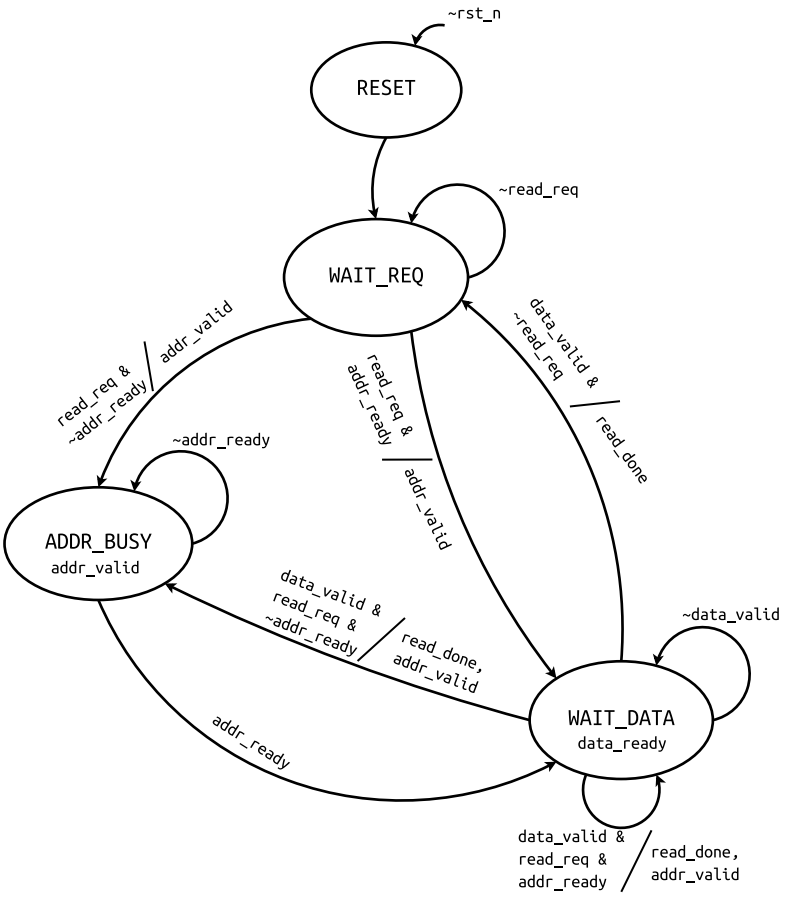


Figure 4.7: Instruction cache interface **FSM!**

?? shows a normal cache read, where the instruction cache is immediately ready to receive an address which hits and produces the requested data at the next clock cycle. From this timing diagram it is also clear why a Mealy **FSM!** is needed: the signal `addr_valid` needs to be asserted combinationally in the same clock cycle in which `read_req` arrives, so that the address handshake can take place immediately. Otherwise, with a Moore machine, one clock cycle would be wasted at each request, rendering impossible to sustain one instruction per clock cycle fetch. Another possibility would have been not to include such signal as a Mealy output of the machine and instead connect it with a wire outside the **FSM!**, which would lead to the same exact result, but was deemed as less readable.

?? shows another possibility when the instruction cache is not ready to receive an address at the time when a request arrives. In this case the **FSM!** waits until the cache become ready in the `ADDR_BUSY` state. The state machine could just as well wait in the `WAIT_REQ` state, but the additional state was introduced for robustness with `addr_valid` as a Moore output, so that this way there is no need

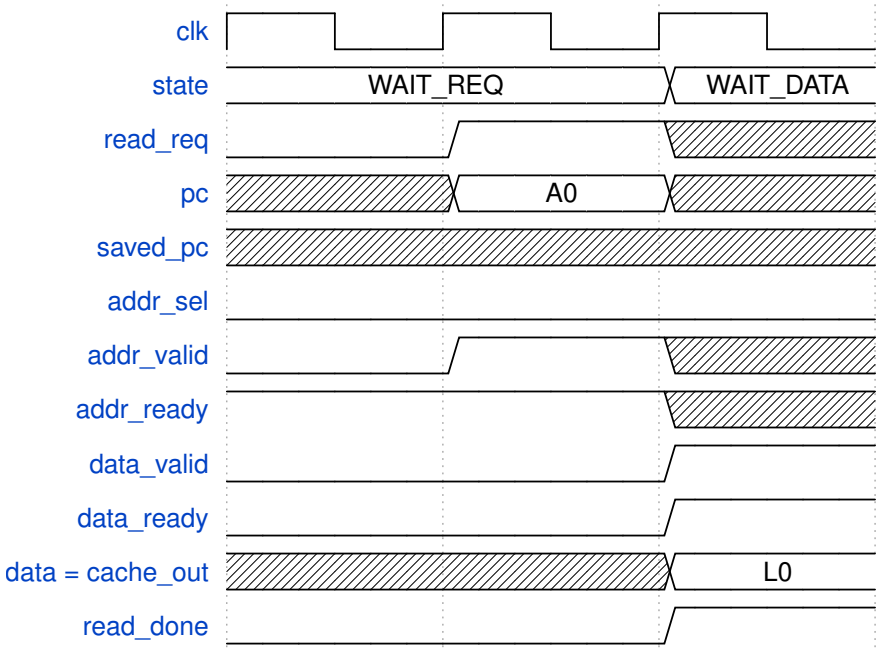


Figure 4.8: Normal cache read

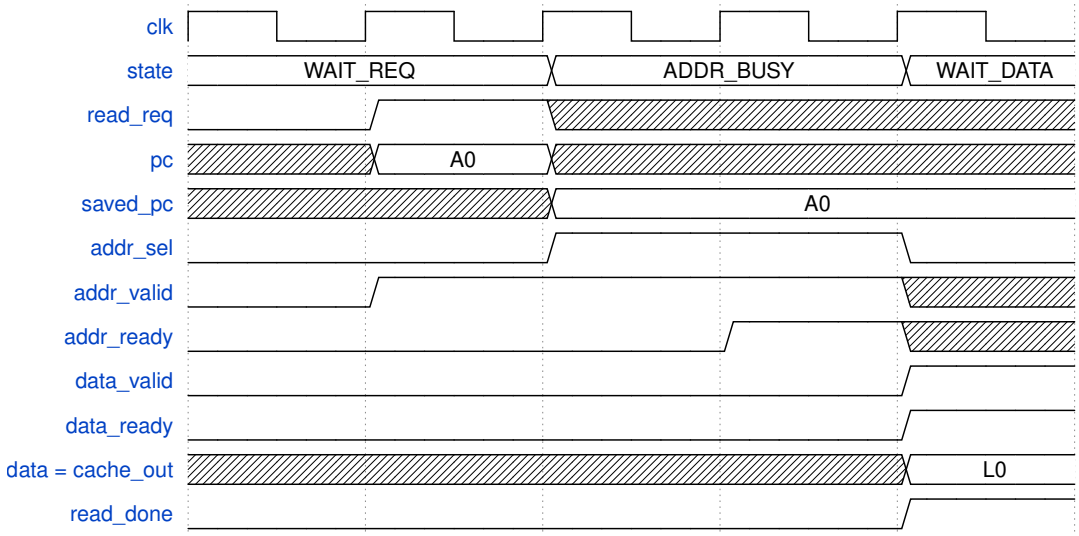


Figure 4.9: Cache not ready on address

for the `read_req` signal to stay active until the cache is ready. This is another point in favor of a Mealy **FSM!** instead of the connection of combinational outputs externally. Moreover, if the **FSM!** loops in this waiting state, the registered address is selected for the handshake, as the original program counter could have changed by that point.

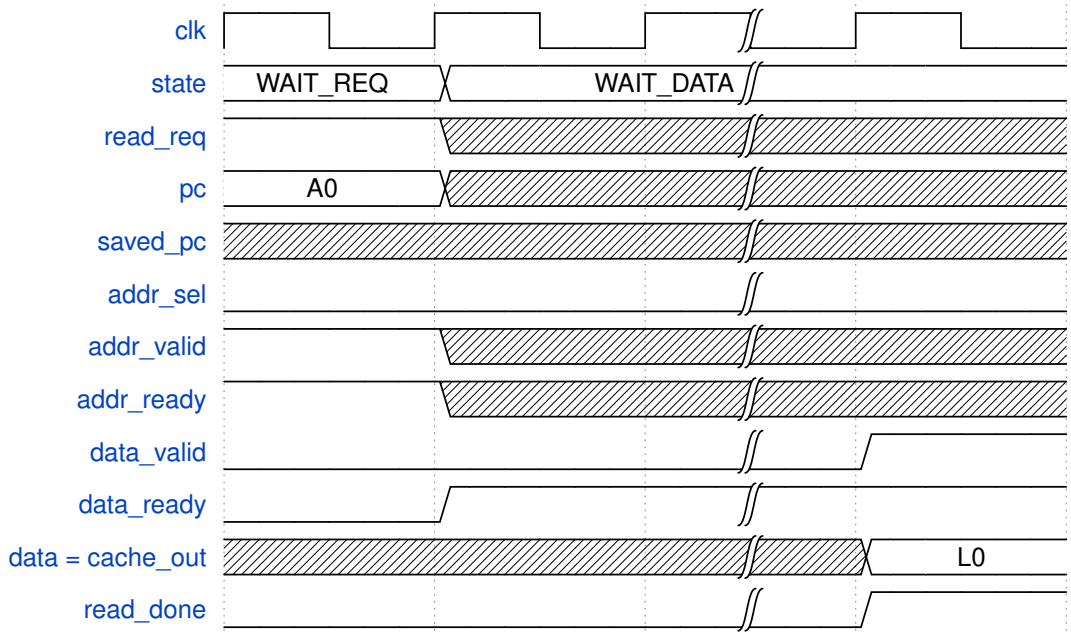


Figure 4.10: Cache miss

Finally, **??** shows the case of a cache miss, in which the **FSM!** waits while keeping `data_ready` asserted. This state could potentially last for many clock cycles.

From a description standpoint, this and all the following FSMs were coded in SystemVerilog using three different `always` blocks, one for the state update register, one for the state transition `case` and one for the output update `case`. This style, as evidenced in [**?**], leads both to less lines of code and better synthesis results in terms of timing and area.

4.4 IFU! (IFU!)

?? shows the top level diagram of the **IFU!**, which has the ability to fetch instructions from thee different locations. The first is the direct output of the instruction cache, from which instructions are taken in case of a memory access. Then, when a cache line containing multiple instructions is read, it is saved into a *line register* along with a valid bit that indicates that such line is valid. Consecutive instructions belonging to the same cache line are then fetched from this register, thus reducing the total number of cache requests. Finally, this line register is in turn saved into a *line backup register* every time that a fetch takes place (i.e. the register is not updated during a stall). This additional location is used every time that the current **PC!** requires a cache access, but the next address refers to an instruction that was present in the previously saved cache line. Without a backup, the line register would be overwritten by the line fetched at the current **PC!** and so the

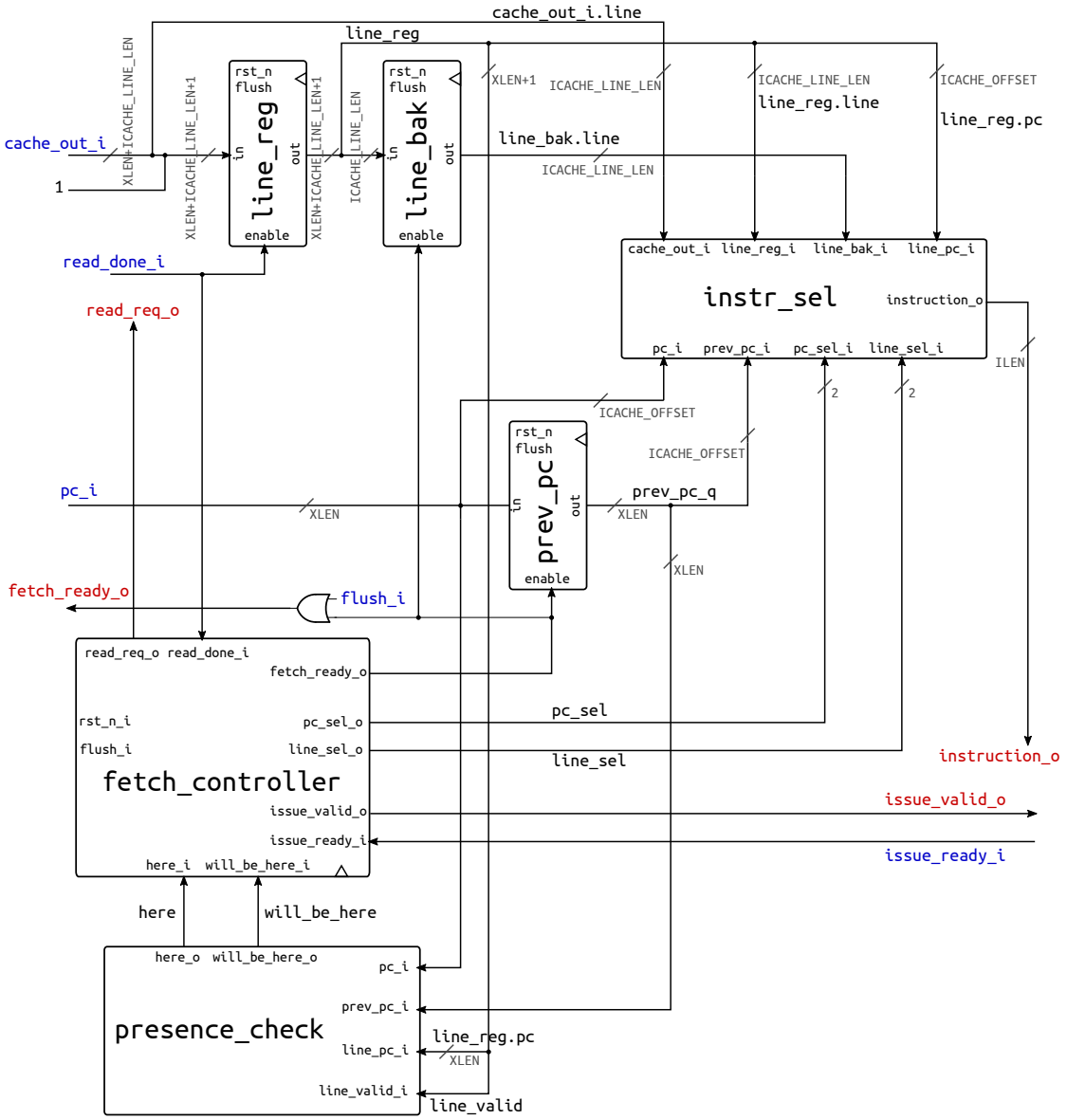


Figure 4.11: **IFU!** diagram (inputs `rst_n_i` and `flush_i` are shown as module port but are not connected with wires to avoid further clutter in the diagram, apart from the output OR gate)

next address would require a new cache read. Using an additional register, on the other hand, allows the **IFU!** to read the next instruction from the previous line. Evidently, this reasoning could be iterated to account for the second-oldest, third-oldest, etc. saved cache line, leading to a longer **FIFO!** of line registers among which to select the current instruction. This is definitely a possible improvement, but including more than two registers was judged out of the scope of the frontend.

An actual improvement should on the other hand come from the memory system, that could for instance include a *trace cache*, to account for subsequent instructions frequently fetched from different cache lines. Should such a feature be included, no other modifications would be needed on the **IFU!** end.

Two blocks, namely the *presence checker* and the *instruction selector* are responsible of informing the fetch controller if the current **PC!** points to an instruction already present in a saved line and of choosing the right source and the right instruction in the cache line respectively. The fetch controller itself is responsible of the orchestration of all the operations carried out inside the **IFU!** and of interfacing with the instruction cache interface as well as the **PC!** gen stage before and the issue queue after.

At the startup of the processor, the first fetch address will be the defined boot address which it is safe to assume that is going to need a cache access, as no line has been read and saved yet in the line register. This means that, even in the best case scenario (i.e. cache hit), the first instruction will be pushed to the issue queue one clock cycle after the corresponding **PC!** has entered the fetch stage, leading to a latency of a total of two clock cycles. In order to maintain the throughput of one instruction per clock, however, the **PC!** generation process must go on before knowing if the cache will hit or miss and that means that the first **PC!** must be saved in a *previous **PC!** register* in order to push the correct instruction to the queue at the next cycle. In other words, at each clock cycle, the **IFU!** is simultaneously checking whether the current **PC!** refers to a saved instruction or if a cache access is needed and pushing the previous instruction to the issue queue. Actually, if the current instruction is already present in a line register, then it could be potentially moved to the queue in the same cycle as no cache latency must be accounted for. This, however, would complicate significantly the timing of this unit, as the latency would be variable according to the need of a cache read or otherwise. For this reason, it has been chosen to maintain a one-cycle latency for every instruction, meaning that each instruction pushed to the queue corresponds to the **PC!** of the previous cycle. This is effectively an additional pipeline stage, introduced to account for the minimum cache latency and not to reduce the critical path.

Finally, in case of an exceptional behavior or a branch misprediction, the **IFU!** must be flushed and all pending cache requests aborted, in order to resume the process at the next cycle when the new starting **PC!** will be provided by the **PC!** gen stage. In order to do so, a `flush` signal that comes from the later execution stages or from the top-level control is propagated to all the sequential elements of the **IFU!**. This acts as a synchronous reset for all the registers and reverts back all the **FSM!**s to their startup state [**?**], by acting directly on the state register. For this latter case, the effect is the same of having a check on the flush signal in each state of the machine, but leaving it as an external signal similar to the initial reset simplifies significantly the state diagram and helps readability.

4.4.1 Presence checker

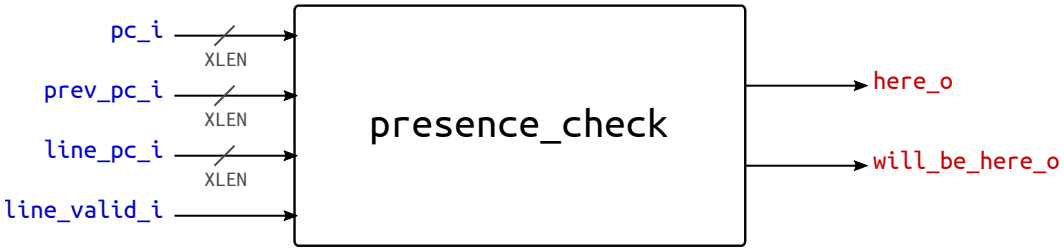


Figure 4.12: Presence checker module ports

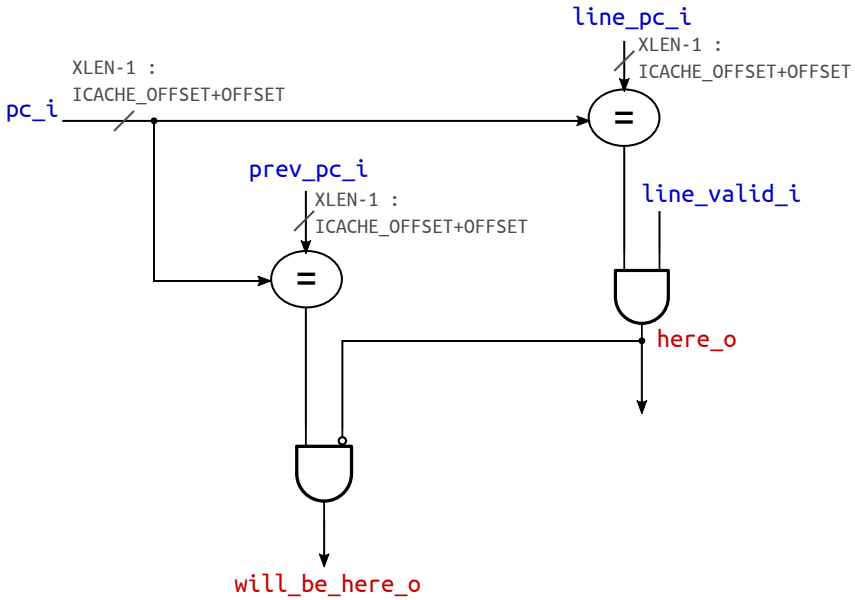


Figure 4.13: Presence checker combinational network

The presence checker block (see **????**) features a simple combinational network that performs two checks in parallel to determine the need of a new cache access by the **IFU!**, in particular:

- If the current address refers to an instruction present in the line register and the saved line is valid, then the `here` signal indicates that no new cache read is needed and that the instruction is to be selected inside the line register.

