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

Candidato
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

BHT Branch History Table

BPU Branch Prediction Unit

BTB Branch Target Buffer

FIFO First-In-First-Out

FSM Finite State Machine

IFU Instruction Fetch Unit

ISA Instruction Set Architecture

PC Program Counter

PHT Pattern History Table

Chapter 1

LEN5 frontend

1.1 General block diagram

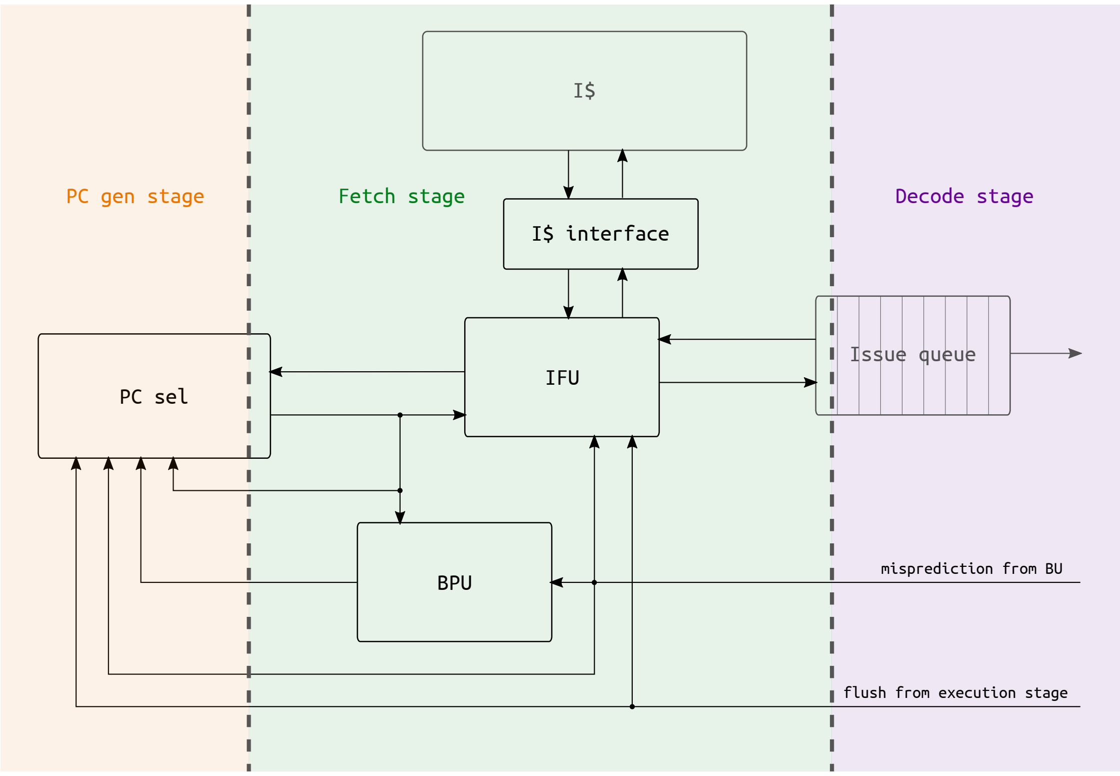


Figure 1.1: LEN5 frontend

Figure 1.1 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 *Program Counter (PC) generation* and the *fetch* stages. Figure 1.1 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.

1.1.1 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 figure 1.1 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.

1.1.2 Fetch stage

In the fetch stage, the PC is used by the Instruction Fetch Unit (IFU) to select and possibly read from memory the next instruction to be pushed to the issue queue. At the same time, the Branch Prediction Unit (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 section 1.4); 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.

1.1.3 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 [12, 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 figure 1.2, the handshake happens at the third rising edge of the clock.

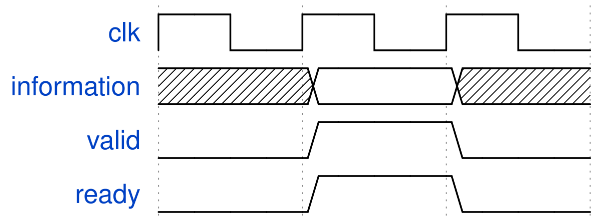
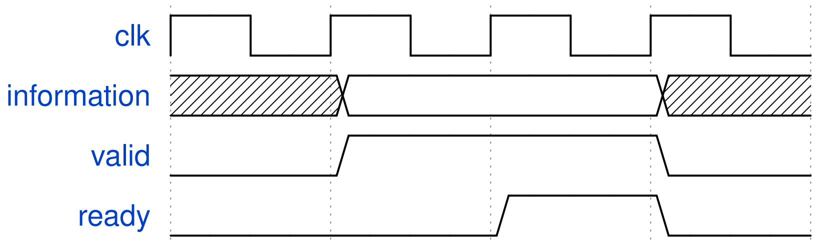
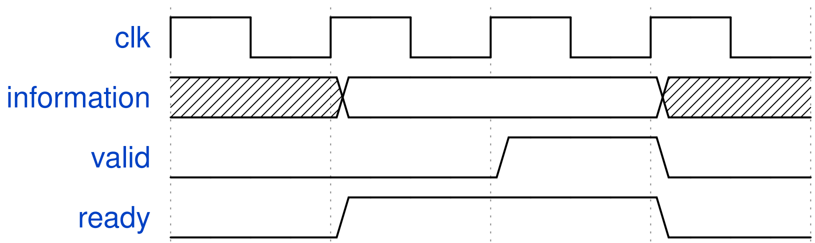


Figure 1.2: AXI handshake protocol

When a source has information available (figure 1.3a), 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 (figure 1.3b), a destination is allowed



(a) Valid before ready



(b) Ready before valid

Figure 1.3: Possible handshake timings

to wait for its 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.

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**[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 section 1.5.3), 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 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 PCs. In behavioral SystemVerilog (listing 1.1) 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.

