POLITECNICO
DI TORINO

Master's Degree in Electronic Engineering

Master Thesis

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

Supervisor
Prof. Maurizio Martina

Candidate
Marco Andorno

Academic year 2018-2019

Abstract

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

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

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

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

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

Acknowledgements

Thanks everybody!

Contents

List of Tables

List of Figures

List of Acronyms

Chapter 1

LEN5 frontend

1.1 General block diagram

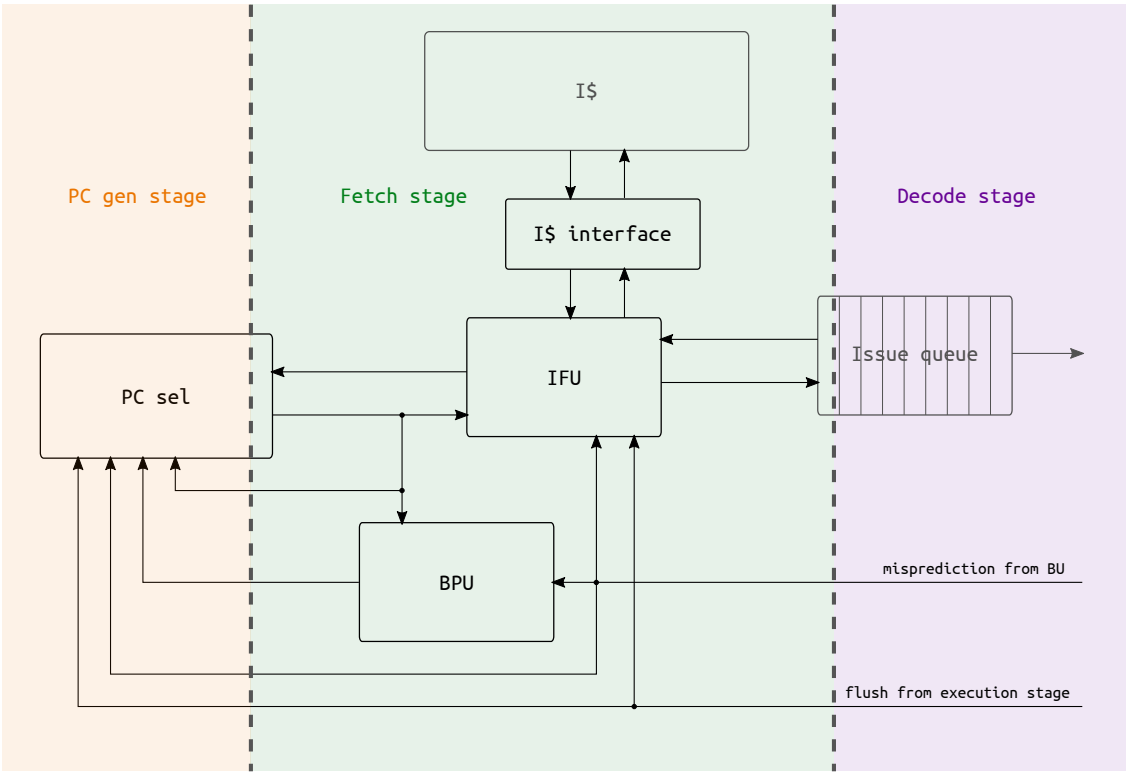


Figure 1.1: LEN5 frontend

?? 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.

The frontend is composed of two pipeline stages, name the ***PC!*** *(**PC!**) generation* and the *fetch* stages. **??** also shows the *decode* stage, where the issue queue is found. This 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 issue to the later stages of the pipeline.

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.

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 **??**), and 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.

1.1.1 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.

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

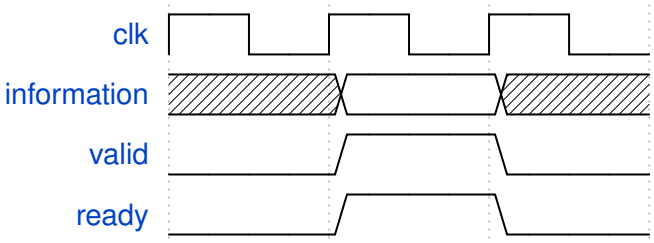
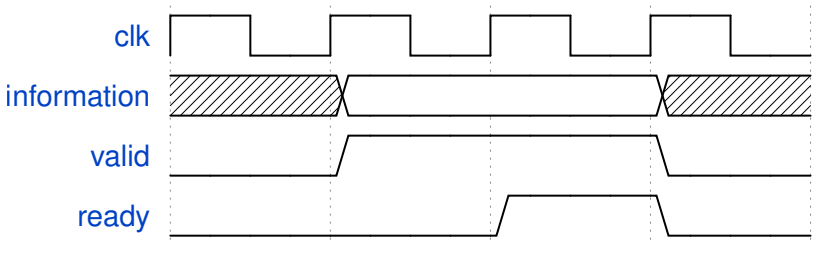
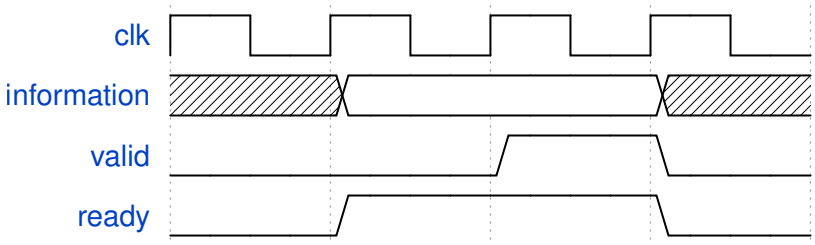


Figure 1.2: AXI handshake protocol



(a) Valid before ready



(b) Ready before valid

Figure 1.3: Possible handshake timings

valid before asserting ready and it can also deassert ready before a corresponding valid arrives.

1.2 PC! gen stage

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

1. **Exception**: 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.