- If the current address refers to an instruction in the same cache line as the previous address, then either that line is already present in the line register, or it will be the next line read from the cache and so it will be saved as soon

as the read completes. In this case a `will_be_here` signal tells the **IFU!** not to request the same line twice to the memory.

If none of these signals are asserted, then a new cache request is sent to the interface.

All the instructions belonging to the same cache line have the same most significant parts of the address, that is, only the N LSBs differ, where:

$$N = \lceil \log_2(\text{Instructions/cache line}) \rceil + \lceil \log_2(\text{Bytes/instruction}) \rceil$$

So, to check whether two instructions belong to the same line, a comparison between the $64 - N$ (or more generally `XLEN` $- N$, where `XLEN` is the parallelism of the processor) most significant bits of their addresses is sufficient. That is what the presence checker does as shown in **??**.

4.4.2 Instruction selector

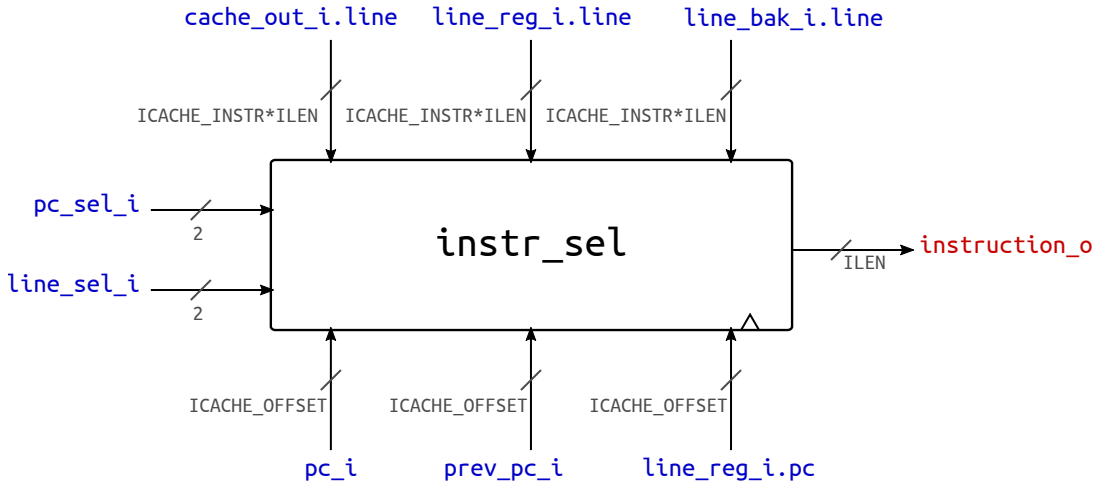


Figure 4.14: Instruction selector module ports

The instruction selector takes as inputs all the three sources an instruction can be fetched from, three program counter sources[3] and the respective selection signals (see **??**). It outputs a single selected instruction to be pushed to the issue queue.

?? shows the combinational selection network of the instruction selector, which basically consists of two multiplexers selecting the desired cache line and the address pointing to that line and another multiplexer choosing the selected instruction inside such line.

[3]The only source actually used, as stated above, is the previous **PC!**, but the others were included in the initial version of the design and have been kept should a future need arise. Given that the synthesizer can optimize unused wires, this choice incurs no overhead.

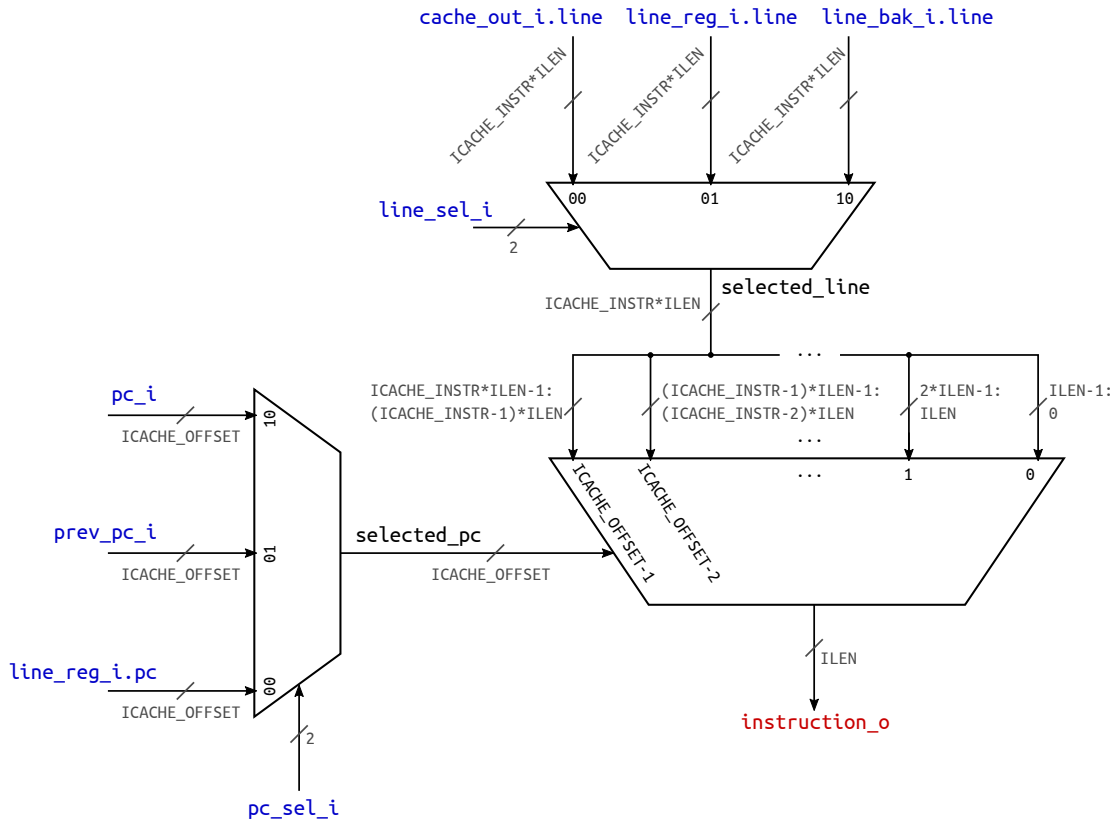


Figure 4.15: Selection network

According to the number of instructions stored in a single cache line, the output multiplexed can become quite large, nonetheless it should not be an issue in terms of total area.

4.4.3 Fetch controller

The fetch controller module, whose interface is shown in **??**, is the control unit of the **IFU!** which is responsible of receiving and generating control signals both for internal blocks and for interfacing with the other stages and the instruction cache. In particular, the controller receives information about saved instructions by the presence checker block (`here` and `will_be_here` signals) and as a consequence determines if a cache access is needed (`read_req` and `read_done` interface signals) and drives the correct selection signals (`pc_sel` and `line_sel`) to the instruction selector block.

Moreover, it handles the handshake signals `issue_valid` and `issue_ready` to and from the issue queue. The valid is asserted every time that the fetched instruction is available, as the result of a cache hit or saved in the line registers, while the

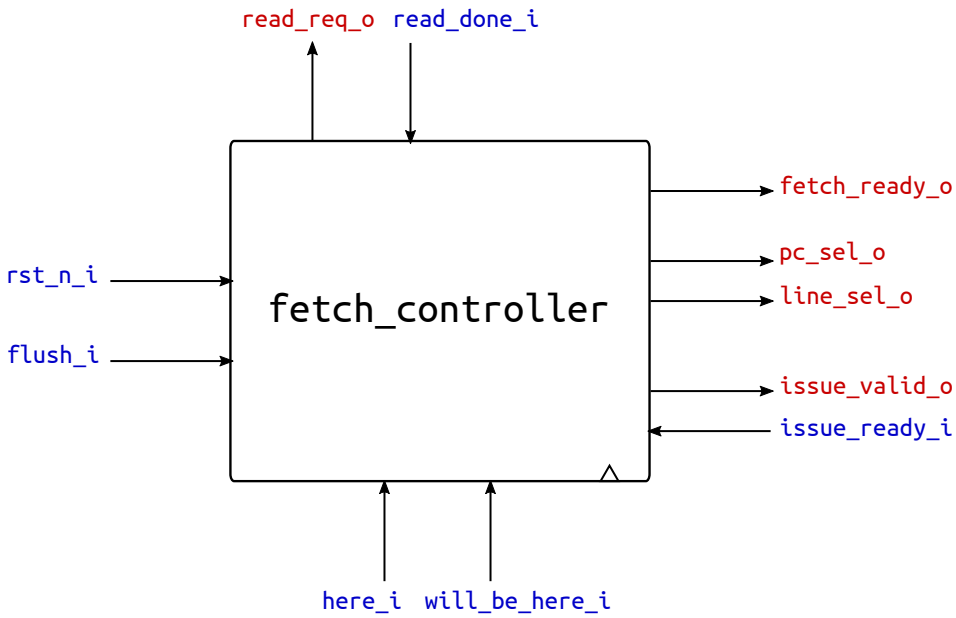


Figure 4.16: Fetch controller module ports

ready signals that the issue queue is not full and has at least available room for one more instruction.

Finally, the `fetch_ready` signal is used in the **PC!** gen stage as the enable of the output register (see **??**) and in the fetch stage as the enable of the previous **PC!** register (see **??**). This signal is deasserted in the case of a stall, that in particular can occur if the cache misses on the requested address or if the issue queue is full and cannot accept more instructions. Note that in case of flush, however, this signal remains active, because the flush operation resets all the data structures in one clock cycle and at the next the **IFU!** is again ready to fetch, so the **PC!** gen stage must provide the new valid start address.

Control unit

The main difficulties reside in high number of possible combinations of events that can occur simultaneously. For instance the issue queue could become full while a memory access is being completed, or on the other hand while an instruction is being selected in the line register. These different conditions must be handled in a per-case manner and that is what the Mealy machine of **??** manages. Here follows a summary of the purpose of each state:

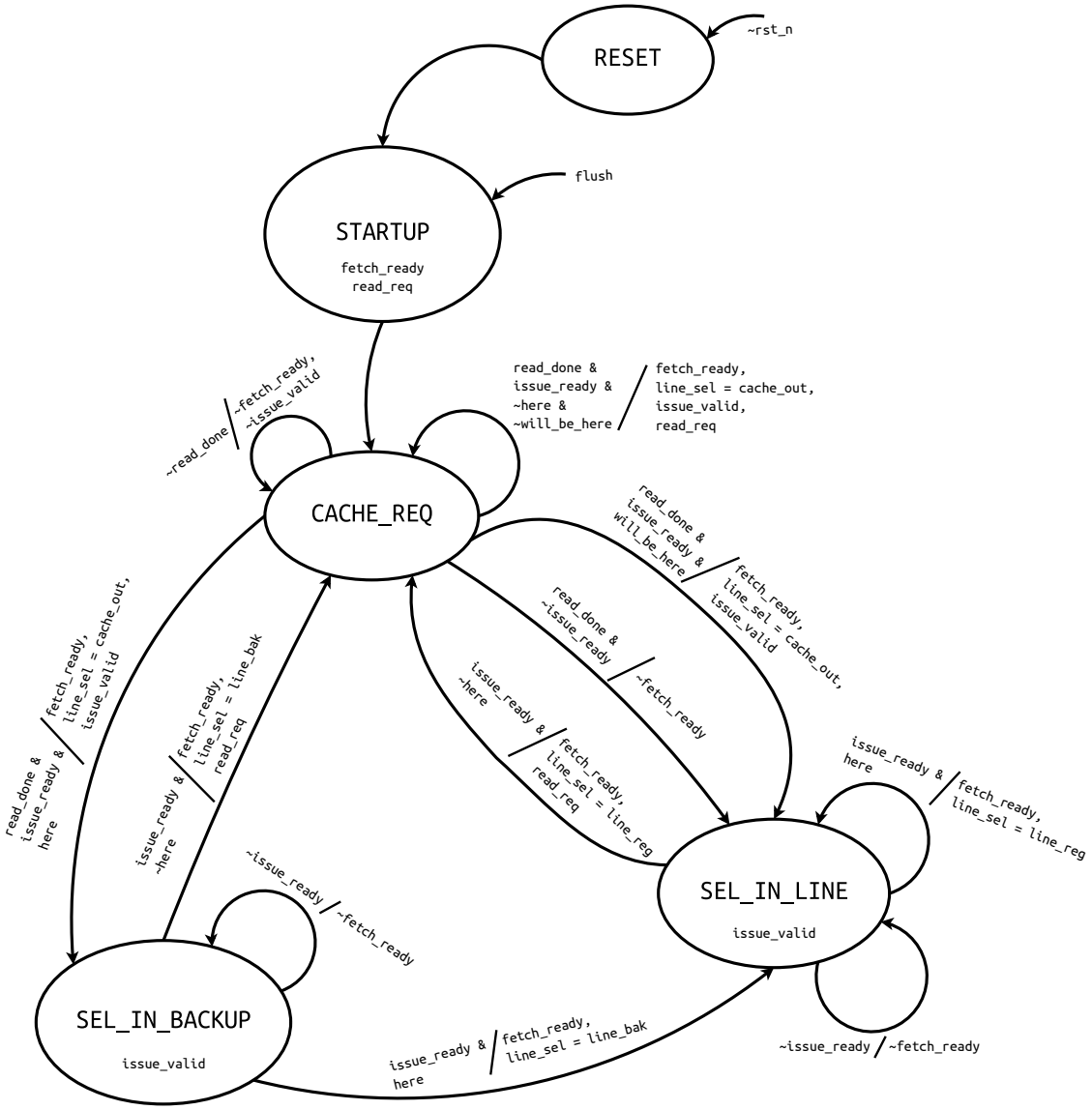


Figure 4.17: Fetch controller **FSM!**

- `STARTUP`: this state can occur only after a reset or a flush. In both cases, at that moment the **PC!** corresponds to the starting address and all the line registers have been reset to zero, so a new cache read is necessary and the `read_req` signal is asserted as a Moore output. The reason why the **PC!** is already valid and correct at this cycle is due to the fact that, in case of reset, the output register of the **PC!** gen stage is not reset to zero, but to the a constant `BOOT_ADDRESS` that initiates the program execution. In case of flush, on the other hand, the **PC!** is set to the next address in the following cycle with respect to when the flush signal arrives, thanks to the OR gate (see

??) that enables the **PC!** register in case of flush, even if the `fetch_ready` signal would have prevented it. The latter is the other Moore output in this state and is needed, as stated before, to ensure maximum throughput before knowing if the cache will have a hit or miss (i.e. the next **PC!** is generated nonetheless and in case of miss the **IFU!** is stalled at the following cycle).

- `CACHE_REQ`: in this state a memory access request is sent to the instruction cache interface. In case the cache is not ready to receive the address or incurs a miss, then the **FSM!** loops in this state until the `read_done` signals is asserted, stalling the frontend by deasserting `fetch_ready`.

  When the miss is resolved or immediately in case of hit, the state machine checks if the issue queue is ready. If it is, then the instruction is pushed and according to the output of the presence checker in that cycle, the next state transition is determined. If `here` is asserted, it means that the next instruction is present in the previously saved cache line, so at the next cycle, when the cache line just read will be saved in the line register and the old line register will be saved into the line backup register, the instruction will be selected from the backup register (move to `SEL_IN_BACKUP` state). If `will_be_here` is asserted, it means that the next instruction belongs to the same line that was just read, so it will be selected from the line register at the next cycle (move to `SEL_IN_LINE` state). If otherwise none of these signals are active, the next instruction belongs to a totally different cache line and so another memory access is necessary and the **FSM!** stays in this state.

  If the issue queue is full or busy, on the other hand, the cache output will be saved to the line register at the next cycle nonetheless, but the handshake with the queue does not take place and the state machine transitions to `SEL_IN_LINE` where it loops until the issue queue becomes ready again. During this time, the line register will not be updated anymore as the fetch is stalled and no new memory accesses can be performed.

- `SEL_IN_LINE`: as mentioned above, in case the issue queue is busy, the **FSM!** loops in this state waiting. On the contrary, if the queue is ready, an instruction is pushed and then the state machine remains in this state if `here` is active, signaling that the next instruction will be selected from the same saved line, or moves to `CACHE_REQ` otherwise to initiate a new cache request.

- `SEL_IN_BACKUP`: this state is the dual of `SEL_IN_LINE` meaning that the **FSM!** loops here when the issue queue is full and goes to `CACHE_REQ` if the next instruction is not saved anywhere, but with the difference that, if the `here` signal is asserted, the next state is `SEL_IN_LINE`, where the instruction will be selected in the line register.

For the same reasons stated before, it should be easy to understand how a Moore machine would not suit this particular control unit. As an example, with a Moore machine, from the conclusion of a cache read to the push of the selected instruction to the issue queue one clock cycle is bound to be wasted, which would hinder the throughput of the **IFU!**.

Timing diagrams

To better understand the behavior of the fetch controller **FSM!**, a list of timing diagrams covering a number of possible scenarios is now presented.

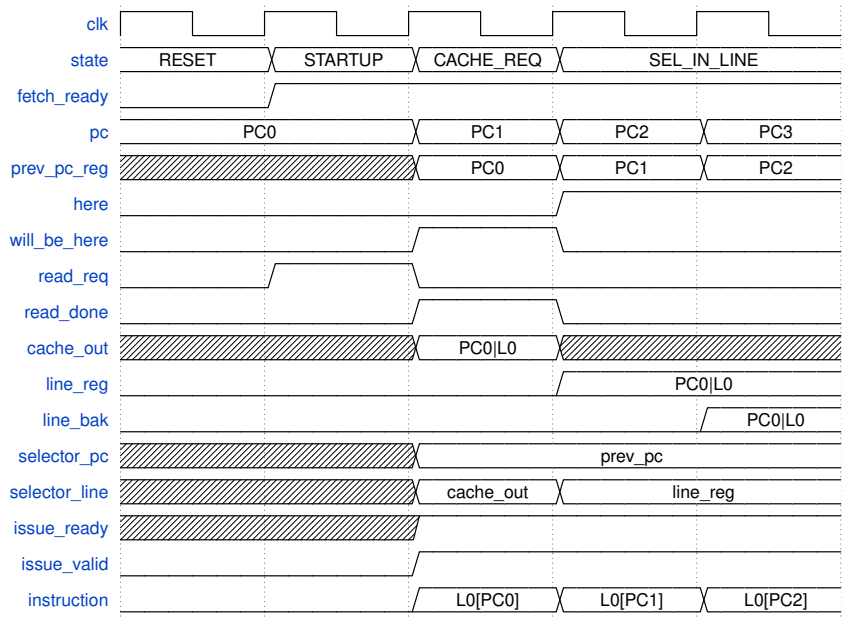


Figure 4.18: Startup and hit (`PC0` is the `BOOT_PC` considered above)

?? shows the boot up after the reset, where the first **PC!** needs a cache access and then the next instructions are fetched consecutively from the same line, now saved in the line register. This is the most ideal situation, with a cache hit and the issue queue ready. It is clear from this timing diagram that the `fetch_ready` signal must be active during `STARTUP` even if the outcome of the cache request is not yet known. If this were not the case, there would be a one-cycle penalty every time.