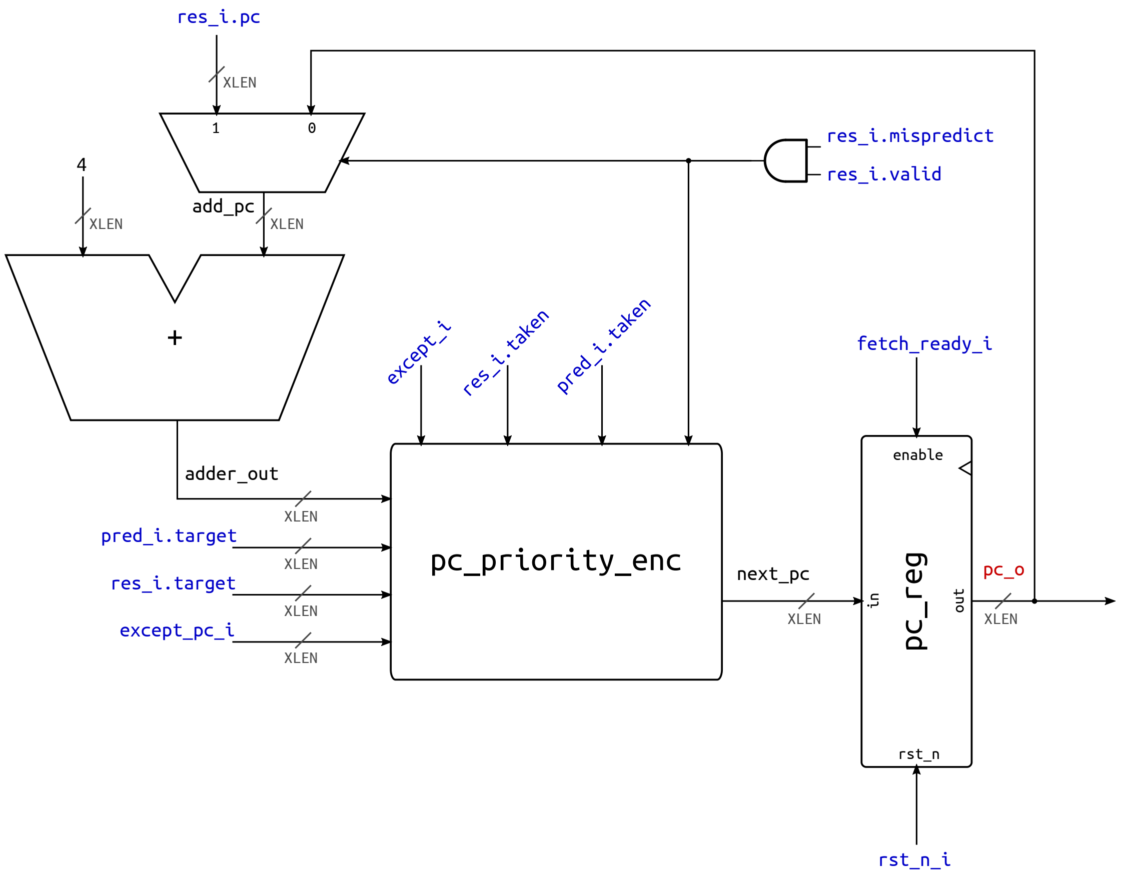


Figure 1.4: PC gen stage diagram

```
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 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 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 section 1.1.3, 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

Talk about FSM coding style

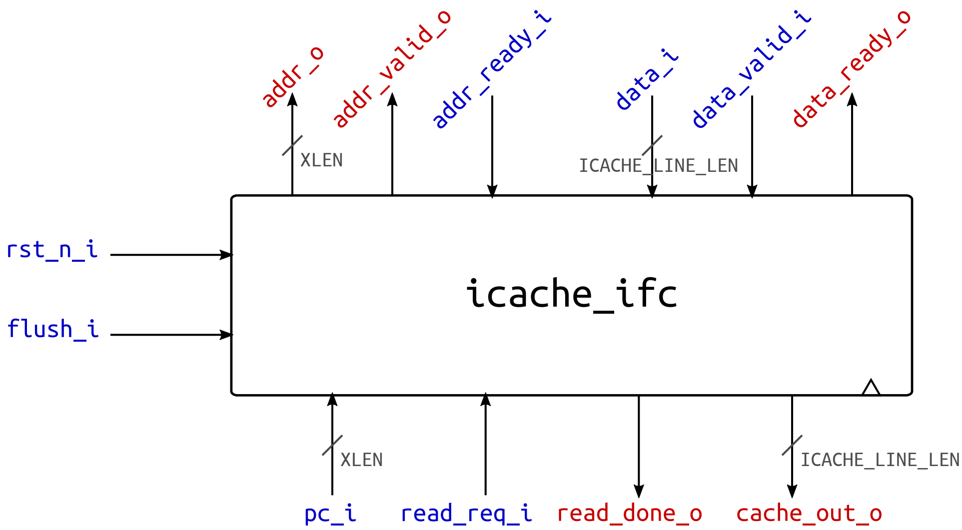


Figure 1.5: Instruction cache interface module ports

The instruction cache interface (figure 1.5) 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.

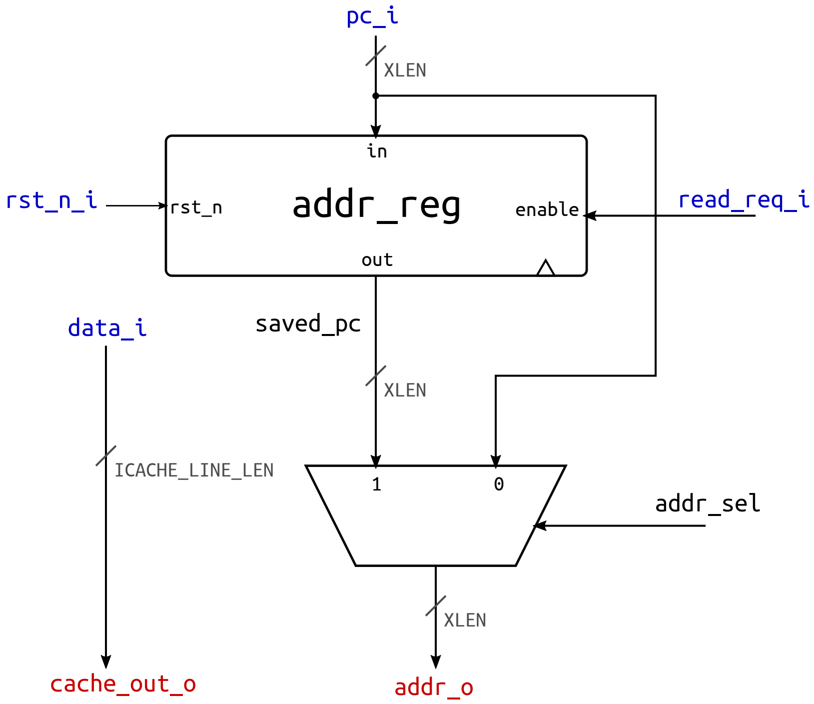


Figure 1.6: Instruction cache interface datapath