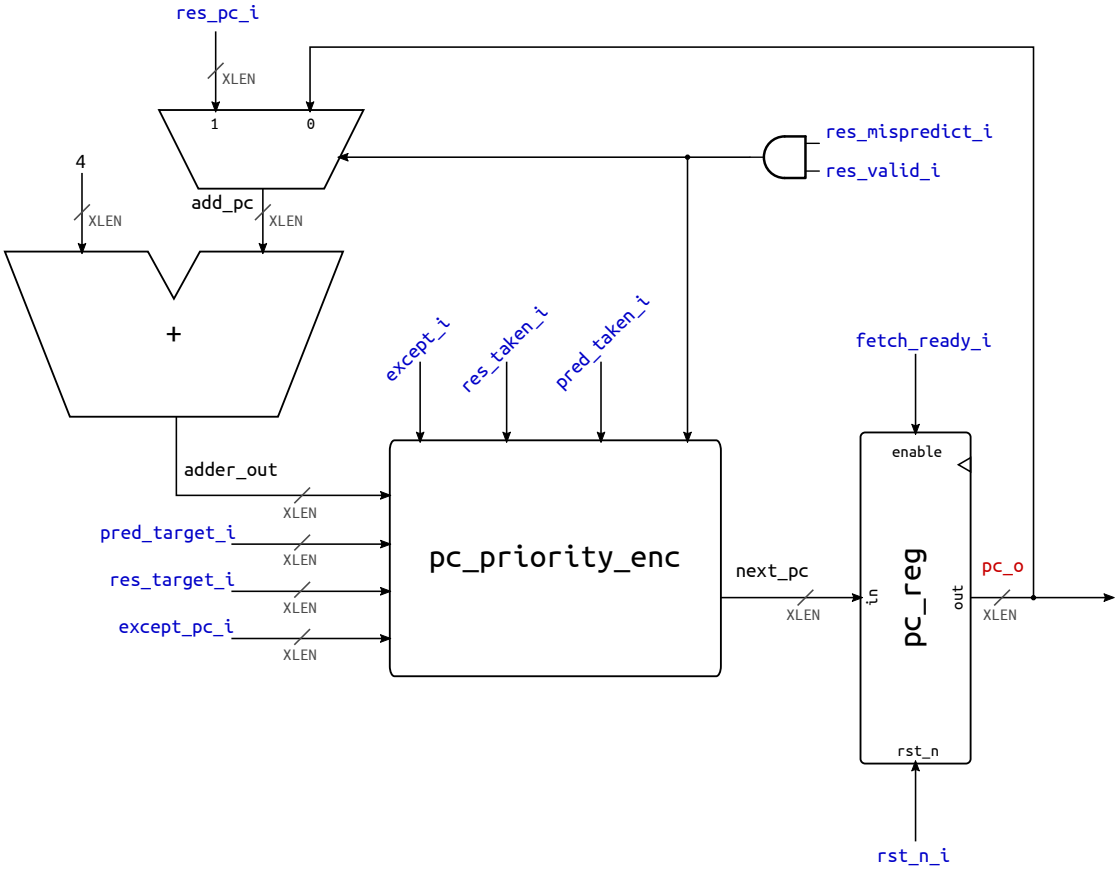


Figure 1.4: **PC!** gen stage diagram

?? 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, output signals are in red, 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 a if-then-else chain, which gets synthesized as a list of cascading multiplexers implementing the desired priority[1].

```
always_comb begin: pc_priority_enc
  if (except_i) begin
    next_pc = except_pc_i;
  end else if (res_valid_i && res_mispredict_i) begin
    if (res_taken_i) begin
      next_pc = res_target_i;
    end else begin
      next_pc = adder_out;
    end
  end else if (pred_taken_i) begin
    next_pc = pred_target_i;
  end else begin
    next_pc = adder_out;
  end
end: pc_priority_enc
```

Listing 1.1: `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 selected 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_i` 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.

1.3 Instruction cache interface

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 block needs to be updated, while the signals coming from the **IFU!** module would remain unchanged.