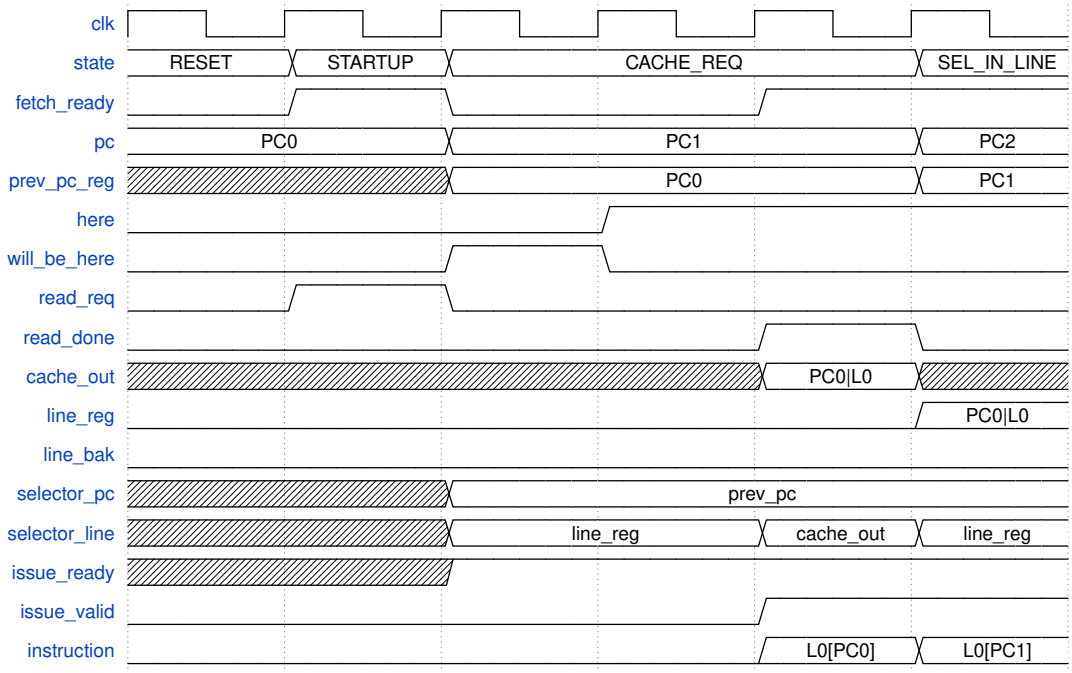


Figure 4.19: Startup and cache not ready/miss

?? shows what happens in the same situation if instead the cache is not ready or has a miss. This timing also explains why the instruction cache interface must save the address as soon as a read request is sent: if the address handshake does not occur at the second cycle when the request is made, then the address changes at the next clock and the read would happen at the new, wrong address.

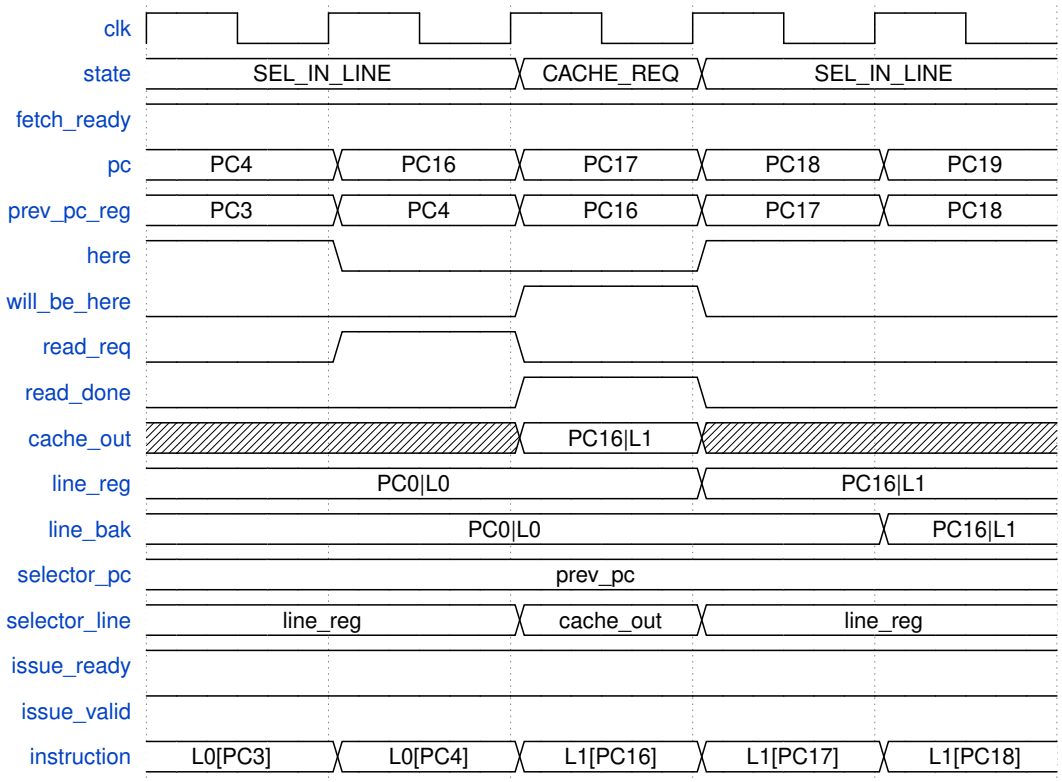


Figure 4.20: Saved line change

?? shows the situation of a cache line change. At first instructions are read consecutively from the line register, then a branch for example makes the **PC!** jump to a location stored on a different cache line, so a read request is sent, the cache hits and the current instruction is read from the cache output. After that, fetch continues sequentially with the other instructions in the new saved line.

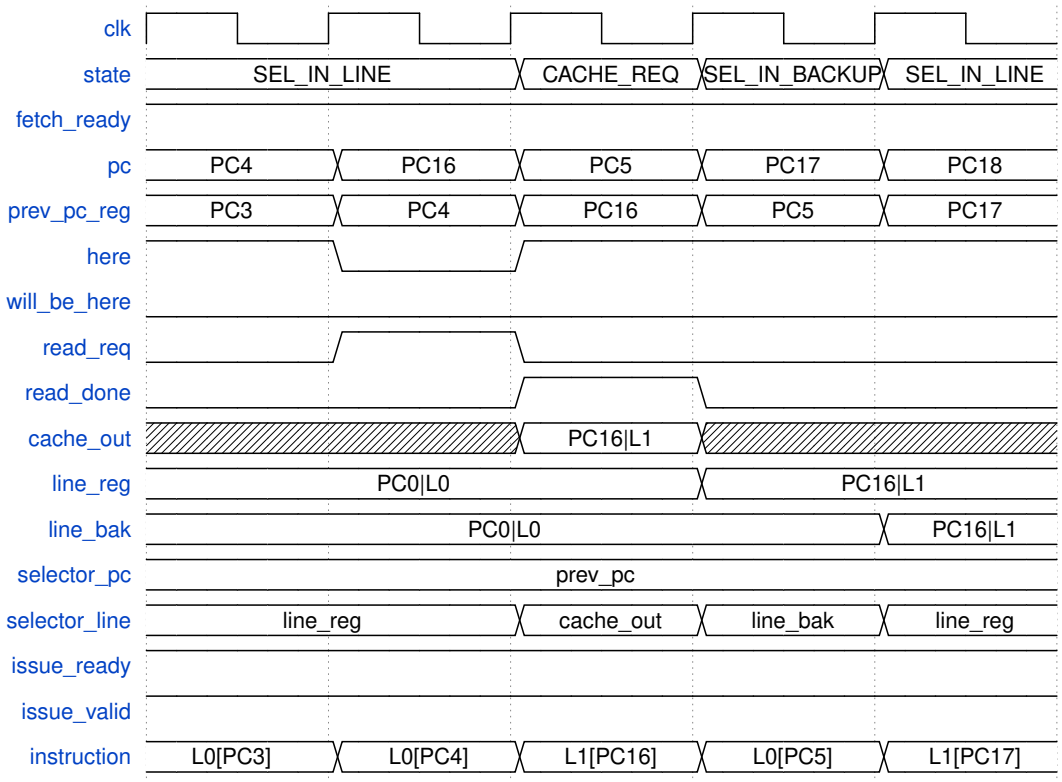


Figure 4.21: Line backup register purpose

The purpose of the line backup register is demonstrated in the timing of **??**, when there is a jump back and forth between two cache lines: the `here` signal during the cache request makes the next instruction be selected from the backup register. Of course, as already mentioned, the limitation of this solution is that if the same jump were to happen just right after this scenario (e.g. if `PC5` arrived again instead of `PC18` at the last shown cycle) even the backup register would have been overwritten and so a new memory access would be needed anyway.

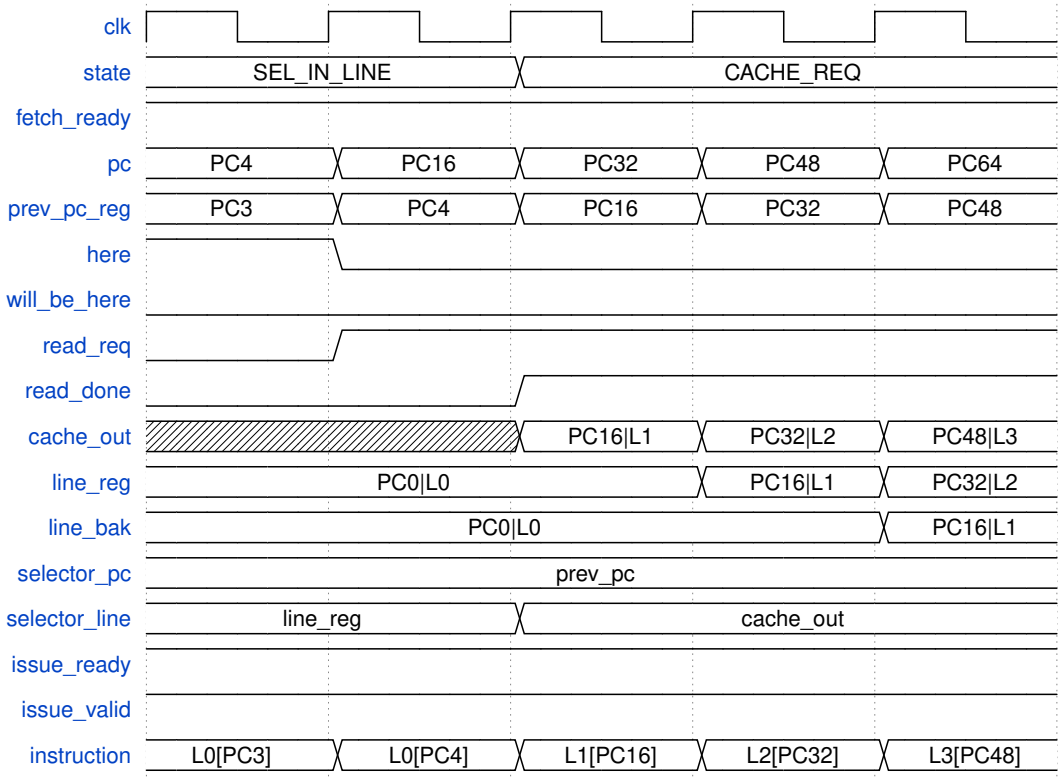


Figure 4.22: Cache read pipeline

?? illustrates the ability to reach a throughput of one instruction per clock cycle in a pipeline fashion even while reading from the instruction cache, if the memory hits on every address. In this case instructions are always selected directly from the cache output.

???? show the issue queue being not ready at two different instants. In **??** the stall happens when the instruction is selected inside the line register and in this case no problem arises, as the fetch stage is simply stalled and no register changes until the issue queue becomes ready again. If the queue is busy when an address would require a cache read, as in **??**, then the actual memory access is delayed until this stall is resolved. In both cases, the `fetch_ready` signal is deasserted to prevent the generation of new program counters.

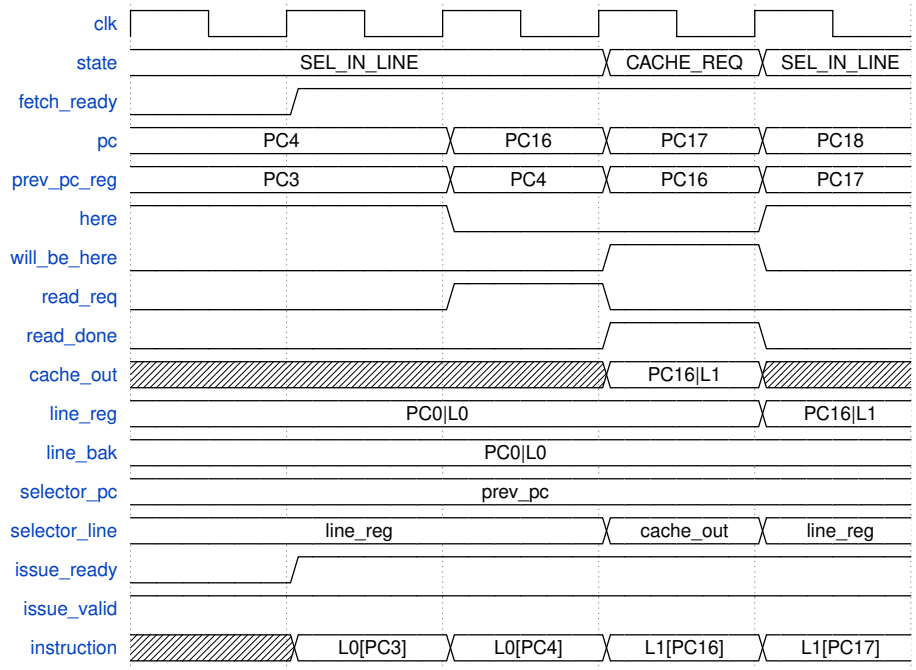


Figure 4.23: Issue queue not ready during selection

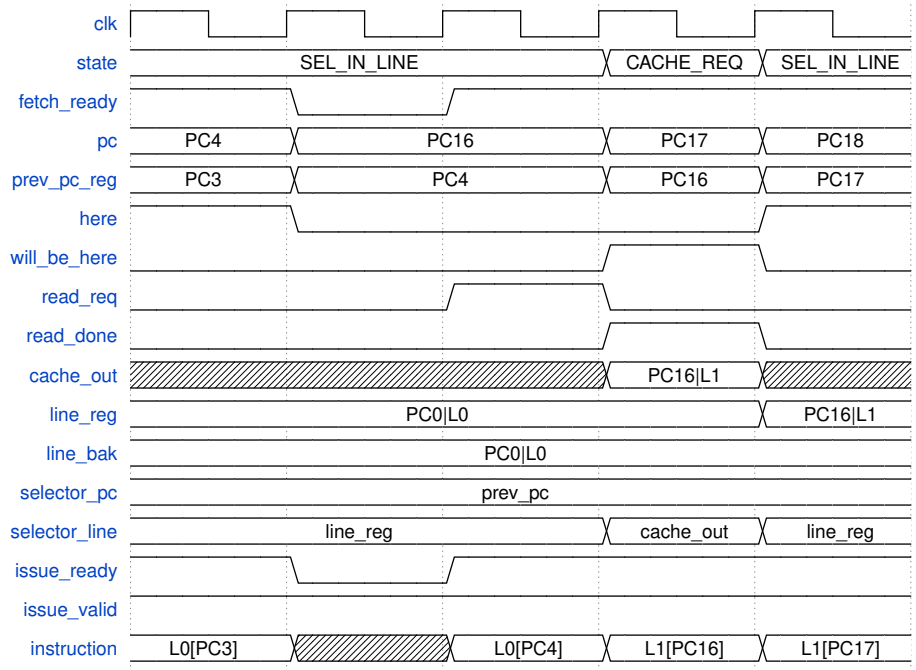


Figure 4.24: Issue queue not ready during request

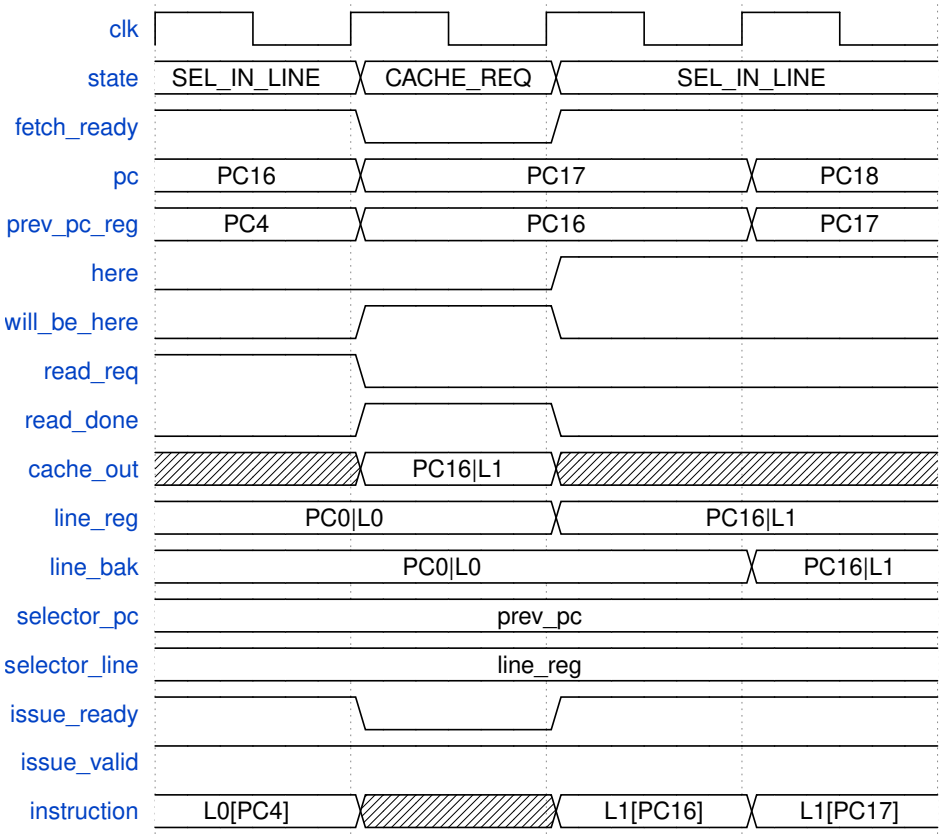


Figure 4.25: Issue queue not ready when cache line arrives

?? shows the case for which the issue queue is not ready when the cache hits and outputs a new line and the current address points to an instruction in that line (`will_be_here` asserted). In this case, there is a stall when no new **PC!** is generated and during which the newly read line is saved into the line register, so that when the issue queue becomes ready again the instruction will be selected from the register and not from the cache output.

The situation shown in **??** is similar to the previous one, but this time the issue queue is not ready when the output is a new line and the current instruction refers to a line previously saved. In this case not even the line backup register can manage it, because as soon as the fetch resume, it is updated with the content of the line register, that is the last cache line read. Thus, for the old line a new memory access is required.

Finally, **??** shows the last timing under analysis, that is the situation in which the issue is not ready when the instruction has to be selected inside the backup register. In similar manner to **??**, here the stall does not cause any issues and the fetch resumes exactly as before when the queue turns ready again.