1.3.1 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 figure 1.6. 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

The control unit of this block is a simple Mealy FSM (figure 1.7) 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.

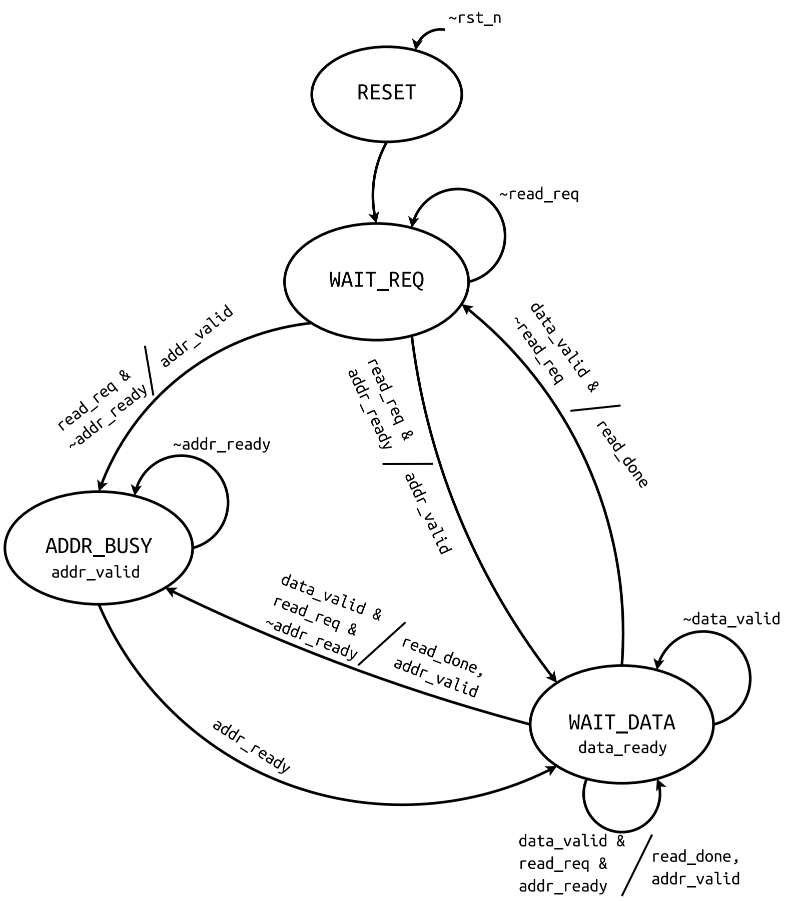


Figure 1.7: Instruction cache interface FSM

Figure 1.8 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.

Figure 1.9 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

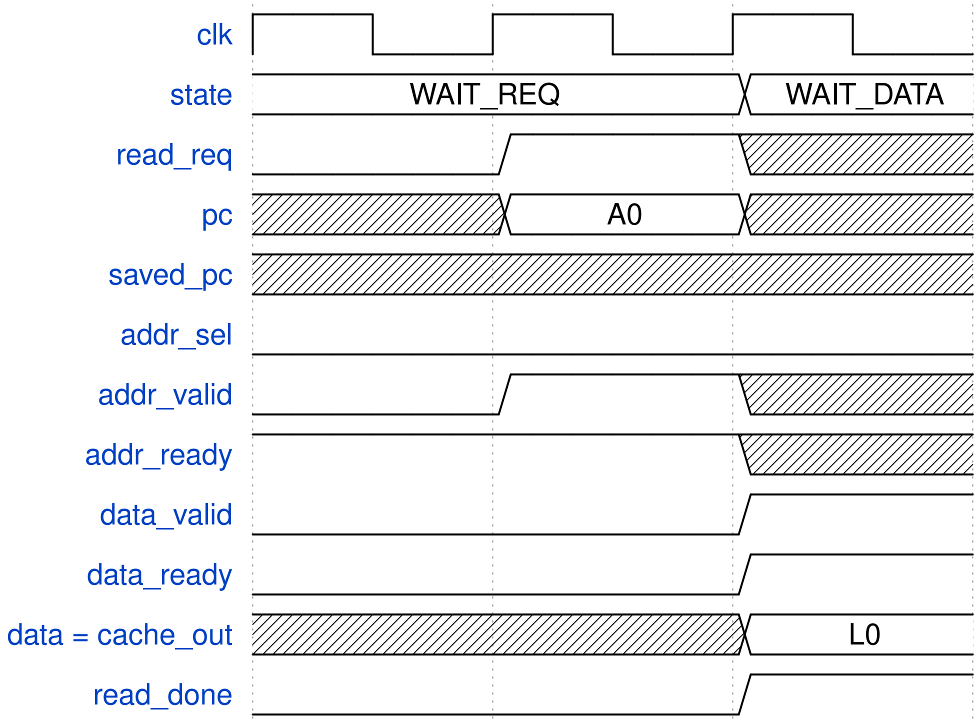


Figure 1.8: Normal cache read

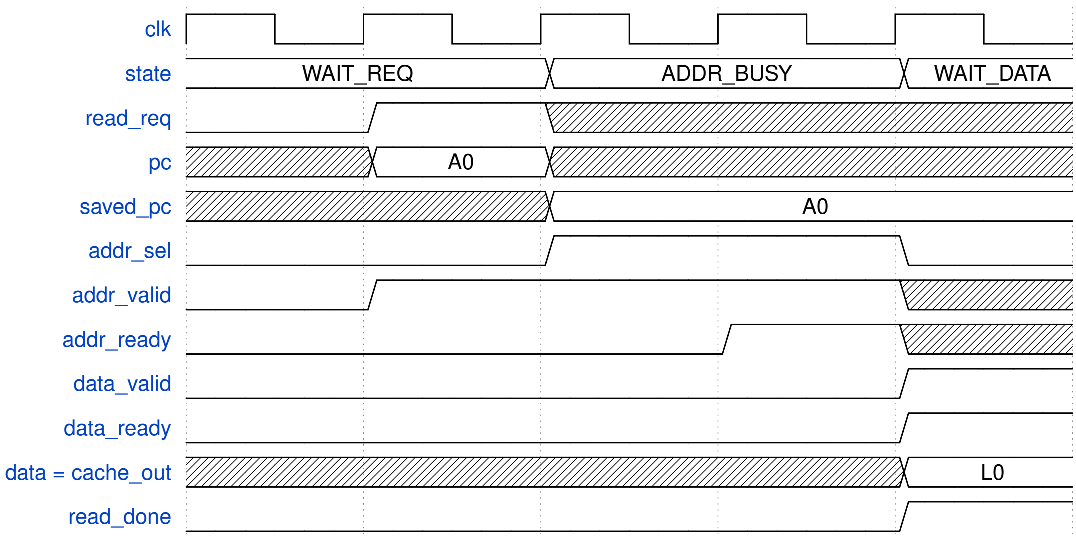


Figure 1.9: Cache not ready on address

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.

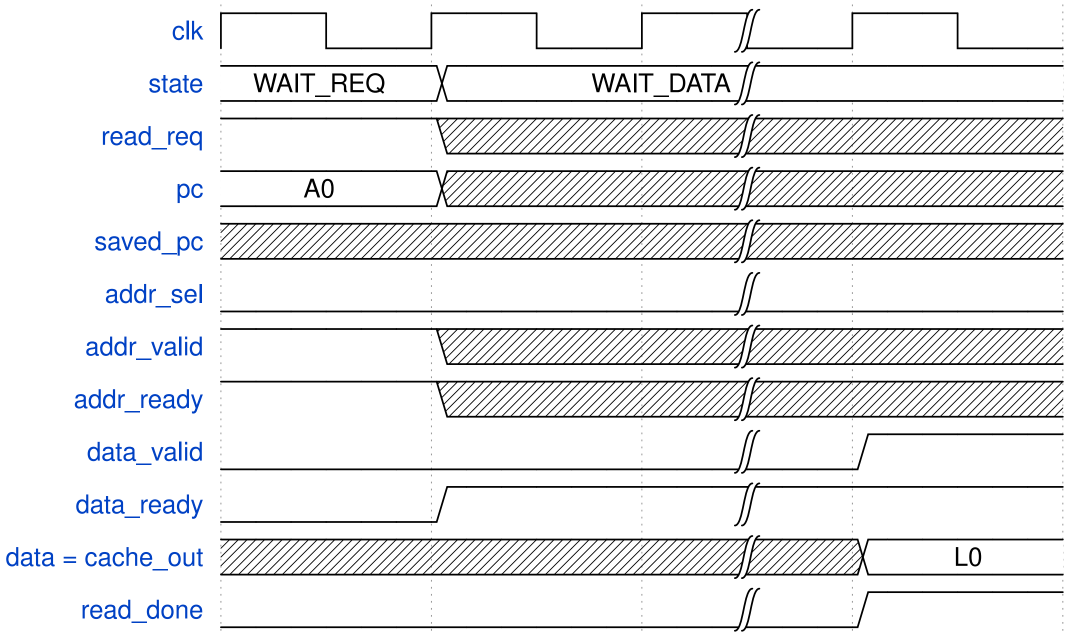


Figure 1.10: Cache miss