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

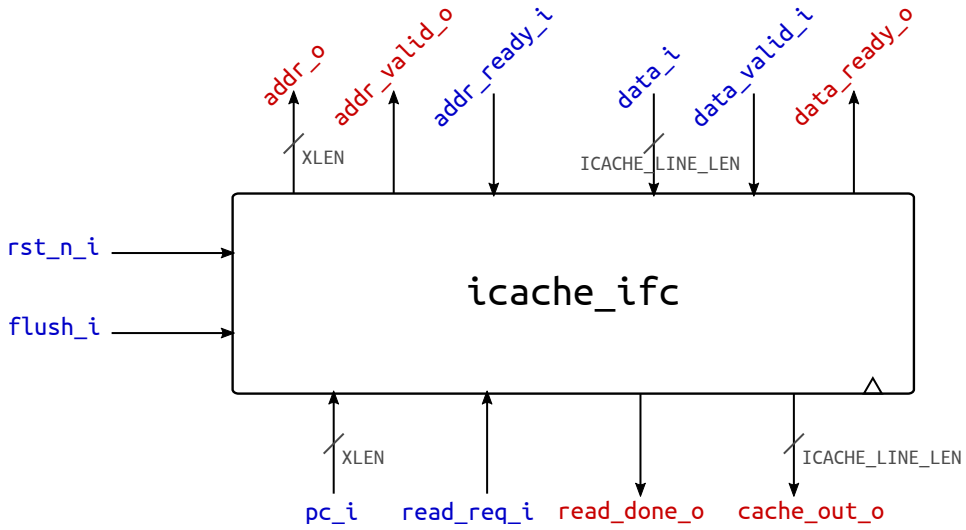


Figure 1.5: Instruction cache interface module ports

1.3.1 Datapath

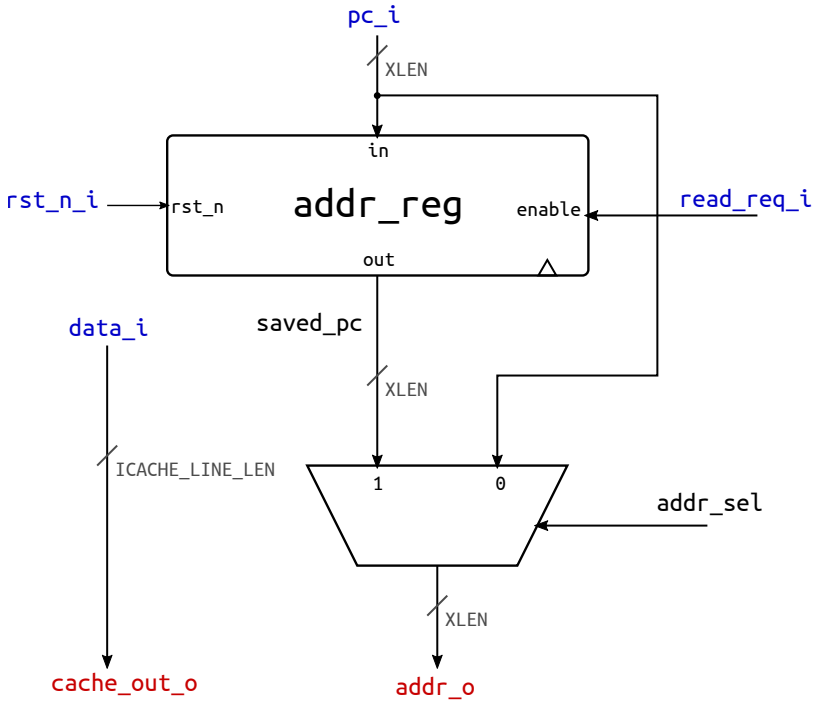


Figure 1.6: Instruction cache interface datapath

For what concerns data signals, the line received from the is directly connected

with the cache_out output, as the instruction cache 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.

1.3.2 Control FSM! and timing

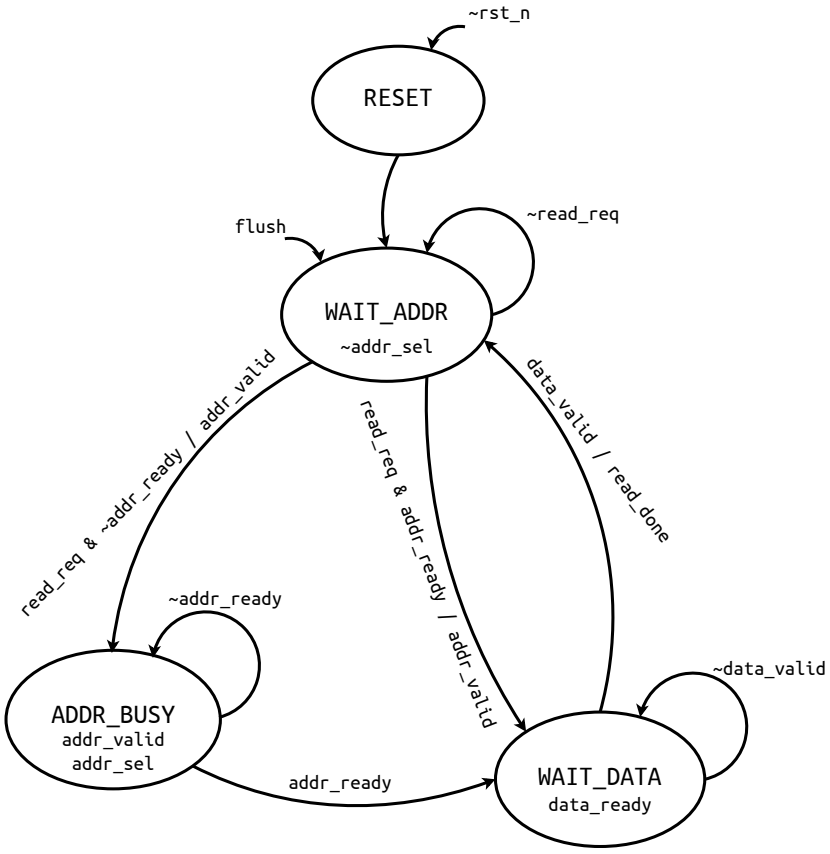


Figure 1.7: Instruction cache interface **FSM!**

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.

?? 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

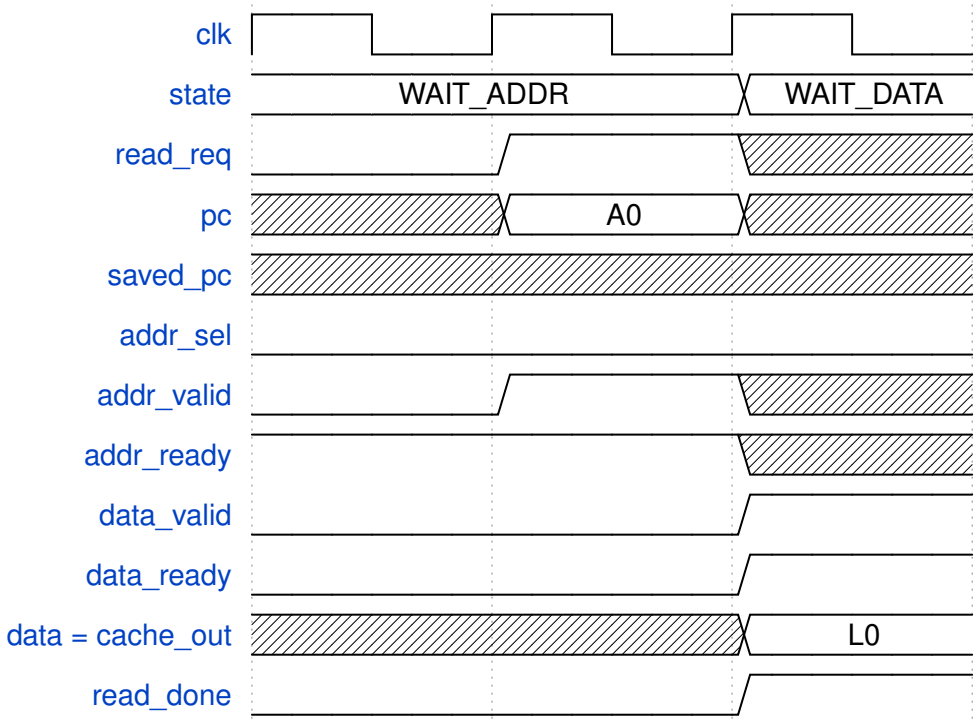


Figure 1.8: Normal cache read

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_ADDR` state, but the additional state was introduced for robustness with `addr_valid` as a Moore output, so that this way there is no need 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.