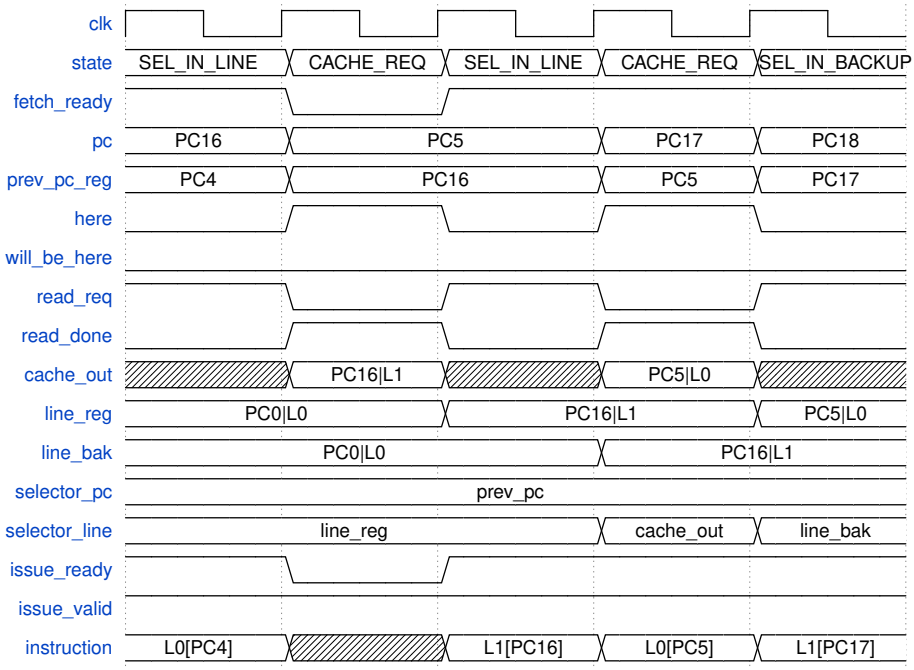


Figure 4.26: Issue queue not ready and causing loss of backup

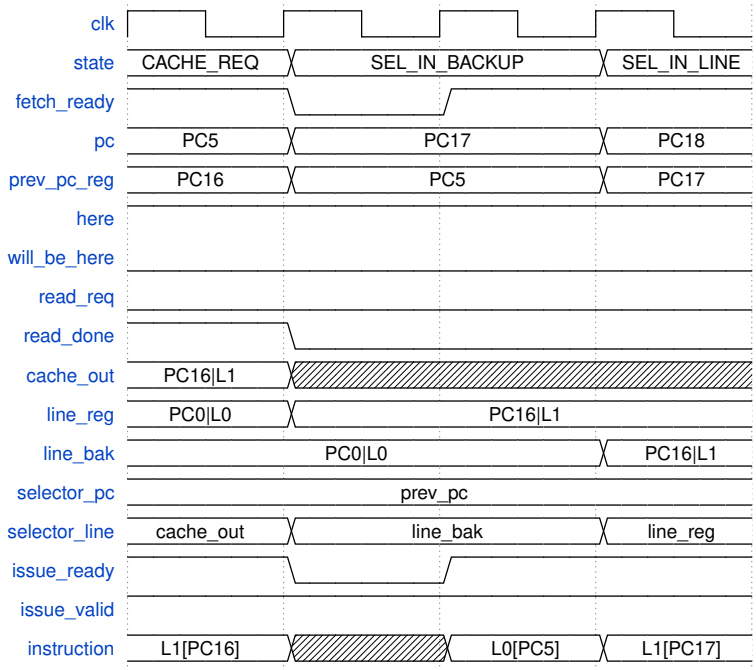


Figure 4.27: Issue queue not ready during backup

4.5 BPU! (BPU!)

As seen in **??**, the **BPU!** resides in the fetch stage and works in parallel with the **IFU!** on each address coming from the **PC!** gen stage. This unit, as shown in the high level scheme of **??**, is composed of two main blocks:

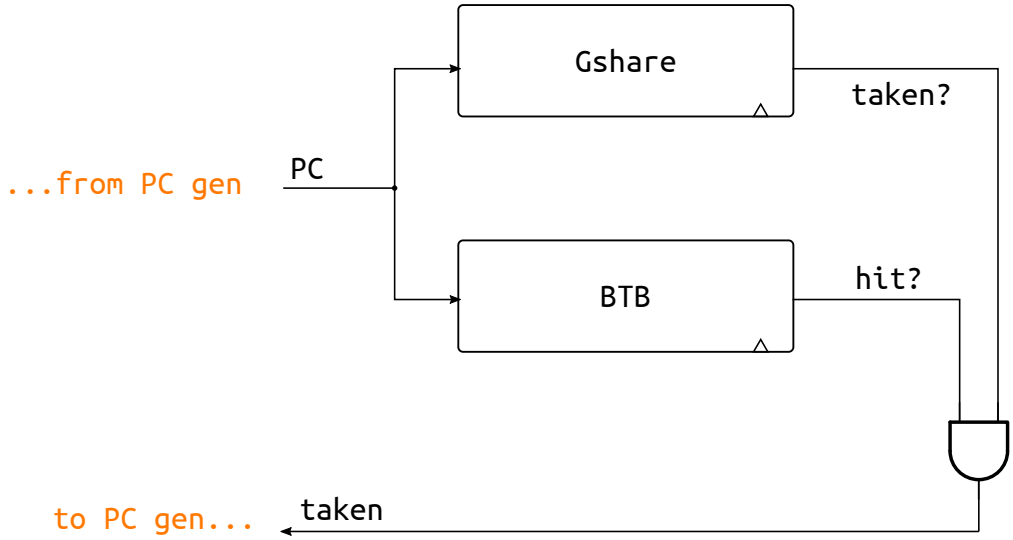


Figure 4.28: **BPU!** general idea

- The **gshare** is the actual branch predictor and is a variation of the two-level predictor described in **??** featuring a table of 2-bit counters and a global history register, of which a detailed explanation is provided in **??**. Being a branch predictor, its output is the predicted direction of the branch. Note that it outputs a prediction on every address, even those who potentially do not correspond to branch instructions. That is where the second block comes into play.

- The **BTB! (BTB!)**, described in **??**, is a small cache that contains the target address of taken branches only. This provides the significant advantage of being able to fetch the instruction after the taken branch with no additional delay, leading to zero overhead branches if the target is correct.

A branch is predicted taken by the **BPU!** if and only if the gshare predictor supposes the branch is taken and the **BTB!** contains an entry with a valid target for that specific **PC!**.

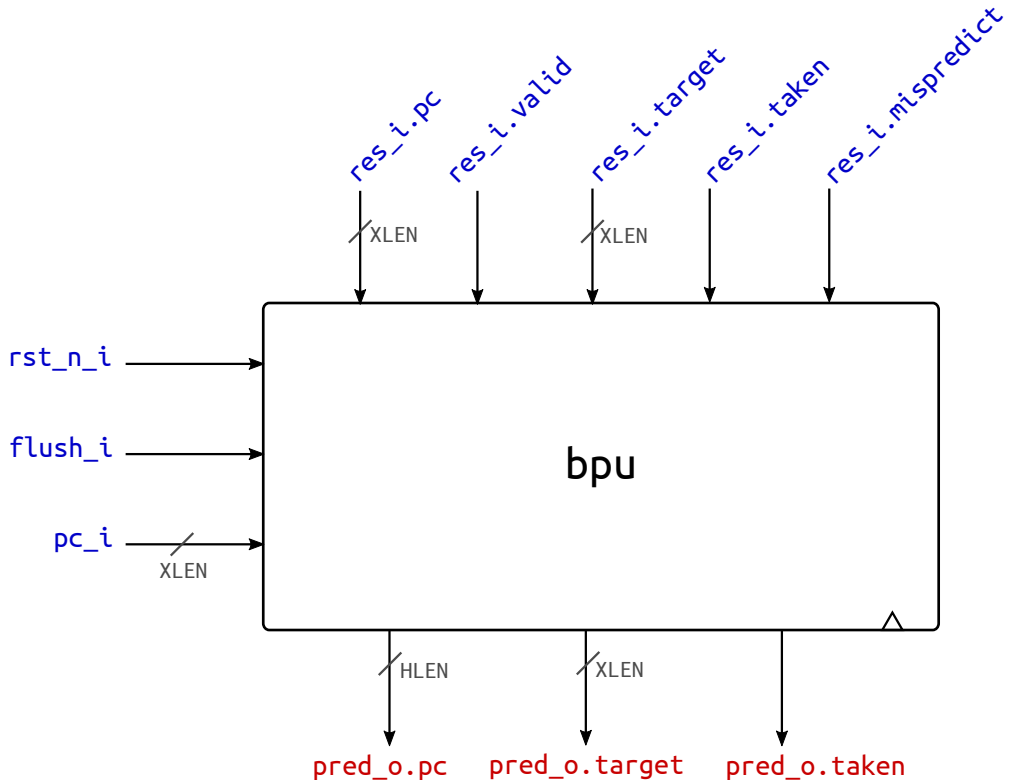


Figure 4.29: **BPU!** module ports

4.5.1 Top-level block diagram

?? shows the module interface of the **BPU!**. Apart from reset and flush signals, used respectively at startup and in case an exception requires the internal data structures to be flushed, the module outputs a prediction based on the current address, composed by a single bit indicating if the branch is supposed to be taken or not, the predicted target and also the branch address, which is later needed to update the predictor structures.

The inputs containing the `res` prefix, for resolution, are the just mentioned branch **PC!** to which the resolution refers, the actual target, the actual branch direction, a bit indicating if there was a misprediction and a valid bit to signal that the present one is a valid branch resolution coming from the execution stage.

This information, in particular the two bits about the actual branch direction and the misprediction, is used to update the data structures and take further action as listed in ??. In case of correct prediction, it is only needed to update the 2-bit counters inside the gshare predictor according to the branch direction. A misprediction, on the other hand, requires different actions to be taken depending on which event caused it. If the branch was correctly predicted taken, but the

Prediction	**Resolution**	**Target**	**Action**
Taken	Taken	Correct	Increment 2-bit counter
Taken	Taken	Incorrect	Increment 2-bit counter Update BTB entry Flush pipeline and restart from correct target
Taken	Not taken	–	Decrement 2-bit counter Remove BTB entry Flush pipeline and restart from branch **PC!**+4
Not taken	Not taken	–	Decrement 2-bit counter
Not taken	Taken	–	Increment 2-bit counter Add BTB entry Flush pipeline and restart from correct target

Table 4.1: Predictor update actions

target in the **BTB!** was incorrect, then the **BTB!** entry must be updated with the correct one. If, instead, the branch was mispredicted taken, the target buffer entry is deleted in order to prevent a potential further misprediction, in the case when the 2-bit counter was in the *strongly taken* state. Finally, if the branch was mispredicted not taken, then the corresponding target is added to the **BTB!**. This time, if the 2-bit counter was in the *strongly not taken* state, the second misprediction cannot be prevented and the same entry will be written once again in the **BTB!**.

In all the misprediction cases, the fetch and execution stages must be flushed and the next **PC!** must be set to the next sequential address after the branch **PC!** if the branch was actually not taken, or to the correct target if the branch was actually taken. Note that this flush operation is different from the flush caused by an exception because, obviously, it does not reset the branch predictor structures, which are only updated with the correct branch result.

?? shows the internal organization of the **BPU!** which is simply composed of the gshare and **BTB!** blocks, along with the AND gate of **??** and a couple of other gates to decide the correct update action based on the results, as in **??**.

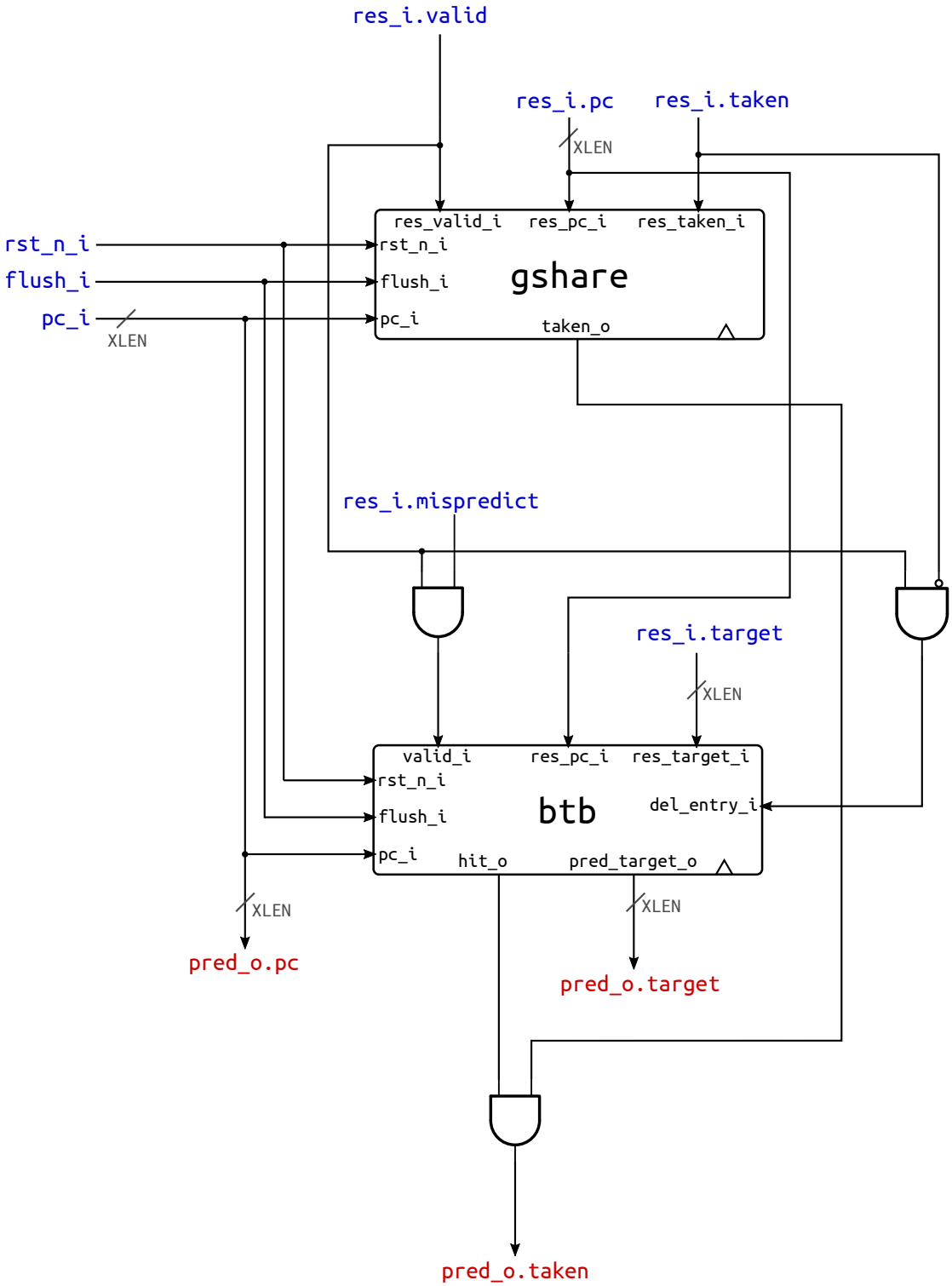


Figure 4.30: **BPU!** block diagram

4.5.2 Gshare branch predictor

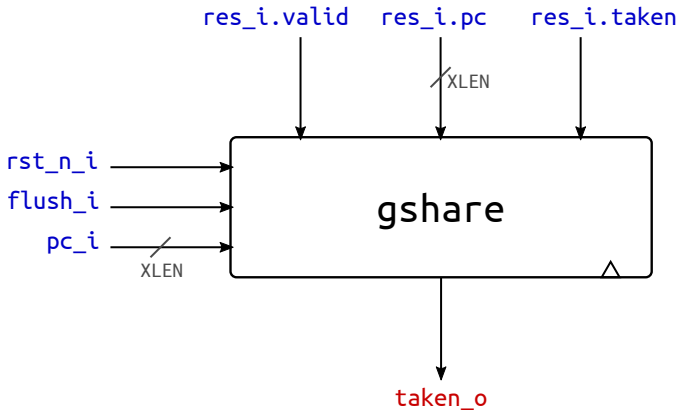


Figure 4.31: Gshare module ports

The idea of the gshare predictor, whose interface is shown in **??**, was first proposed by Scott McFarling in a 1993 paper [**?**] and consists in a variation of the two-level predictor that combines global information from the global branch history and local information of the current branch address by hashing them with the exclusive OR of the history register and the N least significant bits of the **PC!**, where N is the history length. The author noted that this hashed index contains more information useful to identify the current branch than each of the **PC!** and global history alone and the experimental results confirmed this thesis, by outperforming all previous variations of the two-level predictor.

The structure of this predictor is shown in **??** and is composed of a **BHT!** composed of a single shift register where branch resolutions are left shifted in and a **PHT!** that contains an array of 2-bit counters, organized as a register file, so providing synchronous write and asynchronous read. This table is indexed by the XOR of the history and the current **PC!**[4] in order to read the prediction for the current branch, which comes from the most significant bit of the 2-bit counter.

To update the **PHT!** when a branch is resolved the correct 2-bit counter has to be selected using the same index as the one used for the prediction. As an architectural choice, branches in LEN5 are resolved in-order during the execution stages, so that in the time span between the prediction and the resolution of a branch, no other resolution is generated and as such the branch history remains unchanged. This means that the write index used to select the 2-bit counter to be updated can be derived using the resolved branch address and the same current value of the history register. This poses no timing issues, as demonstrated by **??**,

[4] Actually, the 2 LSBs of the **PC!** are excluded as are always zero for 32-bit instruction.

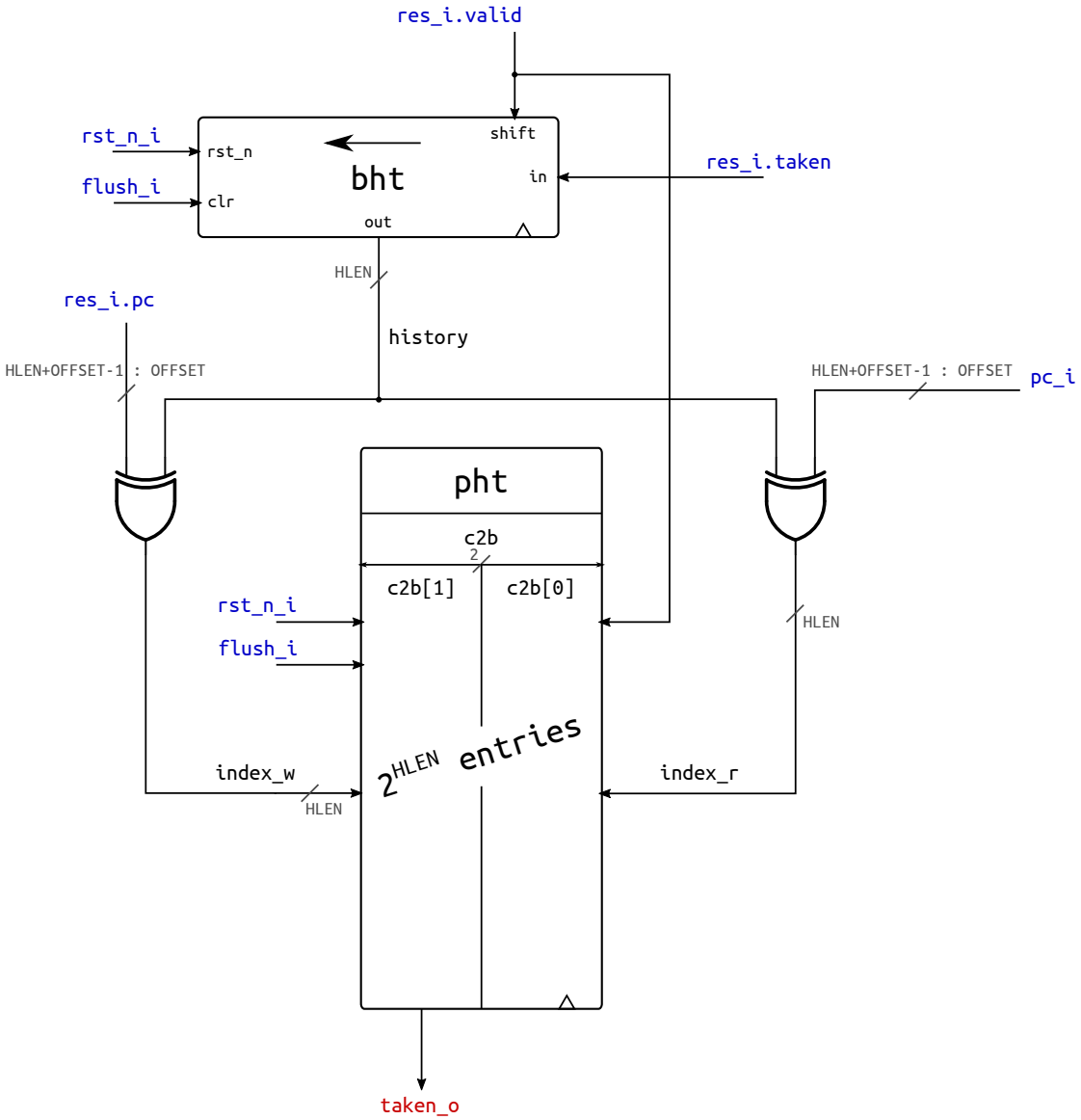


Figure 4.32: Gshare branch predictor

because the history register is shifted only after the clock edge when the `res_valid` signal is active, so at that edge the register still stores the unshifted value and the write operation on the **PHT!** occurs at the correct address.