Finally, figure 1.10 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.

1.4 Instruction Fetch Unit (IFU)

Figure 1.11 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 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

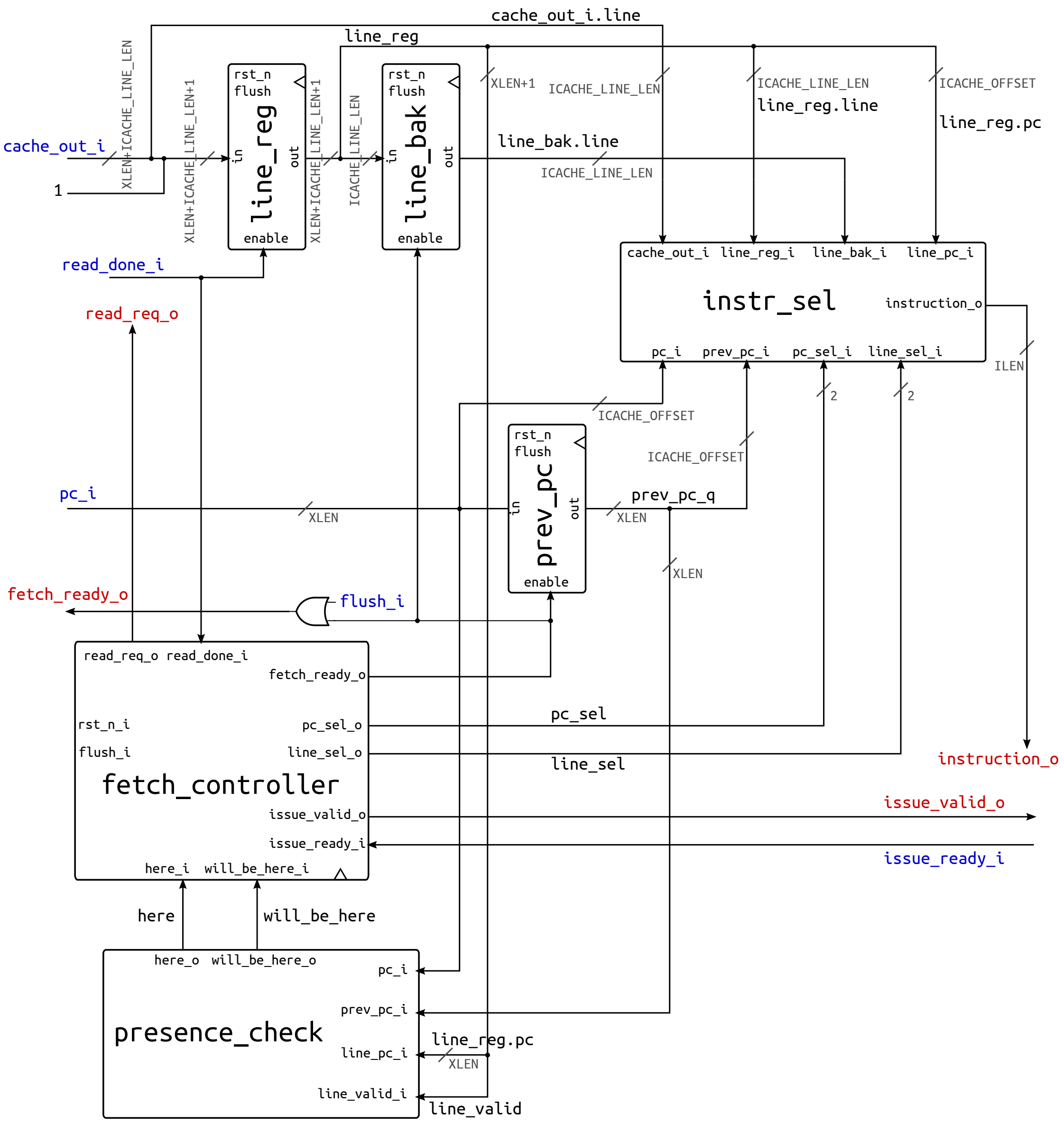


Figure 1.11: IFU diagram (inputs `rst_n_i` and `flush_i` are shown as module port but are not connected with wires to avoid further clutter in the diagram, apart from the output OR gate)

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 FSMs to their startup state [13], 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.

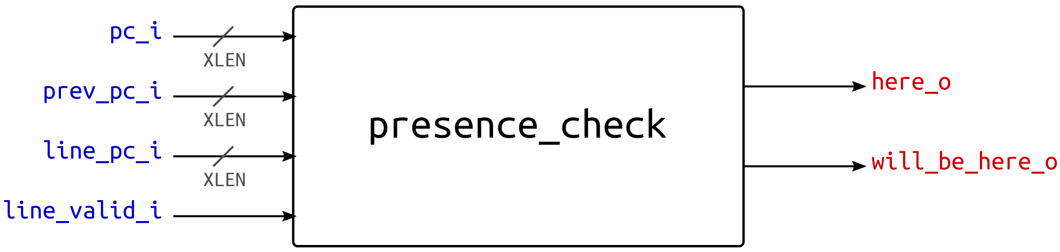


Figure 1.12: Presence checker module ports