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.

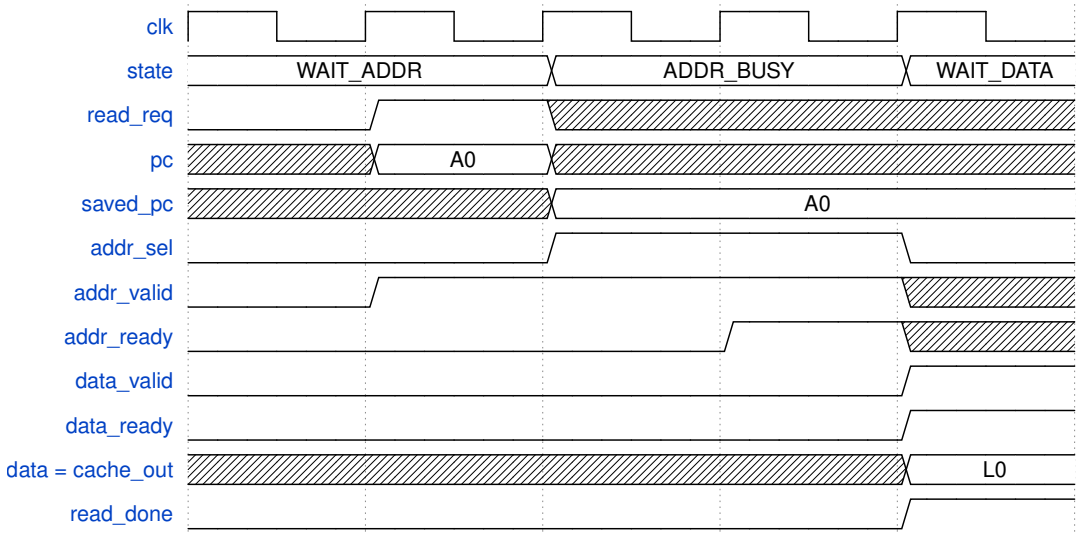


Figure 1.9: Cache not ready on address

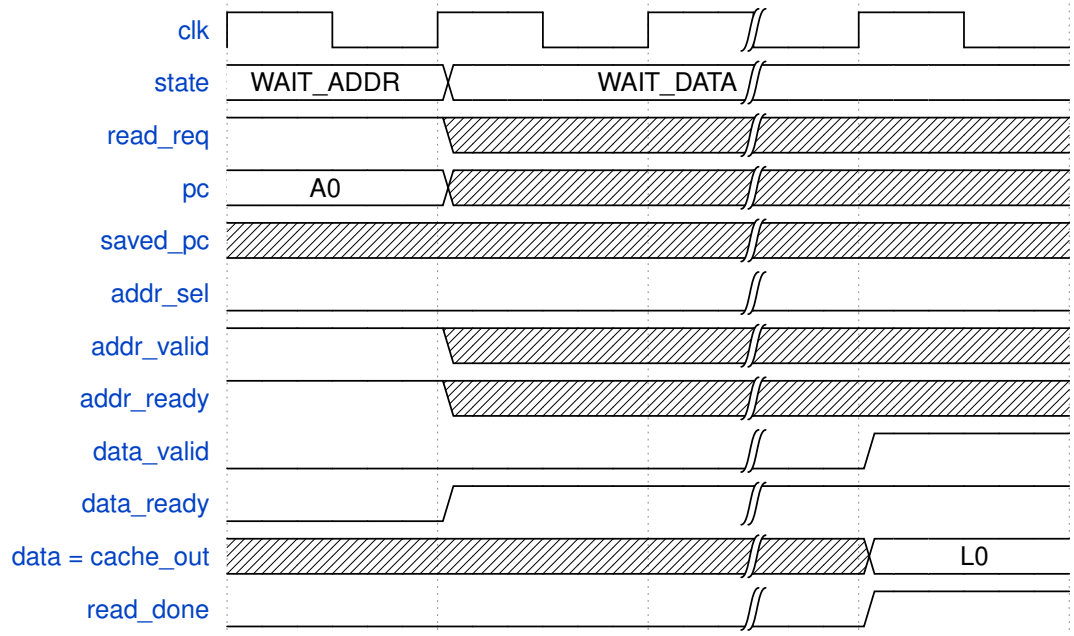


Figure 1.10: Cache miss

1.4 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*

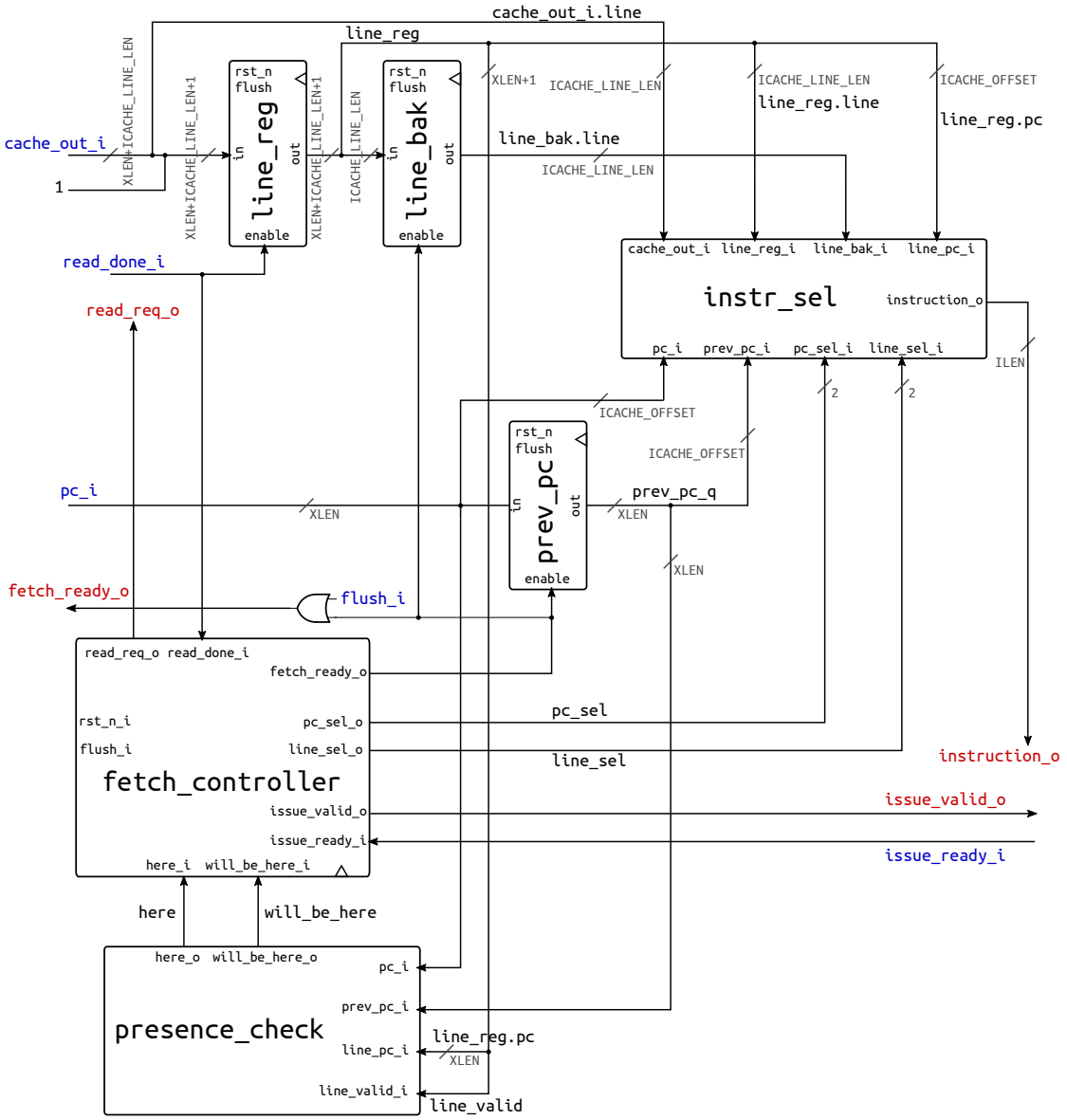


Figure 1.11: **IFU!** diagram (inputs `rst_n_i` and `flush_i` are shown as module port but are not connected with wires to avoid further clutter in the diagram, apart from the output OR gate)