If branches were resolved out-of-order, on the other hand, to restore the correct address to the **PHT!** the index at the moment of the prediction would need to be passed along in the pipeline to return at the moment of the resolution. Resolving branches in-order, on the other hand, not only allows a significant size reduction of data structures like the issue queue and the branch reservation stations by not

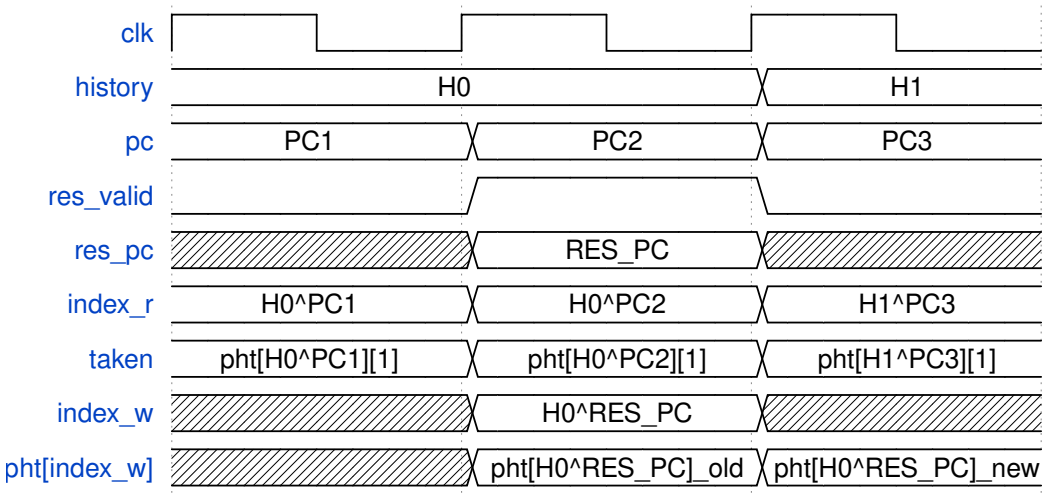


Figure 4.33: Update timing diagram

storing the prediction index, but also simplifies a great deal the instruction commit stage.

The initial version of the design also accounted for a column of valid bits in the **PHT!** to indicate if the prediction coming from each 2-bit counter is valid (i.e. if that 2-bit counter has already been used at least once). This is useful to prevent taken misprediction if the 2-bit counters are initialized in the weakly or strongly taken state. However, if the counters are instead initialized in a not taken state, then the first time they are accessed the prediction will always be "not taken" just as if the valid bit was included, so this bit becomes redundant and only wastes area. After a more thorough analysis, presented in **??**, it was decided to remove the valid bits from the **PHT!**.

4.5.3 BTB! (BTB!)

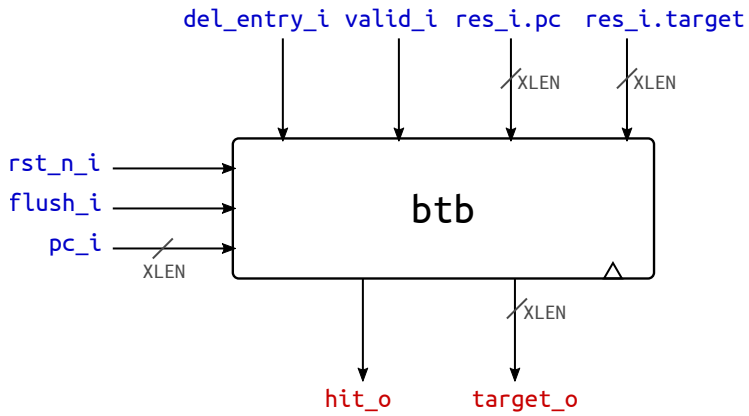


Figure 4.34: **BTB!** module ports

The purpose of the **BTB!** (interface in **??**) is to provide the target of a predicted taken branch in order to eliminate the delay of the computation of the destination address in the later execution stages and allow the fetch to proceed immediately from the new address. The idea of a **BTB!** was first introduced in [**?**] and was since optimized and employed in a great number of processor designs.

Its structure shown in **??** is basically that of a direct mapped cache with $2^{\texttt{BTB\_BITS}}$ entries addressed using the the `BTB_BITS` least significant bits of the **PC!**, as always excluding the offset, where each of them contains the following fields:

- **Valid**: 1-bit field indicating if the selected entry is valid, that is if it has been previously written with the correct target of a resolved branch.

- **Tag**: this field contains the remaining bits of the branch **PC!** not used to address the **BTB!**. The final evaluation on the branch address is a hit only if for the selected location the valid bit is set and the stored tag corresponds to the upper part of the **PC!**. This is done in order to eliminate the aliasing phenomenon, for which multiple addresses could point to the same **BTB!** entry and thus produce incorrect results for the target.

- **Target**: the actual destination address of the branch. The 2 LSBs are not stored as are they are useless for 32-bit instructions, but can save a significant amount of area if the **BTB!** contains many entries.

When the `del_entry` signal is asserted all three fields of the addressed entry are reset to zero.

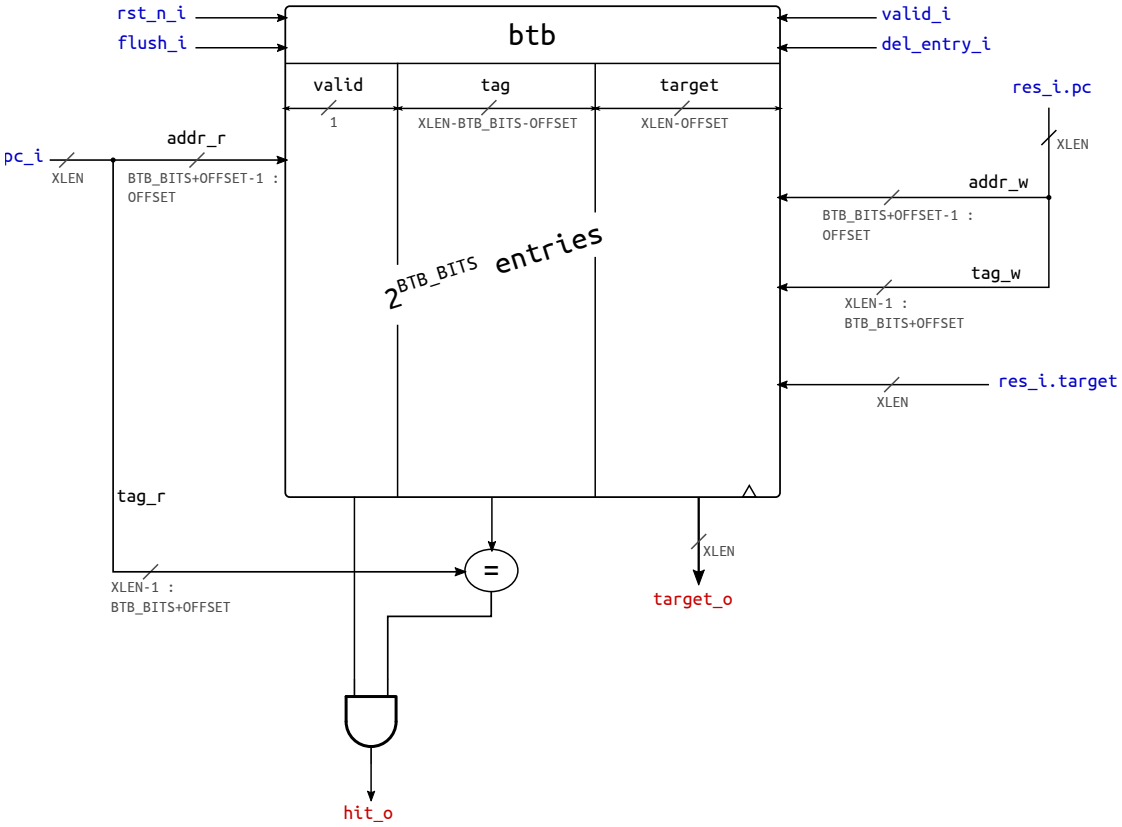


Figure 4.35: **BTB!** diagram

The **BTB!** is perhaps one of the blocks that can organized and optimized in the most number of ways, starting from the mapping of the cache to the information stored. Some variants of the design store the actual target instruction instead of the address to allow for some advanced techniques such as branch folding [**?**]. Being LEN5 an exploratory experiment on processor design, the choice has been made to keep the organization simple and so implement the **BTB!** as a direct mapped cache described as a register file.

For this reason, the same principles presented in **??** apply here, namely that the **BTB!** allows synchronous write and asynchronous read that in turn imply that a prediction and an update can be completed during the same cycle.

4.6 Branch unit

A unit that was not talked about until now and that is not shown in **??** because it resides in the backend is the branch unit, which is an execution unit responsible of determining the actual outcome of branch instructions. Its main tasks are the logic comparison between operand values in order to determine if the branch is taken or not and the sum of the base branch address and its immediate field to discover the target.

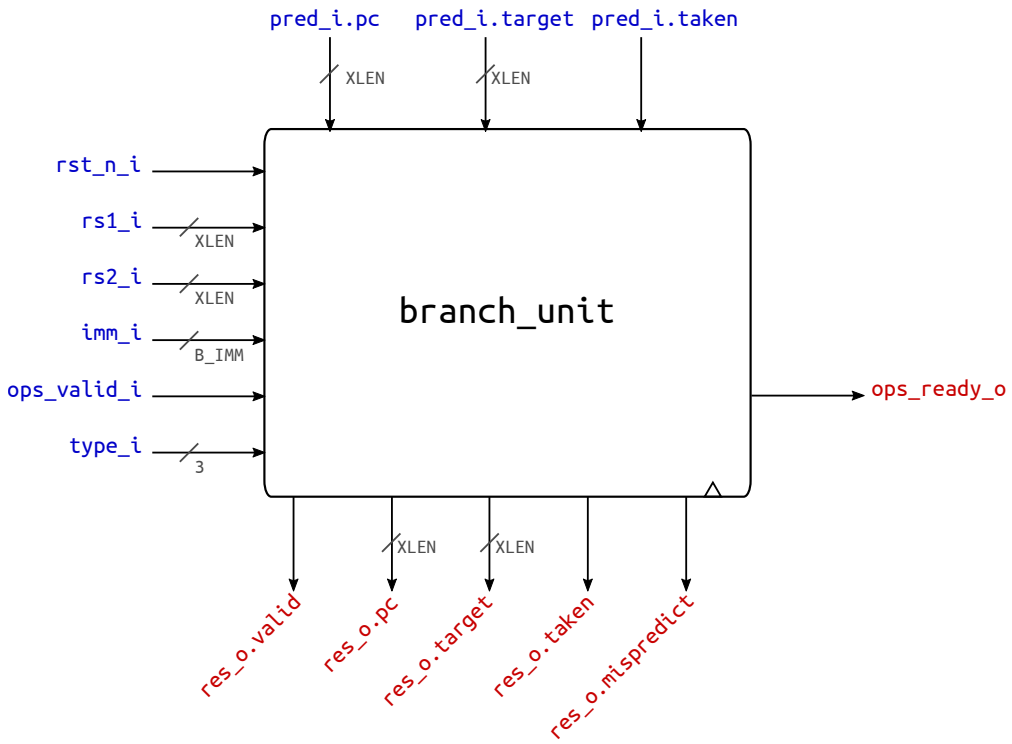


Figure 4.36: Branch unit module ports

?? shows the interface ports of this unit. It receives the register operands `rs1` and `rs2`, the immediate field `imm`, the branch type, encoded on three bits, and the prediction information from the corresponding reservation station, to and from which it communicates via the handshake signals `ops_valid` and `ops_ready` as usual.

Its outputs are the information on the branch resolution, which is passed forward to the commit stage and back to the frontend to update the **BPU!** and potentially stall in case of misprediction.

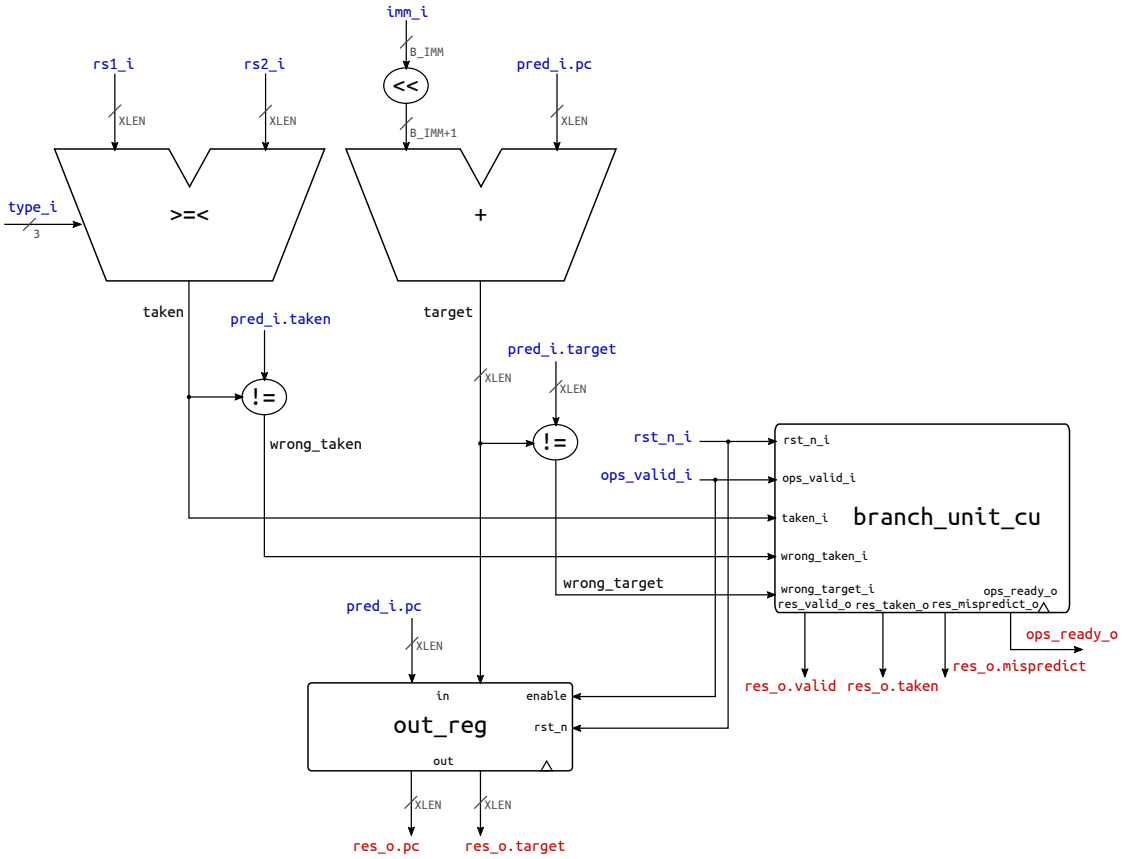


Figure 4.37: Branch unit datapath

4.6.1 Datapath

The datapath of the branch unit is shown in **??** and is composed of two ALUs, which are actually a separate logic unit and an adder, operating in parallel on the comparison and the target computation, which is performed, as per RISC-V specifications, by adding the base address with the immediate left shifted. The output of both units is compared with the predicted taken and target respectively and the results are passed to the control unit.

There are six types of branches in the RISC-V **ISA!**, namely:

- Branch if equal (`BEQ`)
- Branch if not equal (`BNE`)
- Branch if less than (`BLT`)
- Branch if greater than or equal (`BGE`)
- Branch if less than, unsigned (`BLTU`)

- Branch if greater than or equal (`BGEU`)

At the following clock cycle after the ALUs have obtained the results, the control unit outputs the correct prediction information and the branch **PC!** and right target are written to the output register.

4.6.2 Control unit

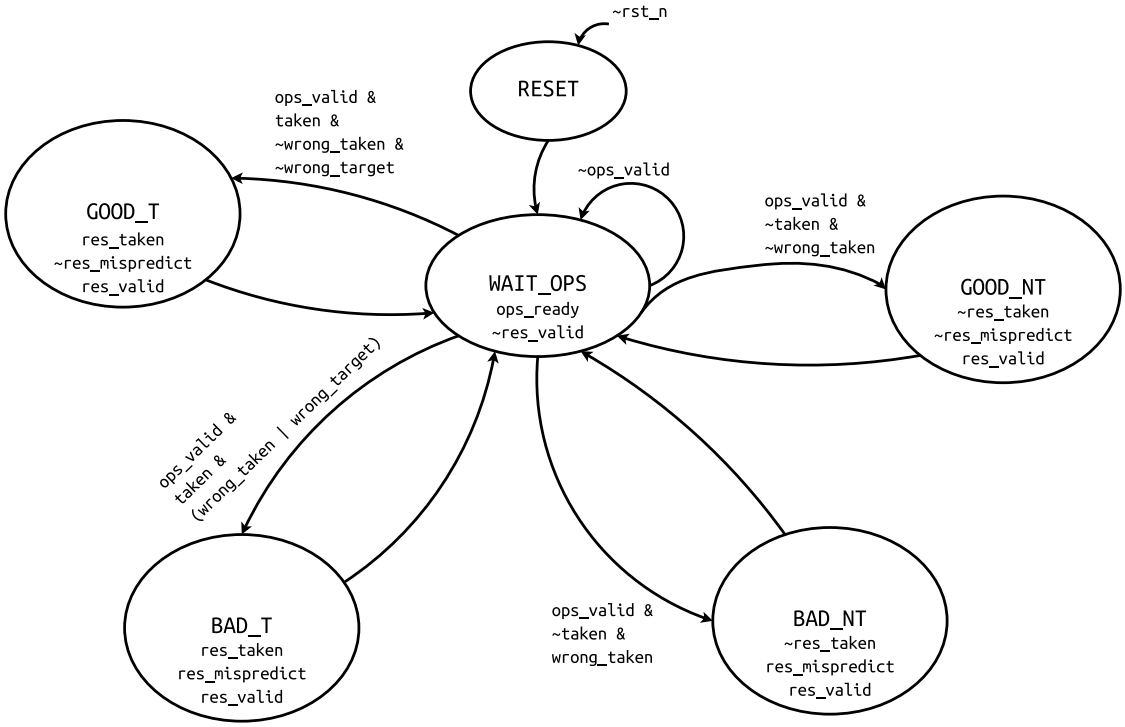


Figure 4.38: Branch unit control unit

The control unit of this module is a Moore machine that after reset starts waiting for valid operands from the reservation station in the `WAIT_OPS` state. When they arrive, in the same clock cycle the datapath performs the required computation, so that according to the value of the control signals, the **FSM!** can move to one of the states named `{GOOD,BAD}_{T,NT}`, based on whether there was misprediction and whether the branch was actually taken or not. In these states, the correct output signals are driven and then the state machine moves back to the waiting loop.

Add timing

Chapter 5

Experimental results

This chapter presents the results obtained from the design, starting from functional verification of the correctness of the implementation, to synthesis and benchmarking. The entire design has been described in SystemVerilog using hierarchical modules and behavioral constructs when possible. This both helps readability and leaves to the EDA tools the freedom of performing optimizations.