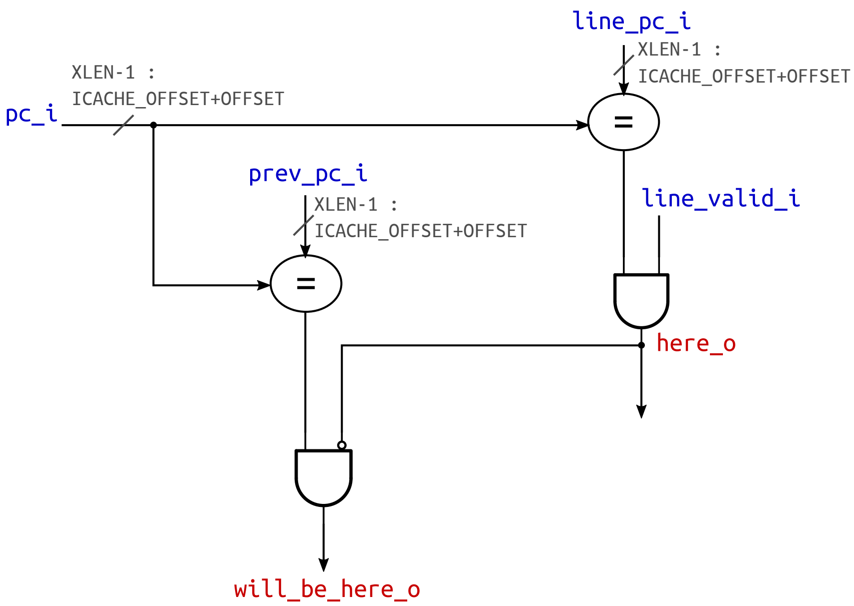


Figure 1.13: Presence checker combinational network

1.4.1 Presence checker

The presence checker block (see figures 1.12 and 1.13) 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 figure 1.13.

1.4.2 Instruction selector

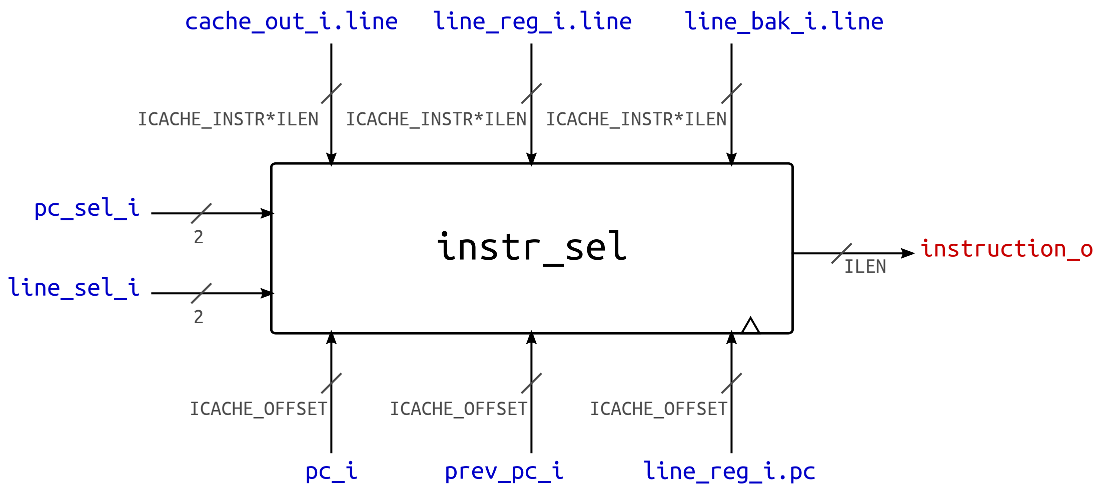


Figure 1.14: Instruction selector module ports

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 figure 1.14). It outputs a single selected instruction to be pushed to the issue queue.

Figure 1.15 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.

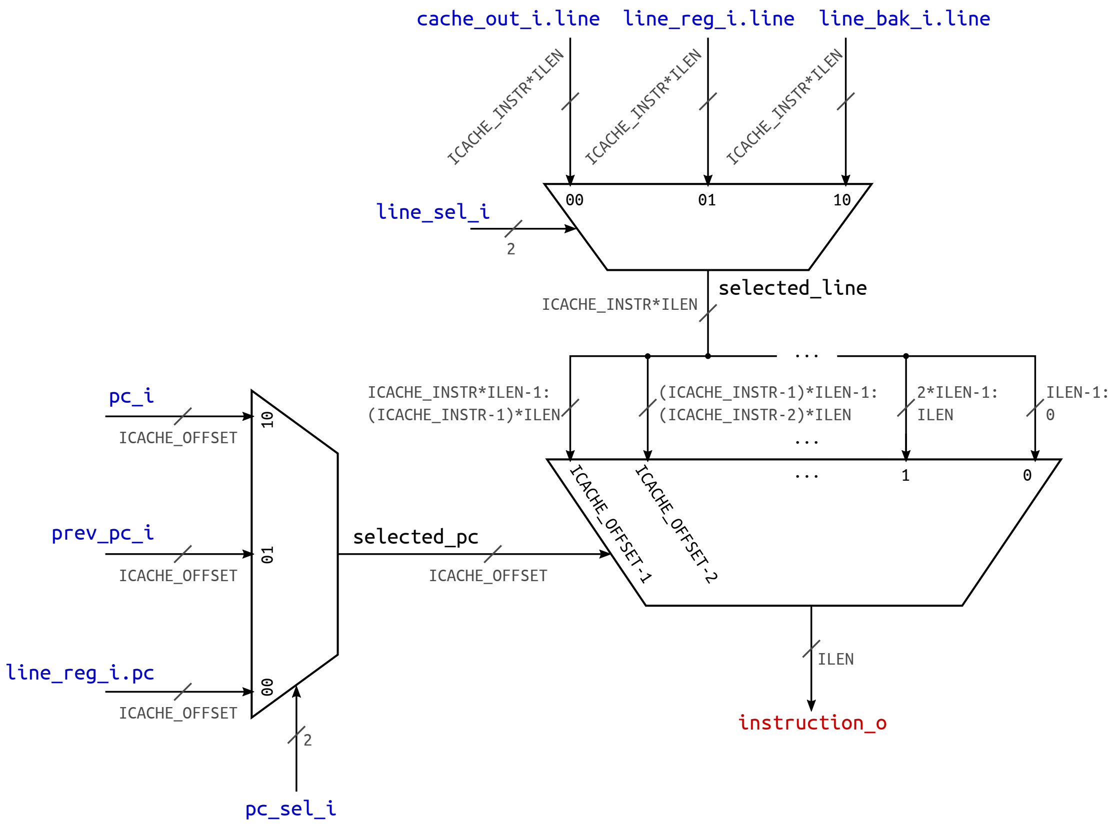


Figure 1.15: Selection network

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.