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 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 not. 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, but leaving it as an external signal similar to the initial reset simplifies significantly the state diagram and helps readability.

1.4.1 Presence checker

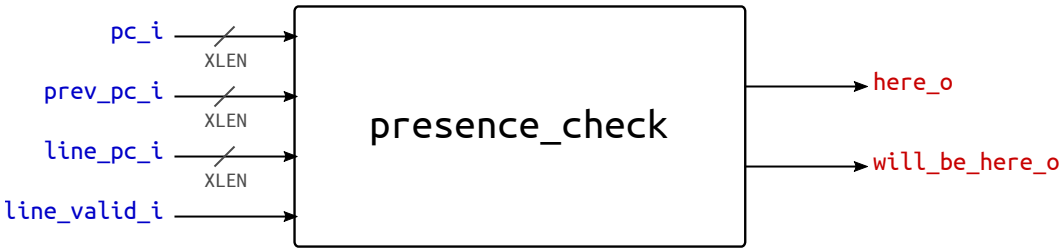


Figure 1.12: Presence checker module ports

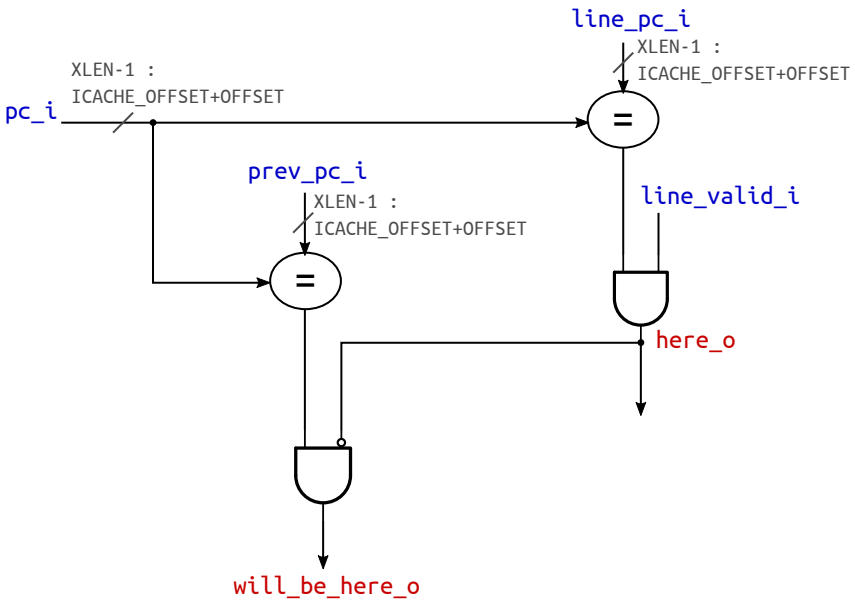


Figure 1.13: Presence checker combinational network

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 addresses, namely only the N **LSB!**s 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 **??**.