Each module has been verified locally using Verilator[1] for linting and Mentor ModelSim Intel FPGA Starter Edition for simulation. Synthesis has been carried out with Synopsys Design Compiler, on the the VLSI server provided by Politecnico di Torino.

5.1 Simulation

The choice of tools mentioned above was made because one of the pros of Verilator is that it has been found to be more verbose than ModelSim when performing syntax check and lint, thus reducing the number of possible issues during compilation and simulation. Moreover, it is completely free and open source, which nicely couples with the philosophy of the LEN5 project.

ModelSim was in the end chosen as the main simulation tool, because it is much more familiar to the designer and the limited time did not allow to learn Verilator for simulation, given that it uses a completely different paradigm, based on translating HDL to C language and using testbench templates in C as well.

The following sections focus on the test strategies used for the **BPU!**, which is almost as important as a standalone design, and the top-level frontend.

[1] https://www.veripool.org/wiki/verilator

5.1.1 BPU!

The testbench of the **BPU!** is based on reading branch addresses from one file, predicting the outcome and then comparing it with the correct branch resolutions coming from a second file. One clock cycle passes between the prediction and the resolution, as to simulate the execution delay, which always takes more cycles. During this time span, the **PC!** is fictionally increased to simulate a normal sequential fetch situation.

Given that the length of the history register in the gshare and the length of **BTB!** address lead to an exponential growth of the **PHT!** and the **BTB!** itself, simulations were performed using a small number of bits for these data structures and in particular the following results refer to a configuration with a 4-bit history register and a 4-bit **BTB!** address. The simulation is needed to verify the correctness of the design and not the prediction performances, so having fewer bits poses no issues.

In order to test the **BPU!** as a singular unit, without all the surrounding processor and in particular without a register file and execution units, some specific test cases have been defined, where branch results could be derived manually without actually executing the program. Loops, in particular, suit well such simple cases.

Single loop

The first test case corresponds to the following simple loop:

```
for (int i = 0; i < 10; i++)
{
  /* loop body */
}
```

Here, the loop condition is tested ten times as true, so the branch is taken, and the last time as false, so the branch is not taken.

Suppose that the instruction testing the loop condition is at address 10 and the beginning of the loop body (i.e. the target of the branch instruction) is at address 24. **??** shows the simulation waveforms for this test case, with the predictions contained in the `pred_o` signal, occurring each time the **PC!** 10 is read from the address file, and the resolutions read into `res_i` every time the `valid` signal is asserted.

Here, the initialization of the predictor structures can be clearly noted. Given that the history register is initialized to zero and the loop branch is always taken at first, the gshare predictor will initially update 2-bit counters which do not correspond to the actual branch history, until the **BHT!** is filled with ones (4 iterations, for the 4-bit register). Then the **PHT!** index will remain the same and so the same 2-bit counter is incremented from the initial strongly not taken state to the weakly taken state when it finally starts predicting correctly (2 iterations).

At the seventh iteration of the loop, the branch is predicted correctly as taken (the `mispredict` signal is deasserted) and this situation lasts until the the loop

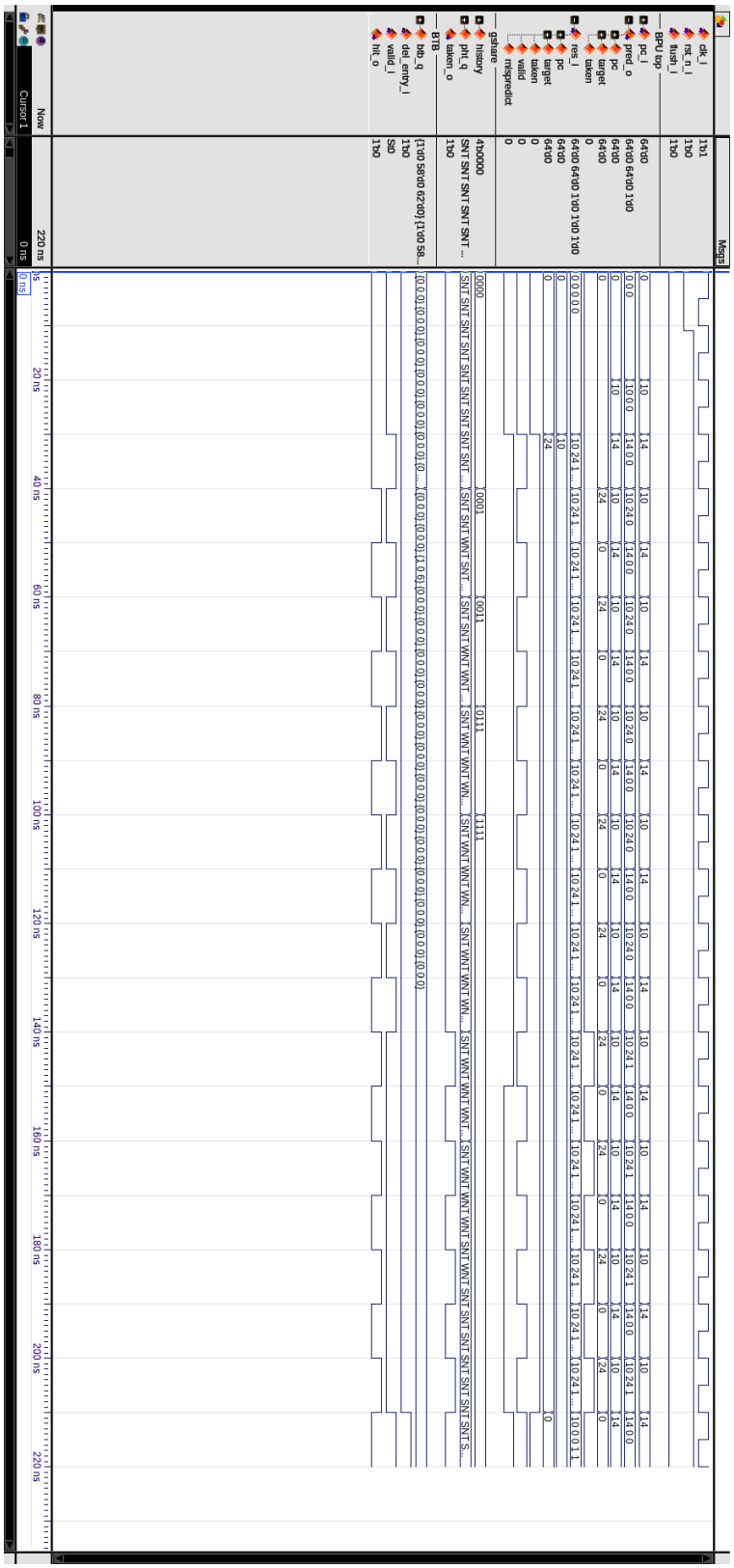


Figure 5.1: Single loop simulation

condition is tested false at the last iteration, leading to a new misprediction.

Meanwhile, the **BTB!** is updated with the correct target at the first iteration, from which it gets a hit each following time.

This *warm-up* of the predictor is intrinsic of its design and cannot be avoided, but anyway **??** demonstrates the correct and expected behavior of the **BPU!**.

Nested loops

Next, the case of two nested loops was tested, as in the following code:

```
for (int i = 0; i < 20; i++)
{
  for (int j = 0; j < 3; j++)
  {
    /* loop body */
  }
}
```

This example, actually taken from [**?**], is intended to demonstrate that the gshare predictor, after the warm-up, can correctly identify taken branches in nested loops, where the outer loop is repeated many times.

?? shows the simulation results for this case, where the address of the condition instruction of the outer loop is 10, the one of the inner loop (i.e. the target of the outer loop) is 80 and the body of the inner loop starts at address 24.

At the beginning, the gshare continuously mispredicts the outer loop and the first iterations of the inner loop, due to the initialization of the **PHT!** as mentioned in the previous case, but then after the warm-up it goes on to predict correctly both loops, until the exit of the outer loop. In particular, the steady state situation is shown in **??**, where each combination of address and history univocally determines the prediction outcome.

Value	Condition	PC!	History	Prediction
`j = 0`	`j < 3`	80	1101	Taken
`j = 1`	`j < 3`	80	1011	Taken
`j = 2`	`j < 3`	80	0111	Taken
`j = 3`	`j < 3`	80	1111	Not taken
`i = n`	`i < 20`	10	1110	Taken

Table 5.1: Nested loops steady state predictions

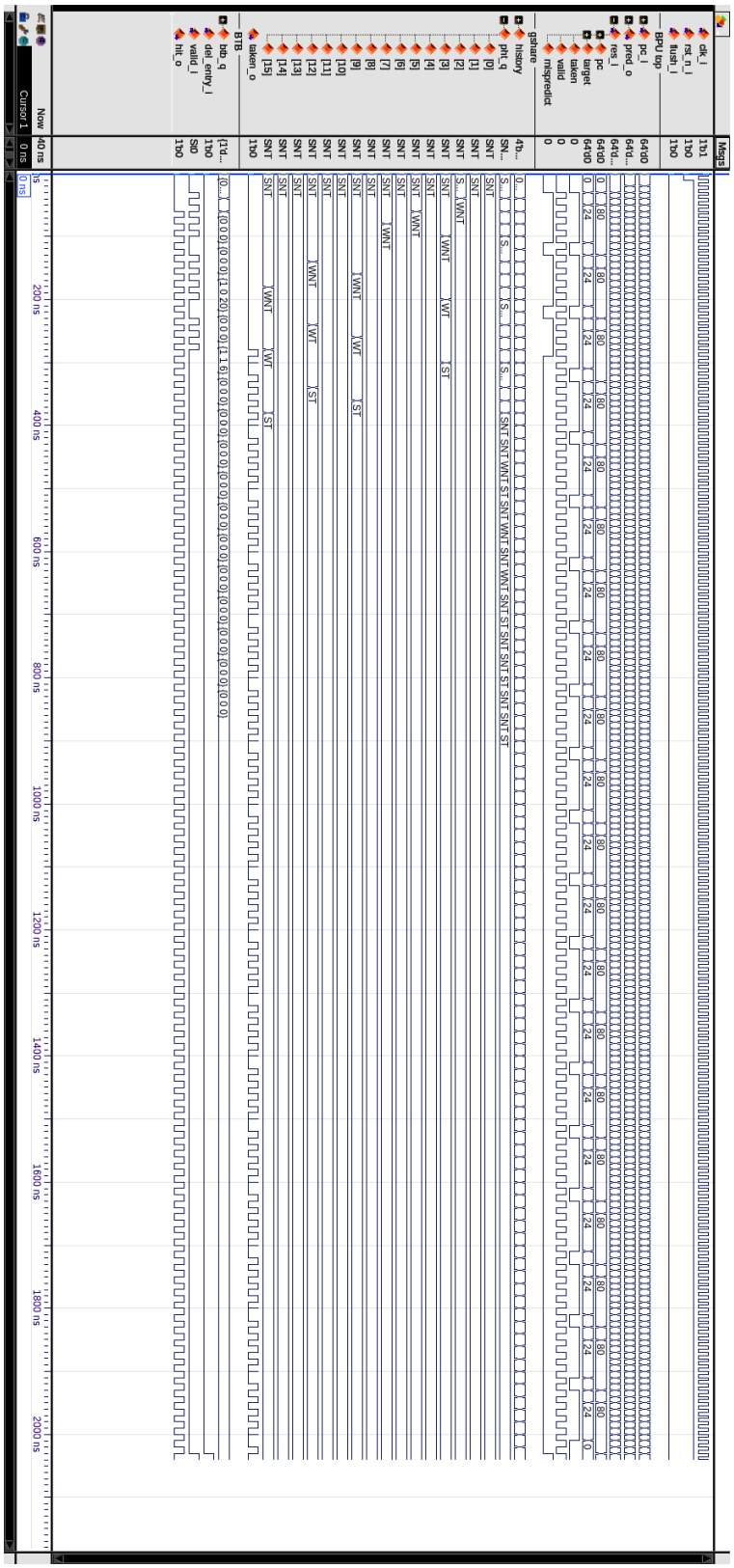


Figure 5.2: Nested loops simulation

5.1.2 Frontend

The testbench designed for the whole frontend is composed of the following blocks that drive its inputs:

- A ***PC!** jumper* used to simulate exceptions and branches by modifying the sequential generation of addresses.

- A *dummy instruction cache* which responds to memory access requests by simulating both hits and misses, with random delays. The fictional data line it provides always contains the **PC!** that generated the request and N instruction fields with the number from 1 to N in order to track the movement of instructions.

- A *dummy issue queue* that simply simulates a busy issue queue by introducing random delays on the `issue_ready` signal.

Using this setup, the frontend was simulated in a number of scenarios corresponding to the different situations analyzed in **??**, of which the most significant are described below.

?? shows the standard situation where subsequent instructions are selected among the same line, saved in the line register after the first memory access at startup (compare with **??**).

???? show the situation of a cache not ready to receive the address or a cache miss respectively, as in **??**. Note also the current states of the instruction cache interface **FSM!** that correspond to the timing diagrams of **????**[2].

?? shows a sequence of cache reads in a pipeline fashion, just as in **??**. Note also how here there is a jump right after the boot address, which is correctly handled by the instruction cache interface.

??, like **??**, show the case when the instruction is selected among the line backup register, due to a jump back and forth to the same cache line.

Finally as an example, **??** shows the situation in which the issue queue is not ready during instruction selection (compare with **??**).

[2]These simulation waveforms show `WAIT_ADDR` as the wrong old name for the `WAIT_REQ` state.