1.4.3 Fetch controller

The fetch controller module, whose interface is shown in figure 1.16, 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

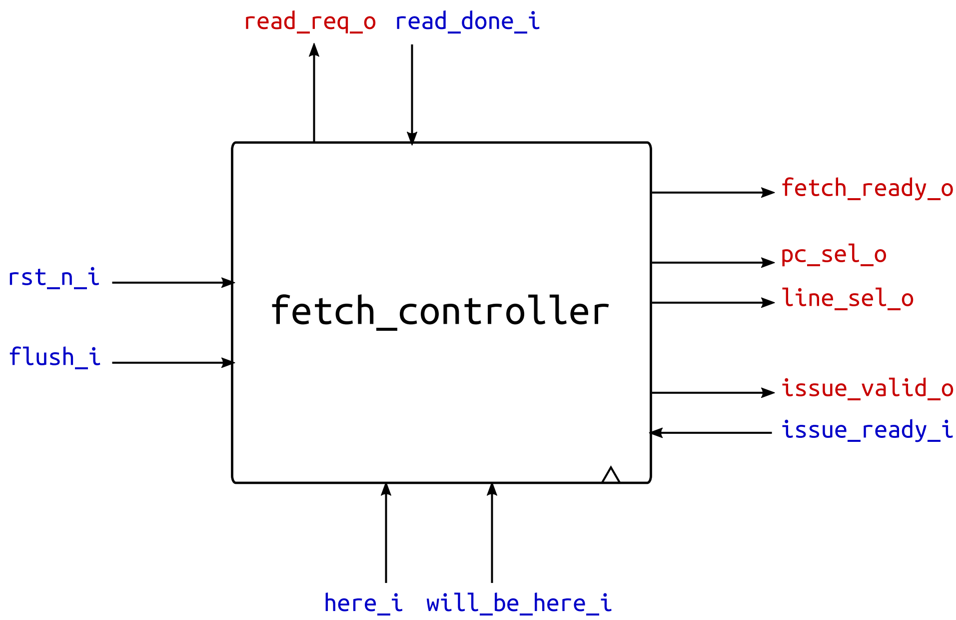


Figure 1.16: Fetch controller module ports

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 figure 1.4) and in the fetch stage as the enable of the previous PC register (see figure 1.11). 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 figure 1.17 manages. Here follows a summary of the purpose of each state:

- `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

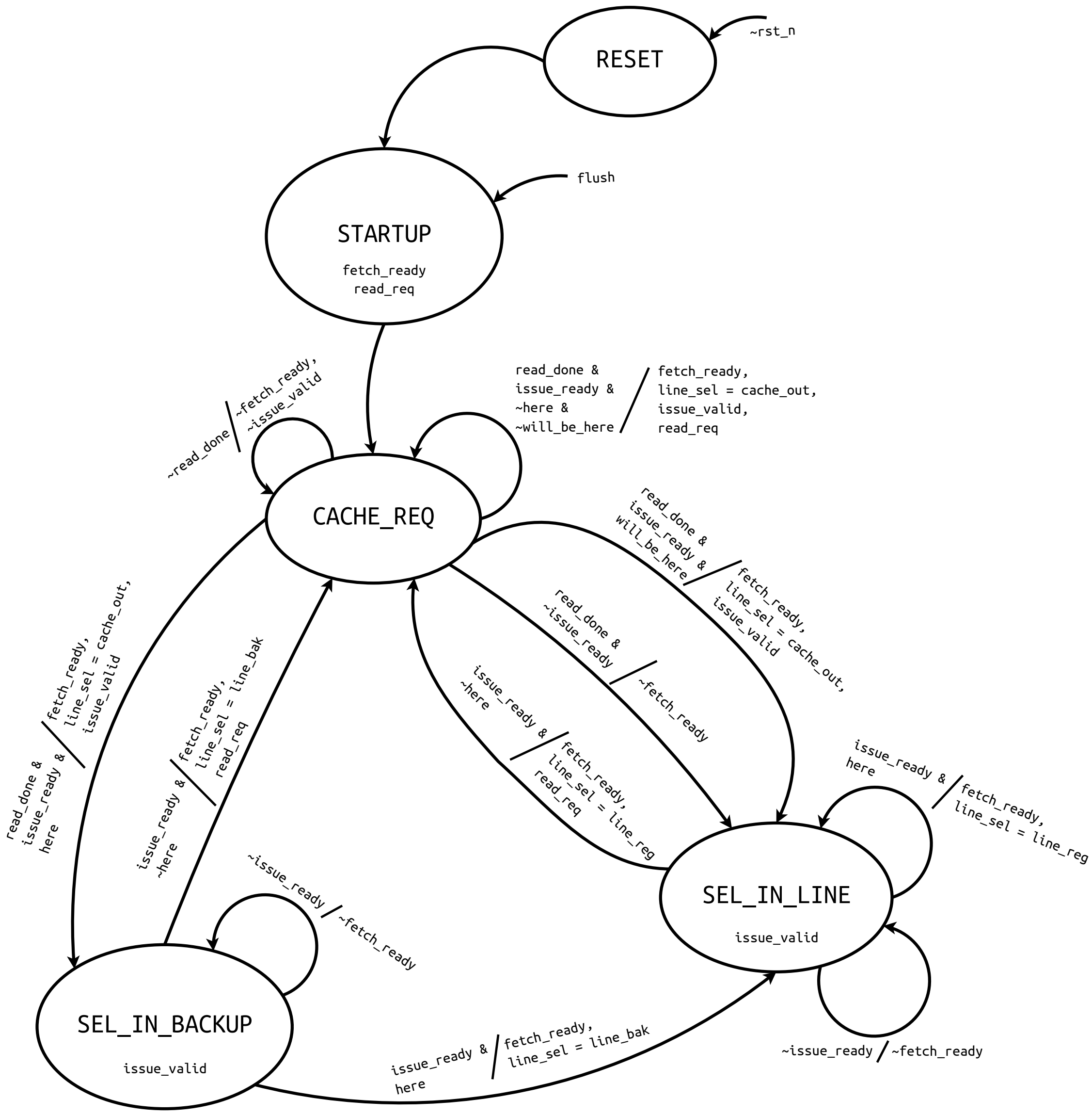


Figure 1.17: Fetch controller FSM