1.4.2 Instruction selector

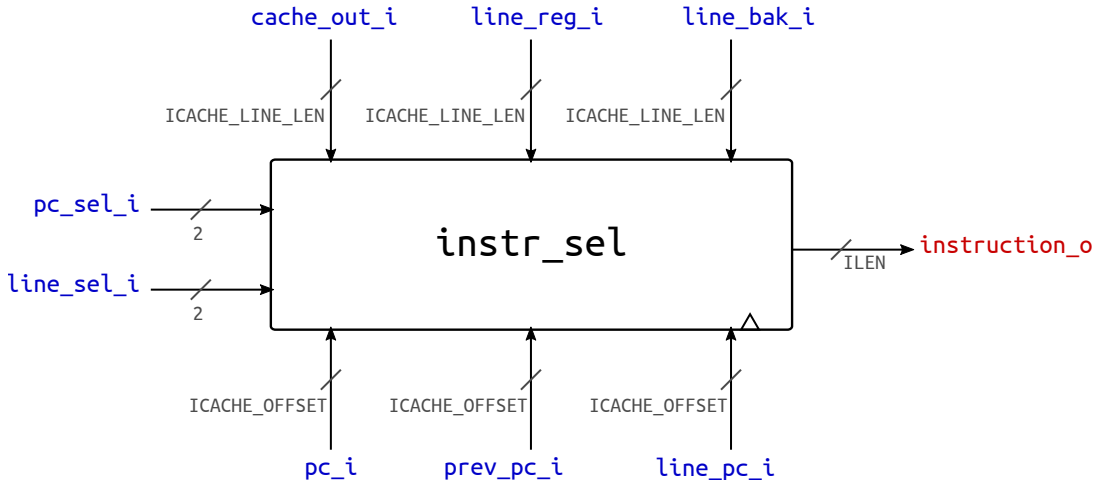


Figure 1.14: Instruction selector module ports

The instruction selector takes as input all the three sources an instruction can be fetched from, three program counter sources[2] and the respective selection signals (see **??**). It outputs a single selected instruction to be pushed to the issue queue.

[2]The only source actually used, as stated above, is the previous **PC!**, but the others were

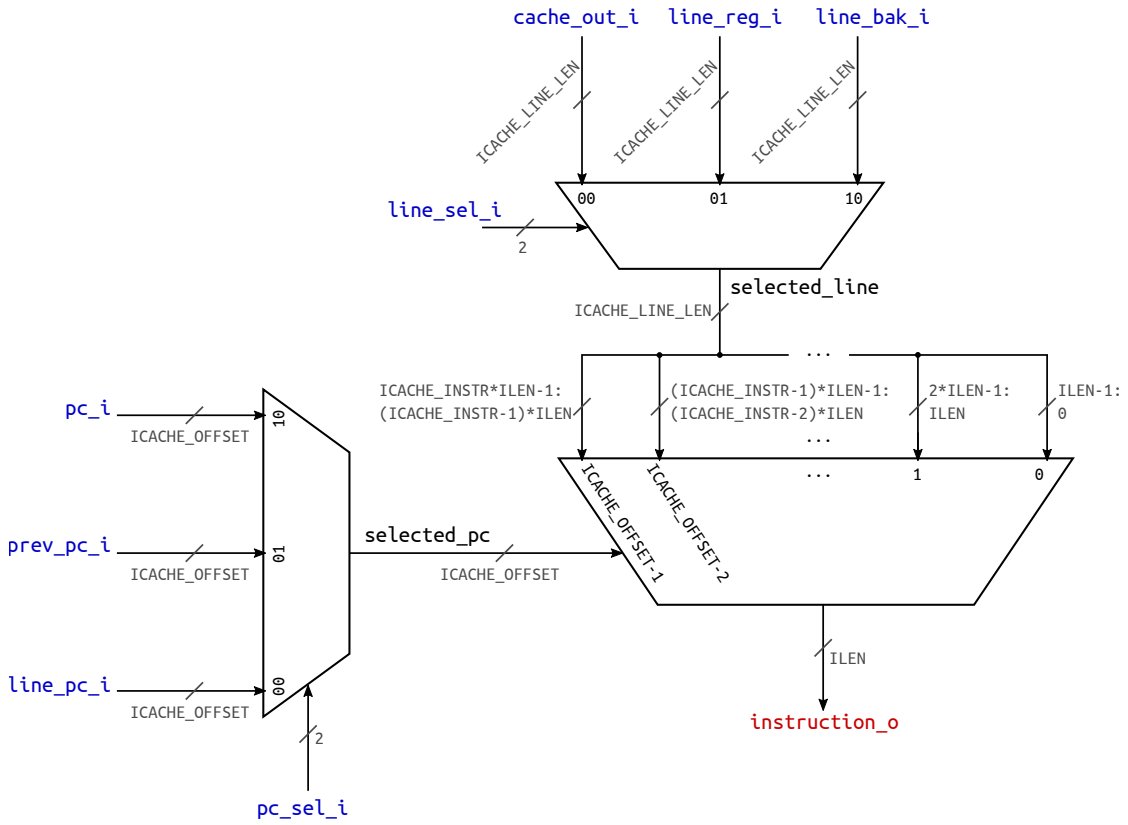


Figure 1.15: Selection network

?? 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.

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.

1.4.3 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

included in the initial version of the design and are kept should a future need arise. Given that the synthesizer can optimize unused wires, this choice incurs in no overhead.