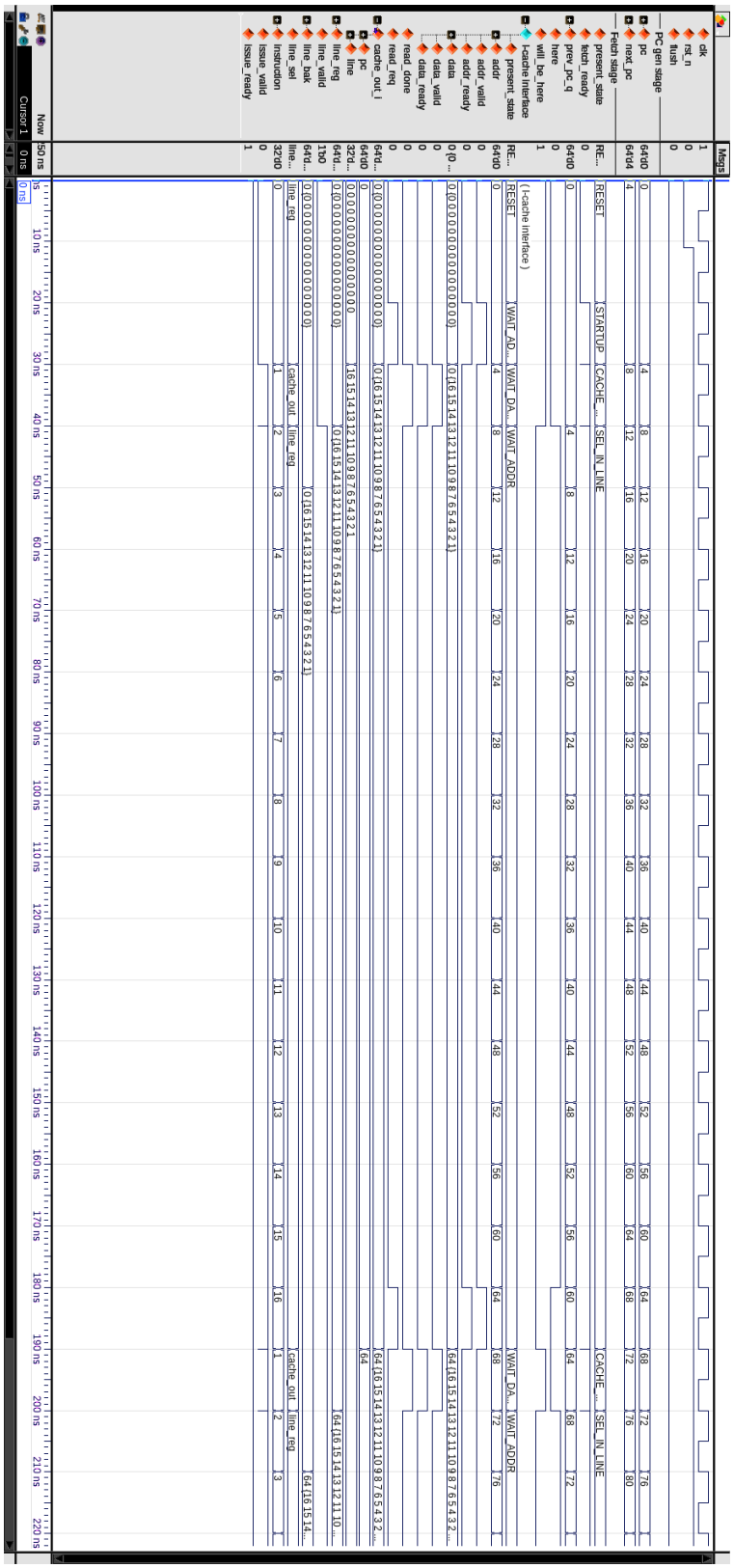


Figure 5.3: Sequential reads from the same line

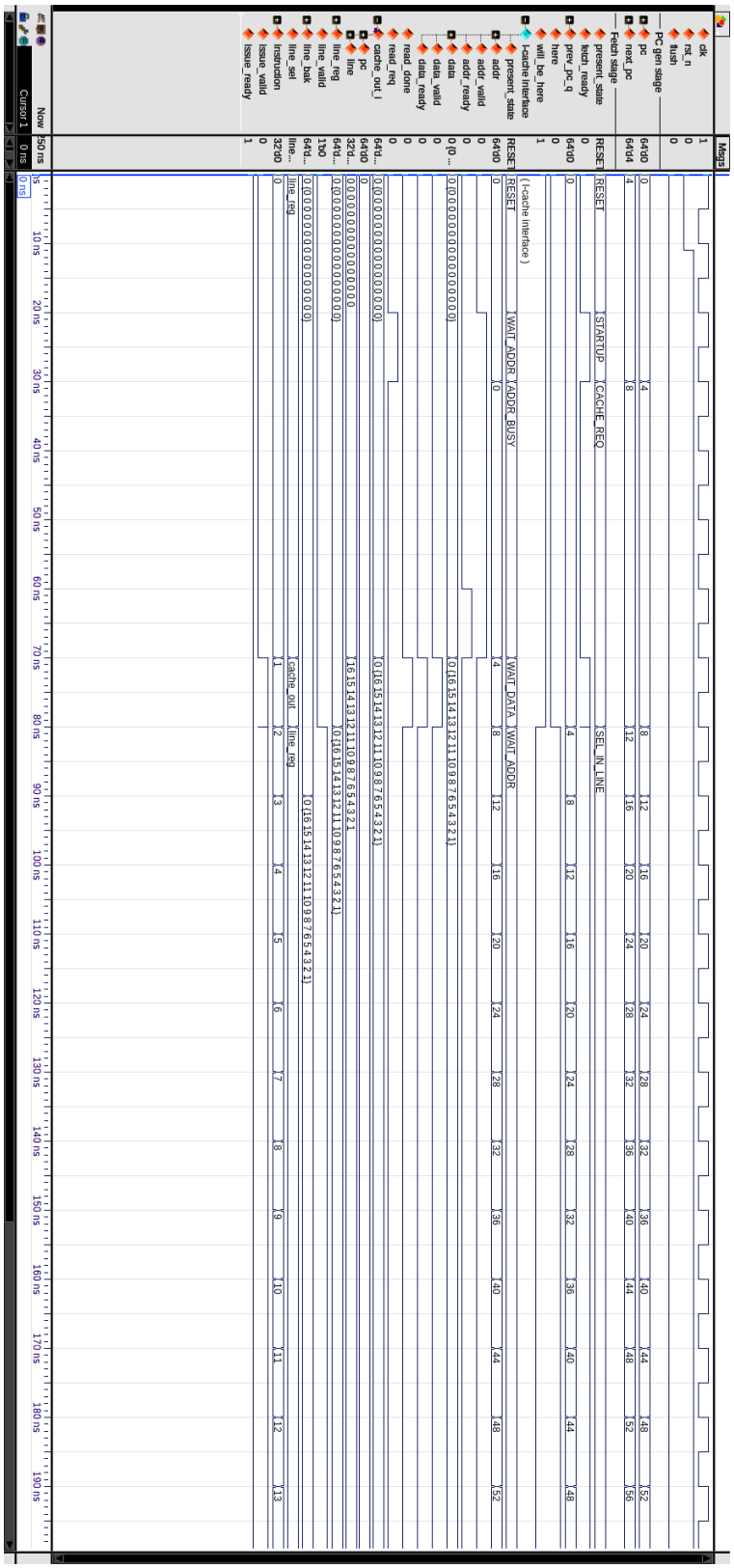


Figure 5.4: Cache not ready on address

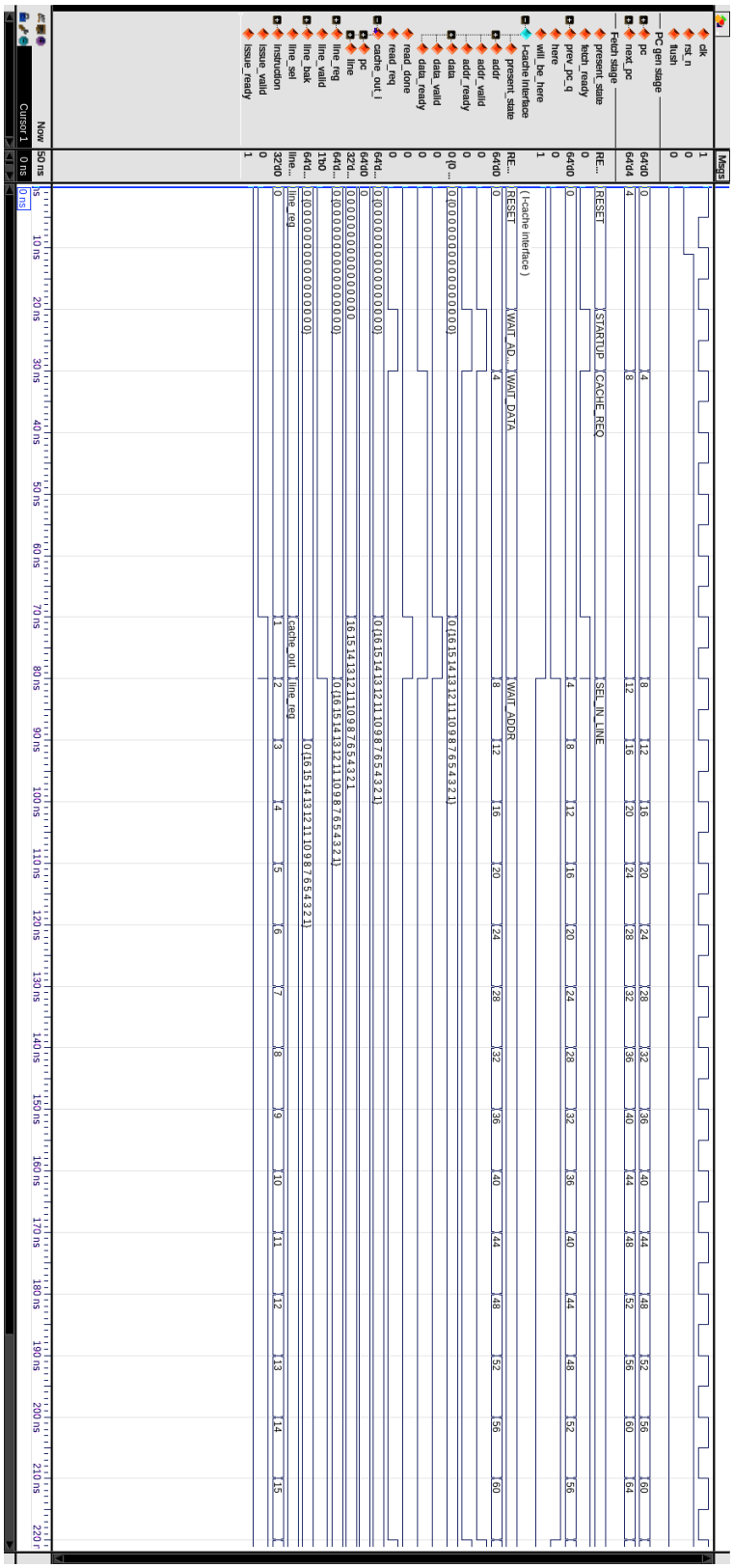


Figure 5.5: Cache miss

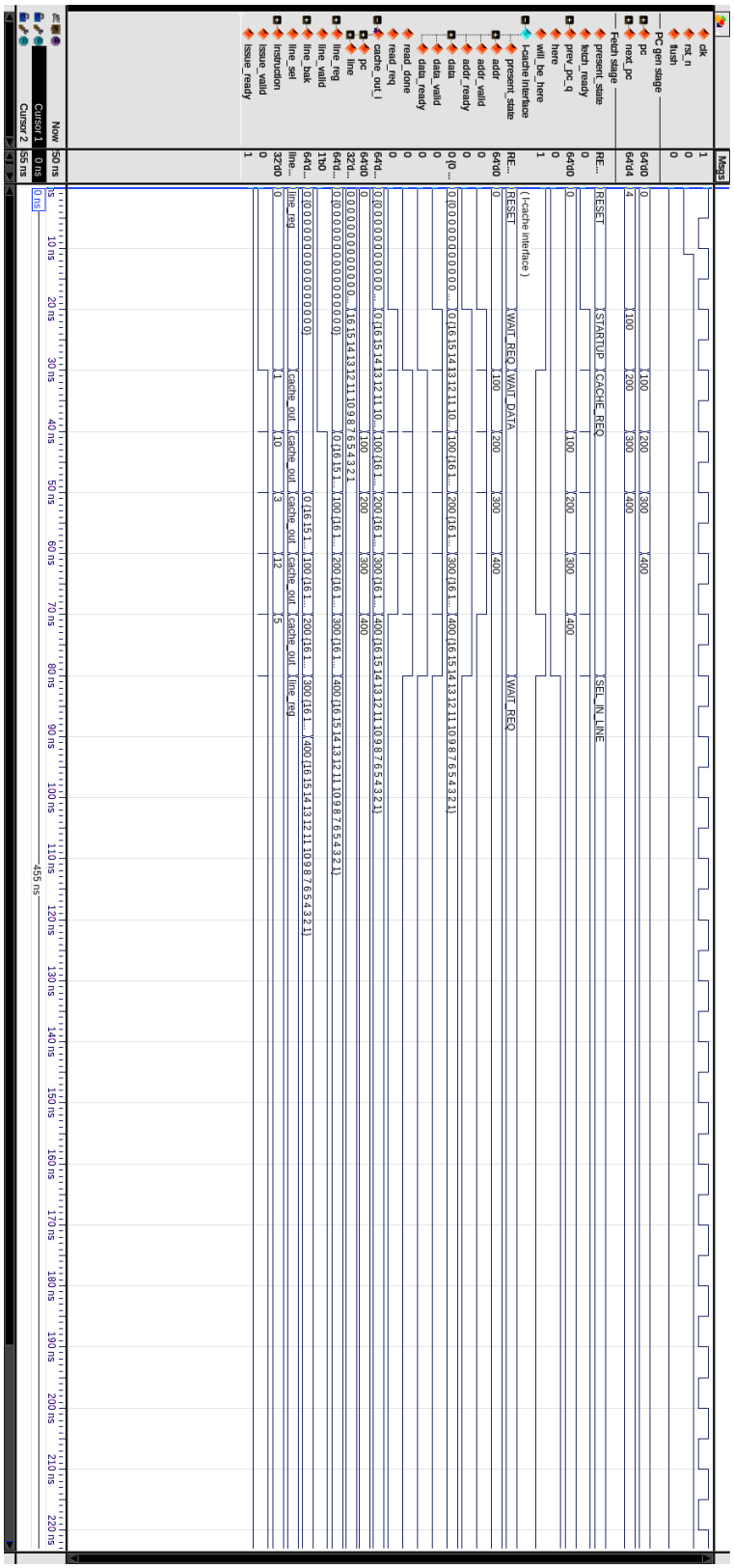


Figure 5.6: Cache read pipeline

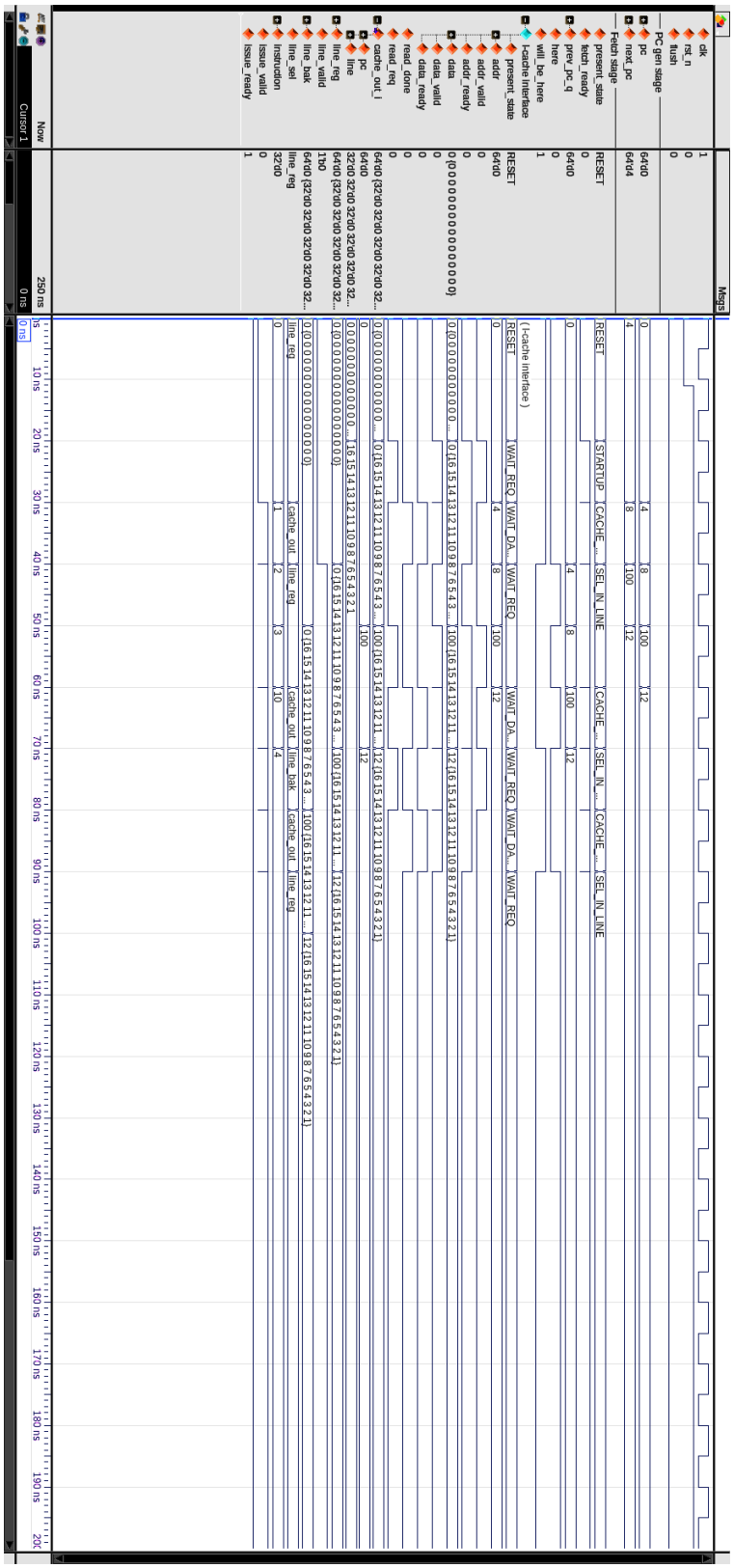


Figure 5.7: Line backup register selection

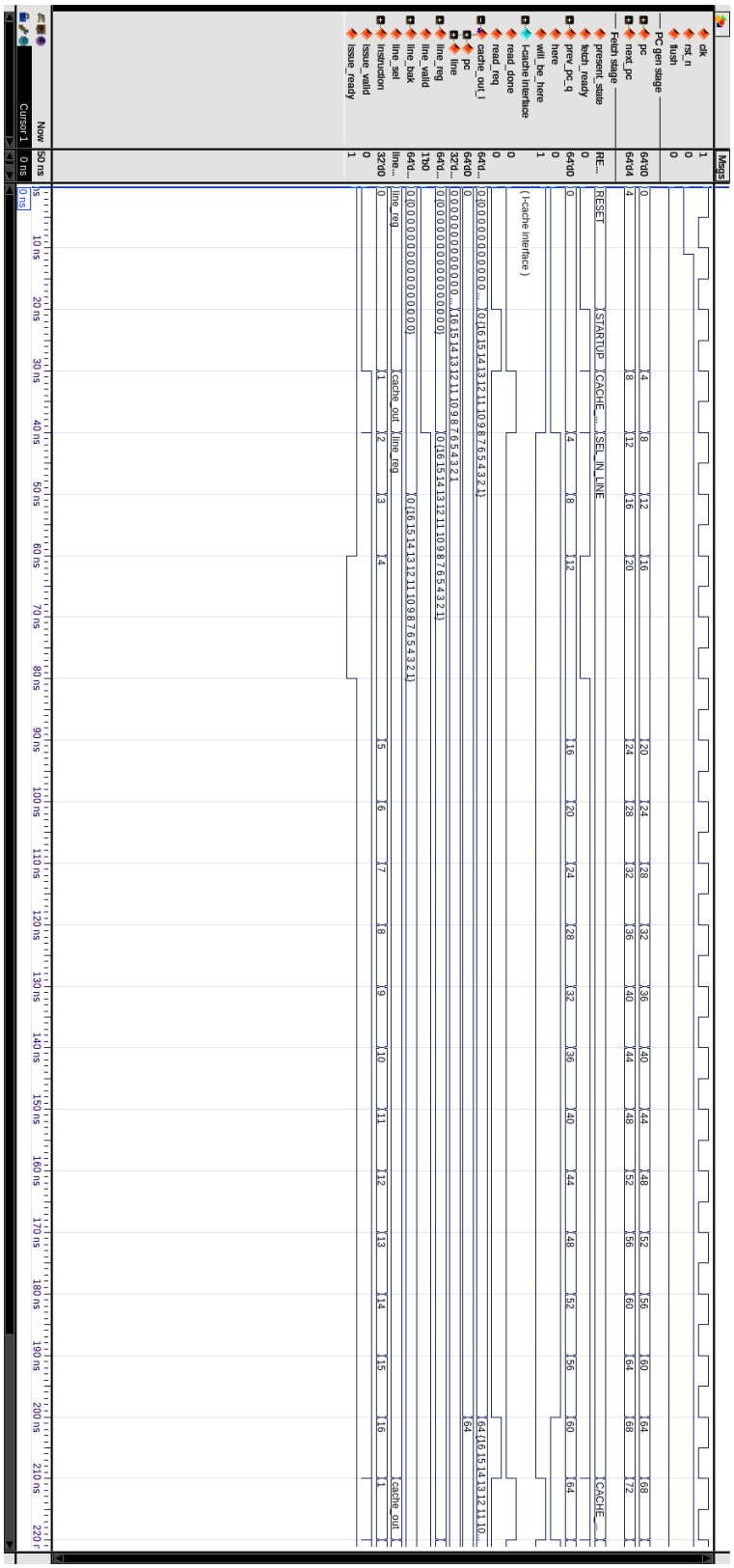


Figure 5.8: Issue queue busy

5.2 BPU! benchmarking

As mentioned before, the **BPU!** is one of the most configurable units in the design, where a number of parameters and design choices come into play. For this reason a software model of this module written in C has been developed, to allow for fast and simple exploration and benchmarking. The model implements the same functions as the hardware and reads an input text file in the form

```
<BRANCH ADDRESS> <OUTCOME>
```

where the address is expressed in hexadecimal base and the outcome as 1 or 0 if the branch is taken or not.

After significant efforts spent to find a way to extract *branch traces* (i.e. the list of branch instructions and their result) from a compiled program, no feasible solution was found and so a decision was made to rely on the trace files provided by a laboratory exercise of the course *Principles in Computer Architecture* held by Prof. Dean Tullsen of the University of California San Diego, available on GitHub[3]. These traces come from a series of benchmarks taken from the SPEC suite, listed in **??**.

Name	Type	Branches
`fp_1`	Floating point	1 546 797
`fp_2`	Floating point	2 422 049
`int_1`	Integer	3 771 697
`int_2`	Integer	3 755 315
`mm_1`	Matrix multiply	3 014 850
`mm_2`	Matrix multiply	2 563 897

Table 5.2: **BPU!** benchmarks

Given that these trace files do not contain the target address associated with each branch instruction, the main limitation of the software model is that it does not account for mispredictions caused by the wrong target being stored in the **BTB!**. It takes into account, however, the case when a **BTB!** too small causes entries to be overwritten frequently, increasing the number of misses and thus mispredictions.

In the following sections, a series of tests is described to evaluate performance and other design metrics on the **BPU!**.

[3] https://github.com/prodromou87/CSE240A

5.2.1 Gshare

For what concerns the gshare predictor, three main parameters can be analyzed:

- The length of the history register, which determines the number of 2-bit counters in the **PHT!**.
- The initial value of those counters, which determines the first predictions for each index.
- The presence or not of the valid bit, along with each counter.

2-bit counters initialization

In order to find the best initial value for the counters, each benchmark was run on two different history register lengths (8 and 16 bits) at each possible initialization, including the alternate version even index/weakly not taken odd index/weakly taken.

Initial value	**Benchmark**					
	`fp_1`	`fp_2`	`int_1`	`int_2`	`mm_1`	`mm_2`
SNT	98.35%	88.62%	69.09%	98.75%	77.50%	82.37%
WNT	98.36%	88.55%	69.09%	98.75%	77.50%	82.37%
WT	98.36%	89.90%	69.09%	98.74%	77.49%	82.37%
ST	98.27%	89.90%	69.09%	98.74%	77.50%	82.37%
even/WNT, odd/WT	98.36%	88.65%	69.09%	98.74%	77.50%	82.37%

Table 5.3: Gshare accuracy on different initializations (8-bit history register)

Initial value	**Benchmark**					
	`fp_1`	`fp_2`	`int_1`	`int_2`	`mm_1`	`mm_2`
SNT	99.15%	99.03%	89.24%	99.63%	96.05%	92.28%
WNT	99.17%	98.85%	88.26%	99.64%	95.86%	92.83%
WT	99.16%	99.04%	89.11%	99.64%	95.88%	92.87%
ST	99.15%	99.04%	89.23%	99.61%	95.97%	92.39%
even/WNT, odd/WT	99.17%	99.04%	88.65%	99.64%	95.88%	92.84%

Table 5.4: Gshare accuracy on different initializations (16-bit history register)

The results are summarized in **????** for 8-bit and 16-bit history registers respectively, where blue cells represent the best accuracy achieved for each benchmark. It is clear that there is no single best initialization value for the 2-bit counters, but it depends heavily on both the benchmark and the history length. Even the most complex initialization to be performed in hardware, which is the alternate one, results the best only in a single case. In any case, there is not much of a difference among the various initial values, so the conclusion is to choose the simplest one, namely the strongly not taken state (i.e. 2-bit counters at zero).

Valid bit

From the results of the previous analysis, it is obvious that the valid bit serves no purpose whatsoever in this design. A valid bit would be used to indicate that the indexed counter has been used before and so the prediction is valid. By initializing the 2-bit counters to zero, however, the first (and second) time they are read, they are always going to predict not taken, exactly like the case with the valid bit. In the end, this bit would only increase by 50% the total **PHT!** size, thus literally wasting a significant amount of area.