`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 figure 1.11) 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.

  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.

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.

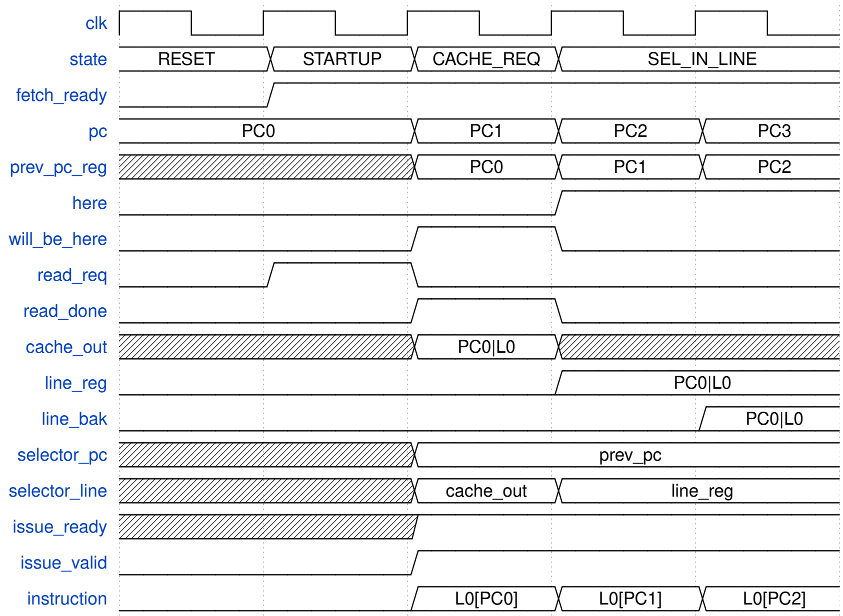


Figure 1.18: Startup and hit (`PC0` is the `BOOT_PC` considered above)

Figure 1.18 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 1.19 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 1.20 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.

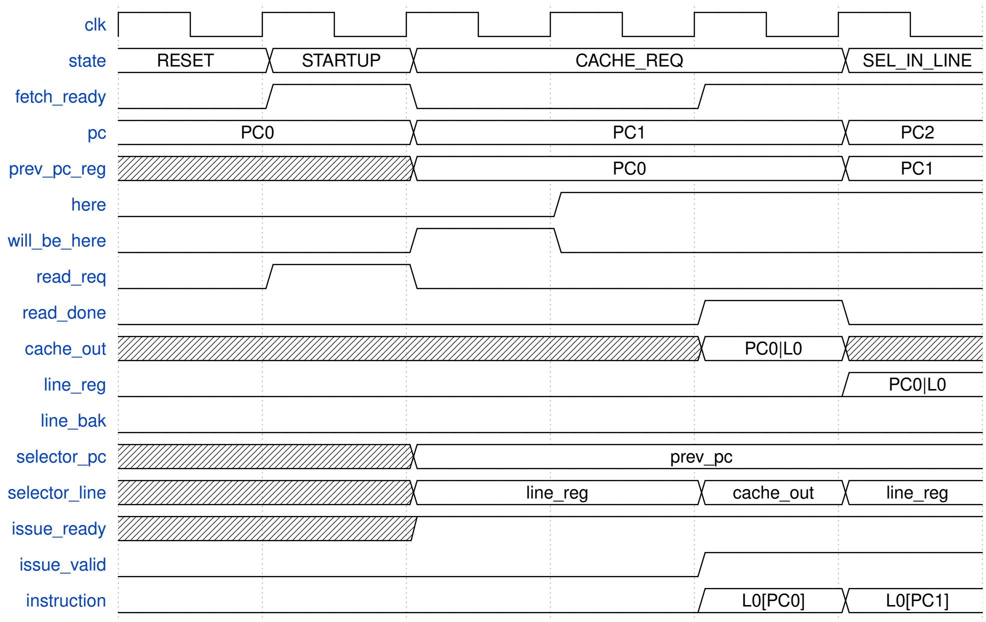


Figure 1.19: Startup and cache not ready/miss

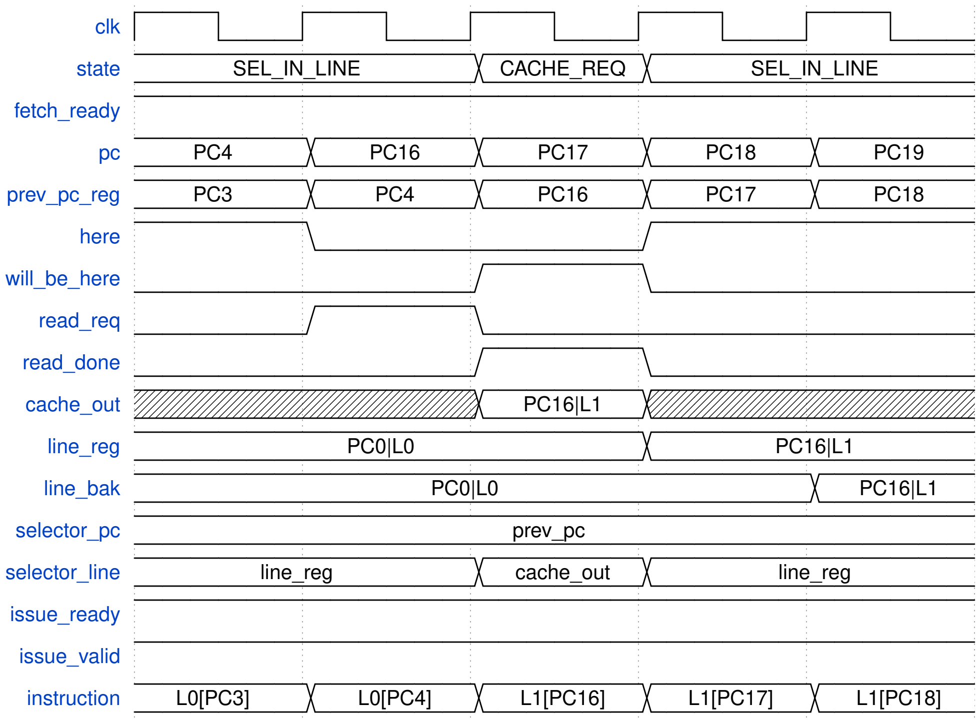


Figure 1.20: Saved line change