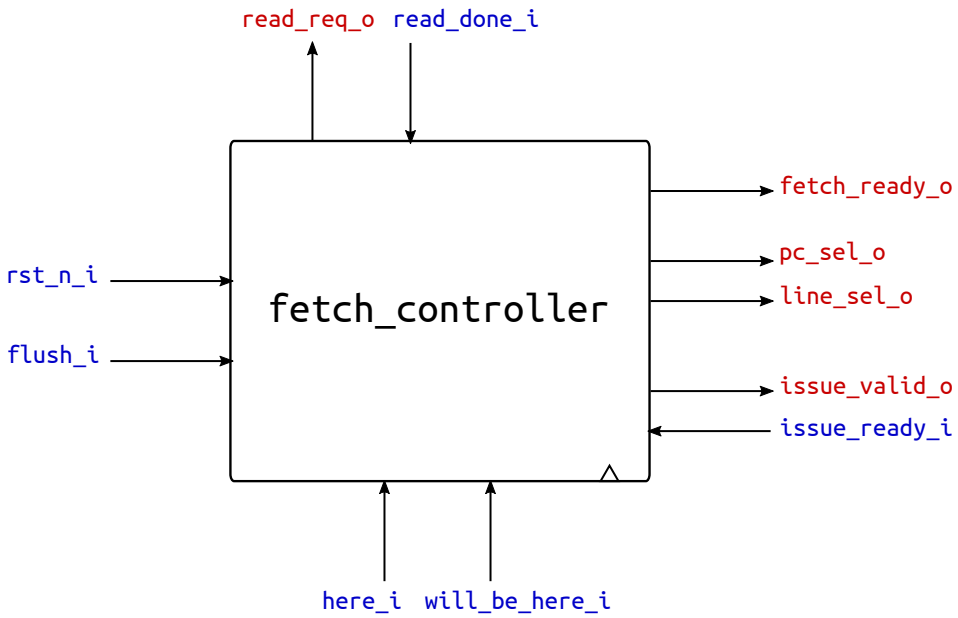


Figure 1.16: Fetch controller module ports

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 ready signals that the issue queue is not full and has available room for one 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. Follows a summary of the purpose of each state:

Why not Moore? Explain.

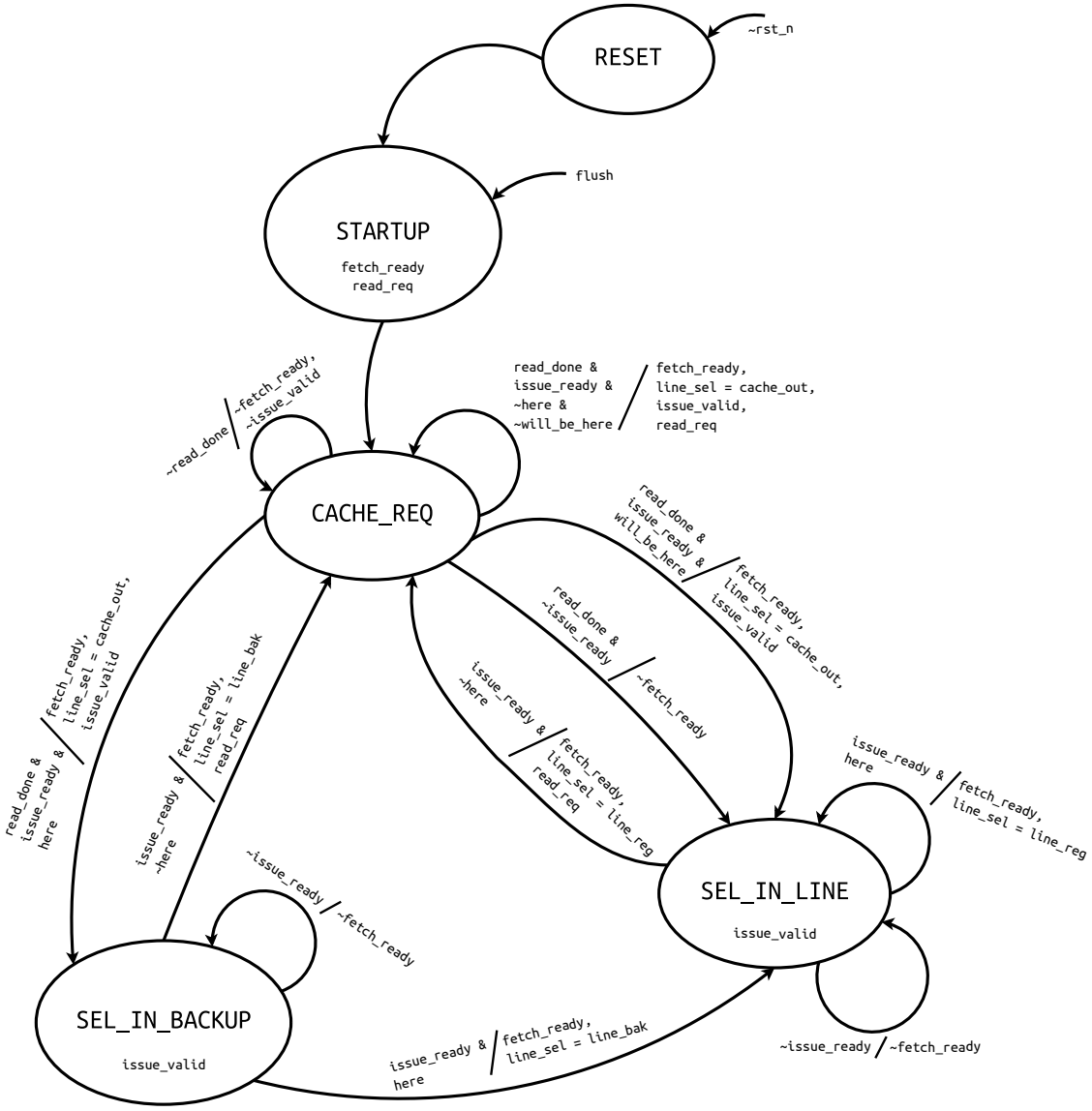


Figure 1.17: Fetch controller **FSM!**

- 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 in 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 will loop 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.

Solution to mask a potential cache miss here?

- `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.

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.