History register length

In order to find the best trade-off between the predictor accuracy and the size of the **PHT!**, all the benchmarks were run at increasingly longer history length.

From the plot of **??**, the majority of the benchmarks saturate at their best accuracy value in the range between 13 and 20 bits, with only `int_1` and `mm_1` being the exceptions that continue to get better results the longer the history.

Given the exponential relation between the history register length and the area of the **PHT!**, this range of lengths corresponds to table sizes going from 2 KB to 200 KB. For this reason, the final value must be chosen by keeping in mind a clear area budget for the final implementation. Having said that, a history length of 16 bits, corresponding to around 16 KB of **PHT!**, seems like a reasonable compromise for which benchmark results are no more than 2% worse than the best.

5.2.2 BTB!

The **BTB!** on its own does not contribute to improving the overall prediction accuracy, which only comes from the gshare predictor itself, but instead removes the target computation latency from branch instructions. Thus, its presence can only lower the prediction accuracy with respect to the baseline of the gshare alone, because additional mispredictions are inserted due to **BTB!** misses, which occur when the **BTB!** does not have a valid entry for the selected address. Moreover, mispredictions take place even when the stored target is then discovered as incorrect, but as mentioned above the available branch traces do not contain the correct target,

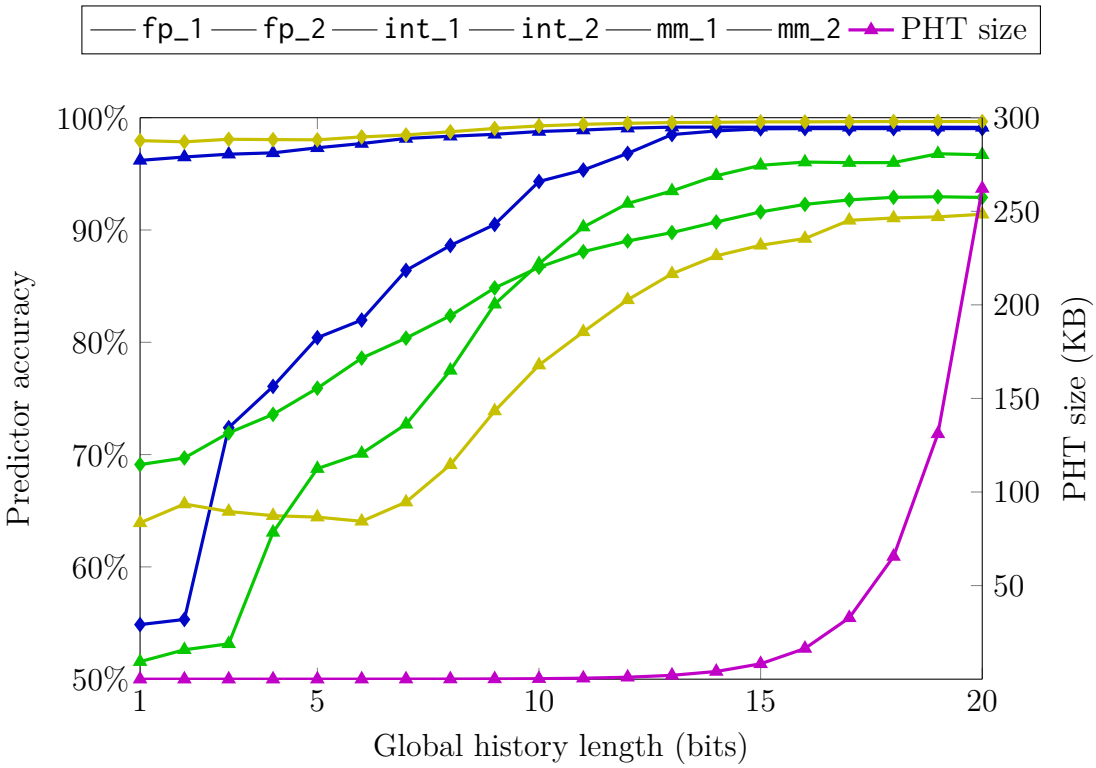


Figure 5.9: History register length versus predictor accuracy

so the software model does not take this kind of mispredictions into account, which contribute only in small part anyway.

?? shows the results of the benchmarks with varying length of the **BTB!** index expressed as the number of mispredictions per a thousand instructions (MPKI), where dashed lines represent the baseline without **BTB!** for each benchmark. Some programs, like `mm_1` and `mm_2` suffer significantly with small **BTB!** and have unacceptably high misprediction rates, while other, notably `int_2` seem almost unaffected by the presence of the **BTB!**. Anyway, around 10 or 12 bits for the **BTB!** index, i.e. in the range 15 KB to 60 KB, most of the benchmark have reached a MPKI very close to the baseline. Going past these values does not seem to offer a significant improvement and thus becomes unadvisable, regardless of the area constraints.

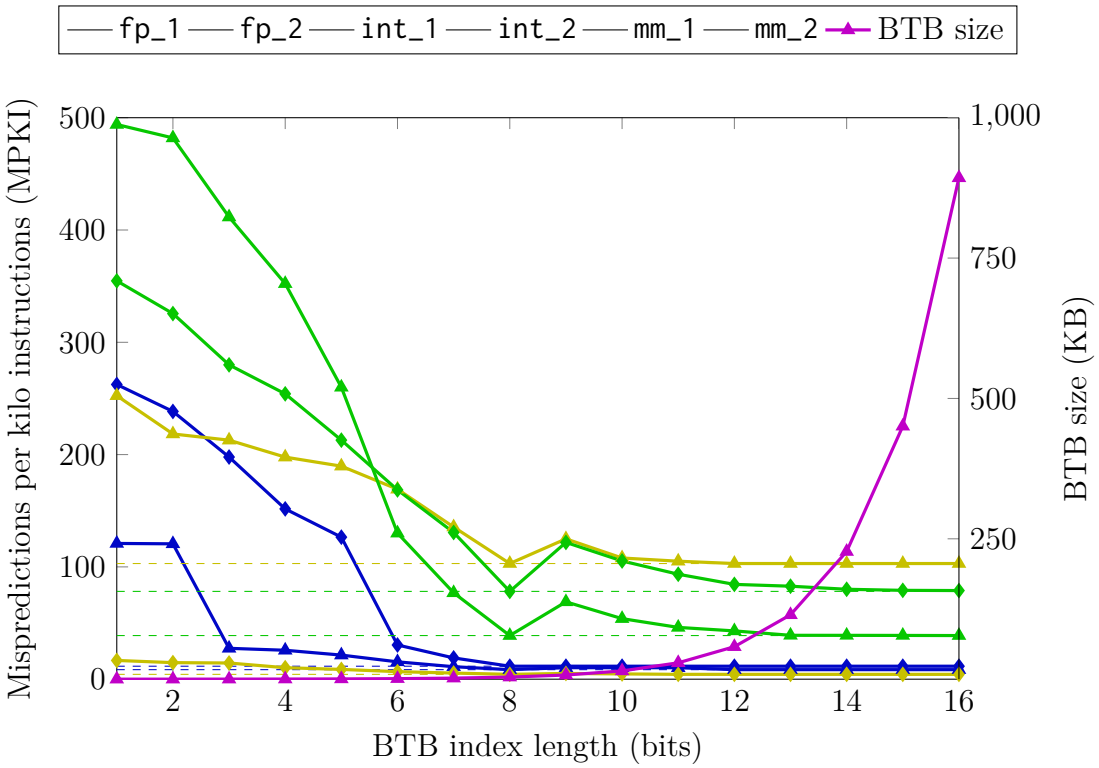


Figure 5.10: **BTB!** index size versus MPKI

5.3 Synthesis

Synthesis has been carried out using Synopsys Design Compiler on the UMC 65 nm low leakage (L65LL) technology library. In order to get reproducible results, scripting was heavily used to input commands to the tool and some notable parameters and settings used in the process are listed below:

- Top down compilation, to allow optimizations beyond module boundaries, even if this means potentially longer compile times and higher memory usage, as stated in [**?**, p.8-6].
- Ideal clock definition (`set_ideal_network` and `set_dont_touch_network`) in order to avoid optimizations on the clock tree, which has to be defined in later place and route stages, and remove warnings about its high fanout.
- Clock uncertainty (skew) 0.07 ns
- Maximum input and output delay 0.5 ns
- Output load of a buffer on all ports (`BUFM10R`, 2.1 fF)
- DC Expert compilation (`compile` command)

5.3.1 BPU!

The **BPU!** was synthesized in different configurations of history register length and **BTB!** size, in order to evaluate the effect of these parameters variations on the design metrics. In particular, from the considerations and the results obtained during benchmarking, the aim was to vary the history length between 2 and 16 bits and the **BTB!** index length from 2 to 12 bits. However, when trying to go past 12 bits of history and 10 for the **BTB!** index, Design Compiler could not sustain the exponential size of the register files and gave up with errors about exceeding the maximum loop iterations possible, nonetheless the results are still significant. Even then, the syntheses, executed with a batch script, took several days to complete.

Area

The total area of the **BPU!** depends almost exclusively on the size of the gshare **PHT!** and the **BTB!**, which are determined by the history length and the **BTB!** index bits. More specifically, given that each entry of the **BTB!** easily contains more than 100 bits compared to the 2 bits of a **PHT!** entry, the effect of the **BTB!** size will weigh 100 times more than the history length on the overall result. In other words, the length of the history register matters only with small **BTB!**s, while it becomes negligible alongside large buffers.

This effect is evident in **??**, which shows the synthesized values of total cell area, demonstrating the expected exponential growth (linear on this logarithmic z axis) with respect to the length of the **BTB!** index. The history length has the effect of slightly increasing the cell area (bending the plane) for low values of the **BTB!** bits, but that effect vanishes for larger buffers and in the end the final area in the case of large **BTB!**s is constant regardless of history length.

Actually, for such large predictor structures, an implementation as register files becomes infeasible and the better way to synthesize them would be to use a memory compiler and a small SRAM to store the **PHT!** and **BTB!**.

Timing

For what concerns timing, the critical path was reported to be the **BTB!** and in particular the path going from the **BTB!** address part of the **PC!** to the `hit` signal, to the `taken` output. Presumably, this corresponds to the decoding network inferred to correctly select the **BTB!** entry. For this reason, **BTB!** size and thus the complexity of such decoder was consistently reported as the only timing critical part of the design, while no variation whatsoever was noted increasing the history bits.

?? shows the results of the maximum achievable clock frequency, obtained by setting the target clock period to zero, with increasing **BTB!** index bits used. Between the two extremes, the critical path grows of about 0.6 ns, which correspond

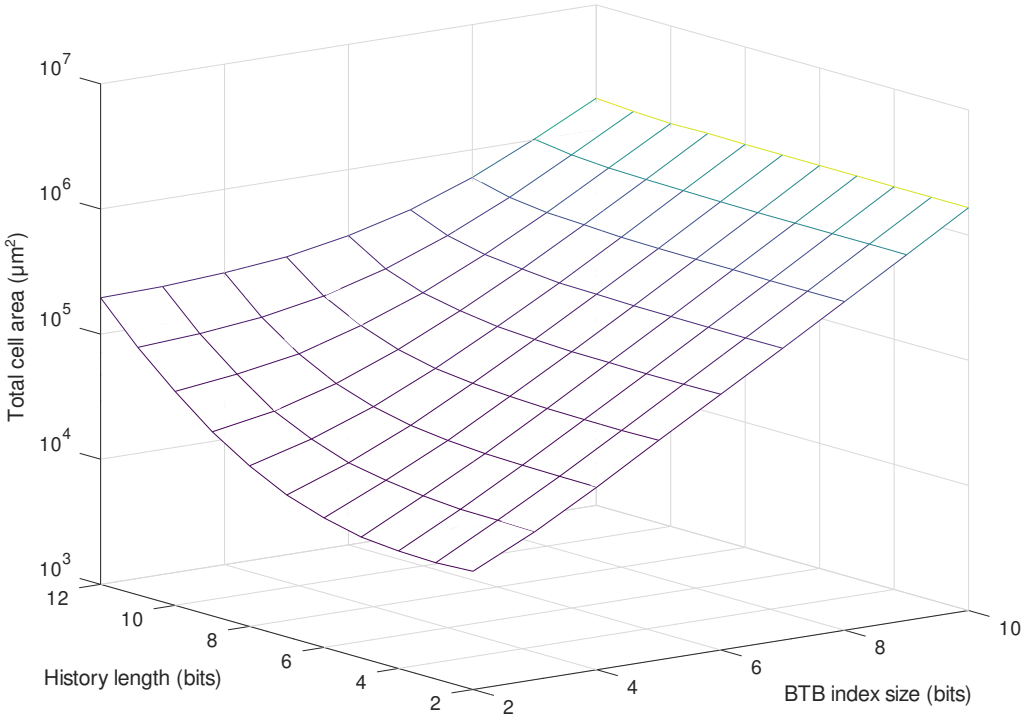


Figure 5.11: Total **BPU!** area versus history length and **BTB!** size

to a non-negligible drop of about 200 MHz in maximum frequency. The seemingly linear negative trend can be explained by reckoning that the decode network to address the **BTB!** is likely to be implemented as a balanced tree of logic gates, whose number of levels thus depends on the base-2 logarithm of the number of entries of the buffer, which in turn grows exponentially with the number of bits, leading to a linear dependence.

In the end, the actual size of the **BTB!** must be chosen by keeping into account the design constraints both in terms of area budget and of target clock frequency, trading them off with the prediction accuracy and hit rate. History bits, on the other hand, do not influence timing at all.

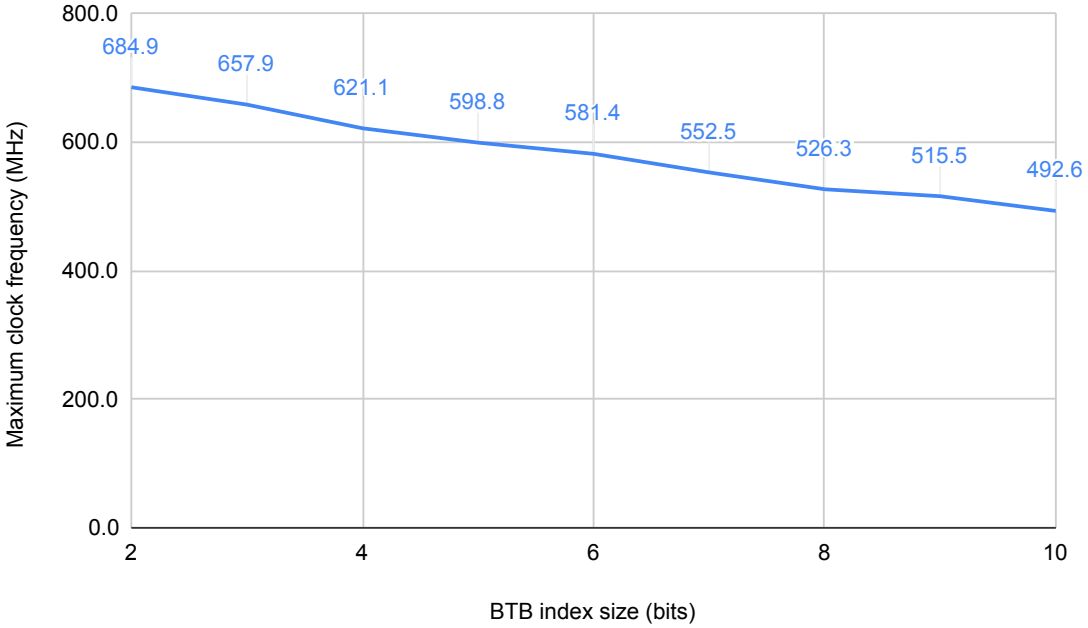


Figure 5.12: Maximum **BPU!** clock frequency versus **BTB!** index bits

5.3.2 Frontend

Given the considerations of the previous section, the entire frontend was then synthesized using 8 bits for both the **BPU!** history length and the **BTB!**, in order to have a configuration of average complexity. **??** summarizes the results.

	BPU only	**Whole frontend**
Area	433 233 µm^2	451 526 µm^2
Maximum frequency	540 MHz	537 MHz

Table 5.5: Comparison of BPU and frontend synthesis results

It is evident that the **BPU!** is by far the limiting element of the frontend, both in terms of area and timing. Given its large data structures, the surrounding logic of the rest of the frontend becomes almost negligible and represents only 4% of the total area for this parameter configuration.

The same can be concluded for timing, as the decoding of the **BTB!** is unfortunately the critical path even in the whole frontend, which leads to almost the same clock frequency in both situations.

To solve this issue, which could really hinder the performance of the entire design, a better implementation, as mentioned before, would employ a SRAM memory with optimized decoding networks instead of a slow large register file. This synthesis must be then only considered as indicative and preliminary, because more advanced tools and compilers could lead to better results.

On a final note, the fact that the instruction selector multiplexer could become quite large with many cache instructions per line, as anticipated in **??**, does not incur any issue. This is because this multiplexer grows linearly with the width of the cache line, so its effect is completely masked by the exponential growth of the **BPU!** structures.

Chapter 6

Concluding remarks

This work provided a basic implementation of an out-of-order processor, based on the RISC-V **ISA!**. Although numeric results are not bad, for instance with the branch predictor reaching easily over 99% accuracy, they are nowhere near the ones provided by modern state-of-the-art processor architectures, which offer much better performance for lower area. These processors, however, are the result of many years of incremental optimizations and advancements in technical know-how of industry leader companies.

The aim of LEN5 was never to compete with the giants, but to be an exploratory experiment able to provide valuable insights on the challenges that such complex designs pose and hopefully serve as the starting point for future RISC-V projects.

6.1 Future work

From an architectural standpoint, there are a number of improvements that the frontend of LEN5 could benefit from, for example:

- A better but more complex branch predictor could be implemented. For example, modern variations of the **TAGE!** predictor, such as [**?**] and [**?**], can achieve almost ten MPKI less than gshare for the same hardware budget.

- Avoid stalling and pausing cache requests when the issue queue is busy, in order to save time on the next read and mask a potential cache miss latency.

- Push to the issue queue more than one instruction in parallel, to increase the issue width.

In any case, the most important future developments concern putting together the final design by merging the three parts developed separately. Then, LEN5 could be used for teaching advanced processor architectures hands on, allowing

to see every detail of the internal organization, which usually are well hidden in commercial products.

Moreover, thanks to the many **ISA!** extensions of RISC-V, LEN5 could also be improved for research applications, such as machine learning accelerators, by implementing dedicated vector and possibly matrix units.

On a final personal note, my fellow designers and I hope that this first open processor experiment carried out at Politecnico di Torino will be able to become a relevant project within our university, with future students improving this basic design and exploring future applications and developments.

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.

[12] ARM, *AMBA AXI and ACE Protocol Specification*, 2017.

[13] Cummings C.E., Mills D., "Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?", *Synopsys Users Group Conference, San Jose, CA, 2002*, User Papers, 2002.

[14] McFarling S., "Combining branch predictors", *Digital Western Research Laboratory*, vol. 49, technical report TN-36, 1993.

[15] Lee J.K.F., Smith A.J., "Branch prediction strategies and branch target buffer design", *Computer*, vol. 1, pp. 6–22, 1984.

[16] Perleberg C.H., Smith A.J., "Branch target buffer design and optimization", *IEEE transactions on computers*, vol. 42(4), pp. 396–412, 1993.

[17] Synopsys, *Design Compiler User Guide*, 2011.

[18] Parasanna S., Sarma R., Balasubramanian S., "A study on improving branch prediction accuracy in the context of conditional branches", *Int J Eng Technol Sci Res*, vol. 4, pp. 654–662, 2017.

[19] Michaud P., "An Alternative TAGE-like Conditional Branch Predictor", *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15.3, p. 30, 2018.

[20] Cummings C.E., "Synthesizable finite state machine design techniques using the new SystemVerilog 3.0 enhancements", *Synopsys Users Group Conference, San Jose, CA, 2002*, User papers, 2003.