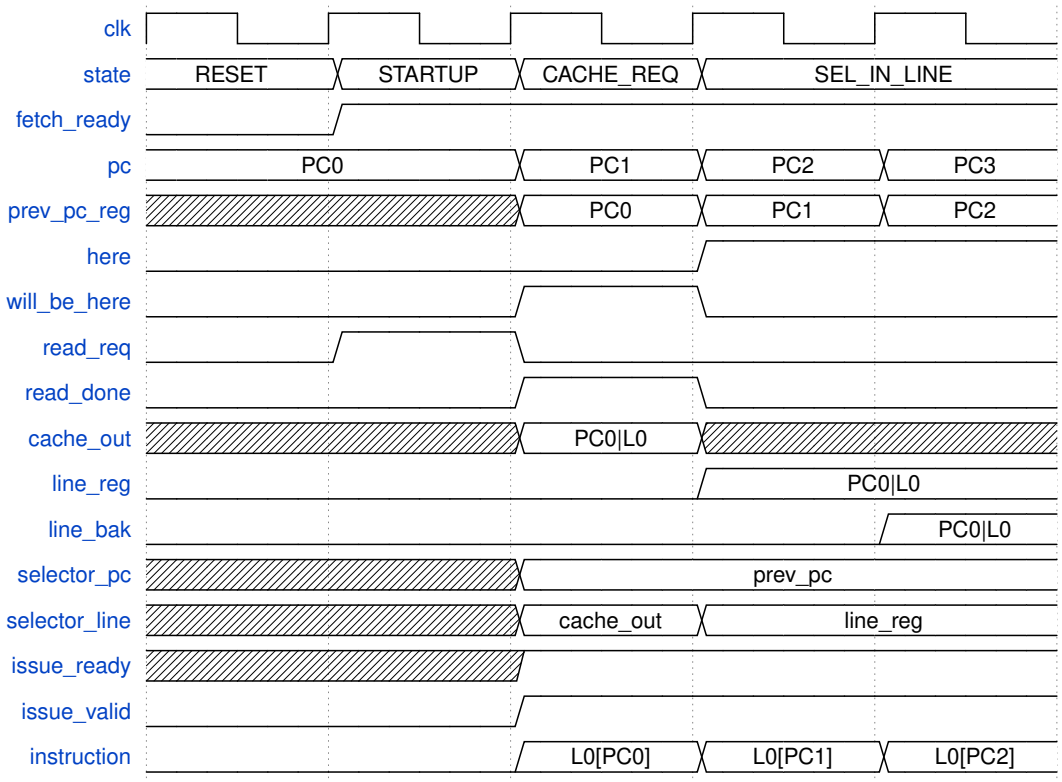


Figure 1.18: Startup and hit (`PC0` is the `BOOT_PC` considered above)

?? 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.

?? 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.

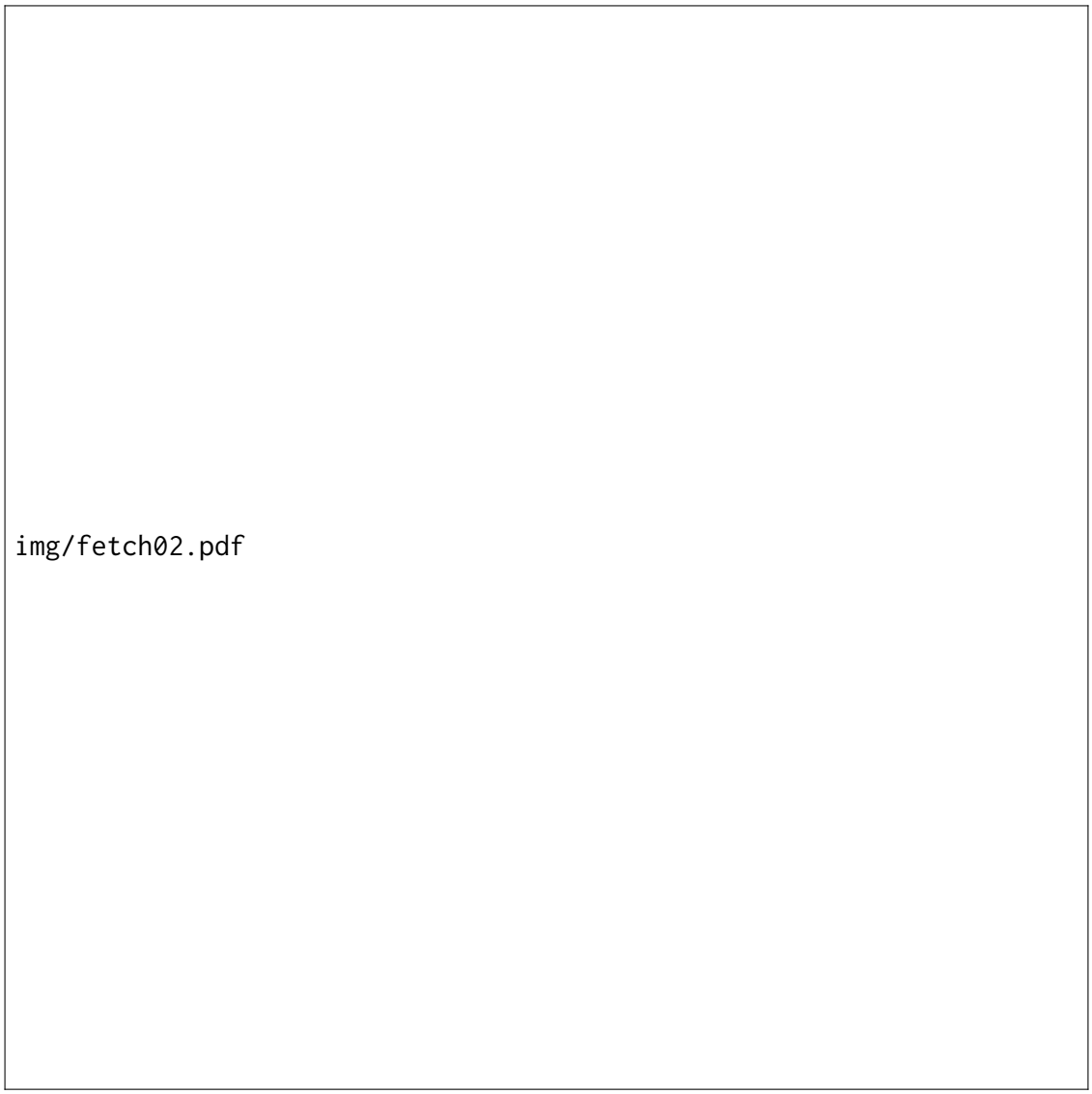


Figure 1.19: Startup and cache not ready/miss

1.5 BPU! (BPU!)

1.6 Branch unit

Chapter 2

Results

2.1 Simulation

2.1.1 Fetch unit

2.1.2 BPU

2.2 BPU benchmarking

2.3 Synthesis results

Chapter 3

Concluding remarks

3.1 Future work

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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