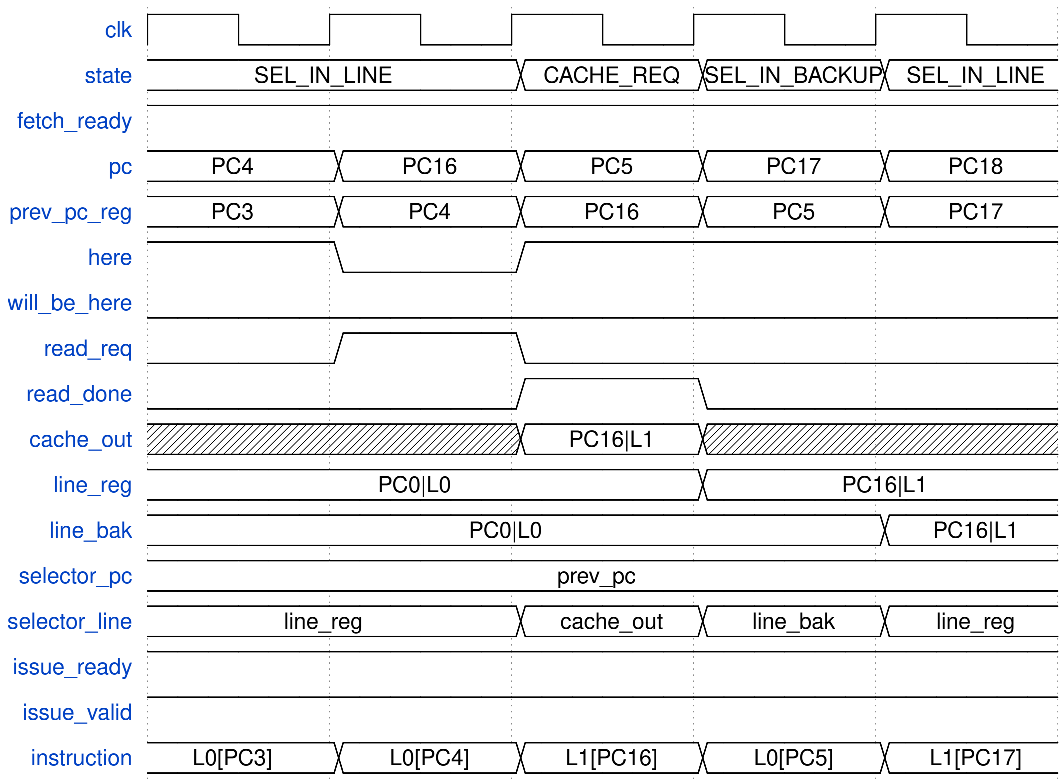


Figure 1.21: Line backup register purpose

The purpose of the line backup register is demonstrated in the timing of figure 1.21, 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 1.22 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.

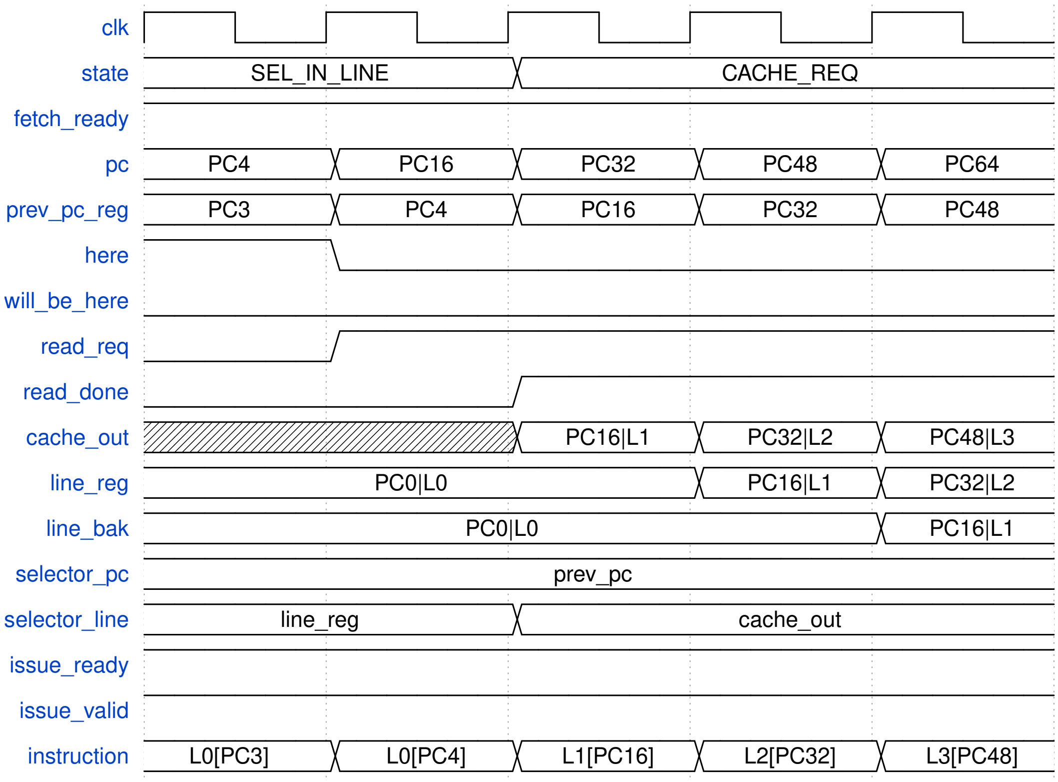


Figure 1.22: Cache read pipeline

Figures 1.23 and 1.24 show the issue queue being not ready at two different instants. In figure 1.23 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 figure 1.24, 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.

C'mon, solve this

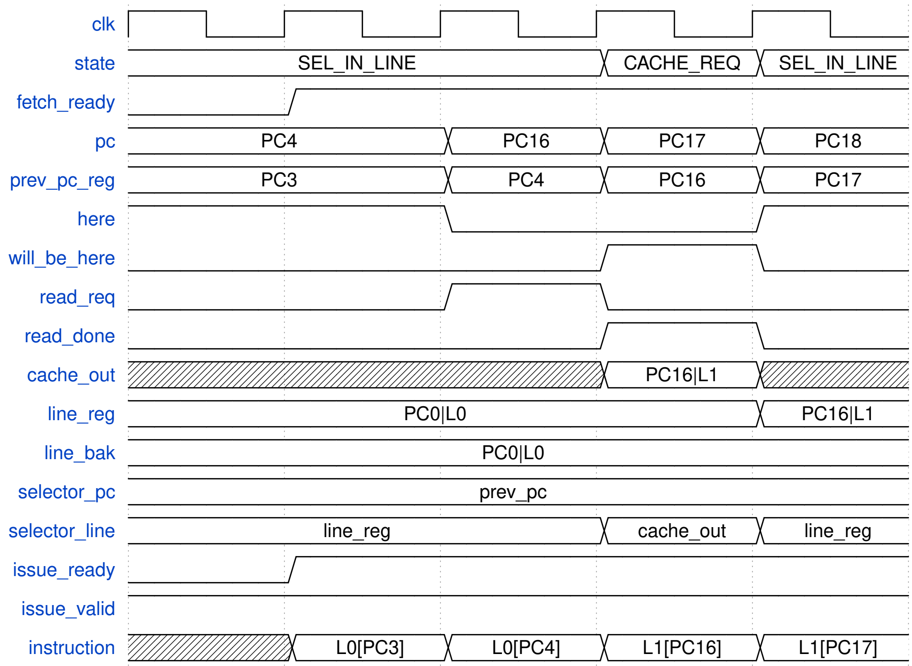


Figure 1.23: Issue queue not ready during selection

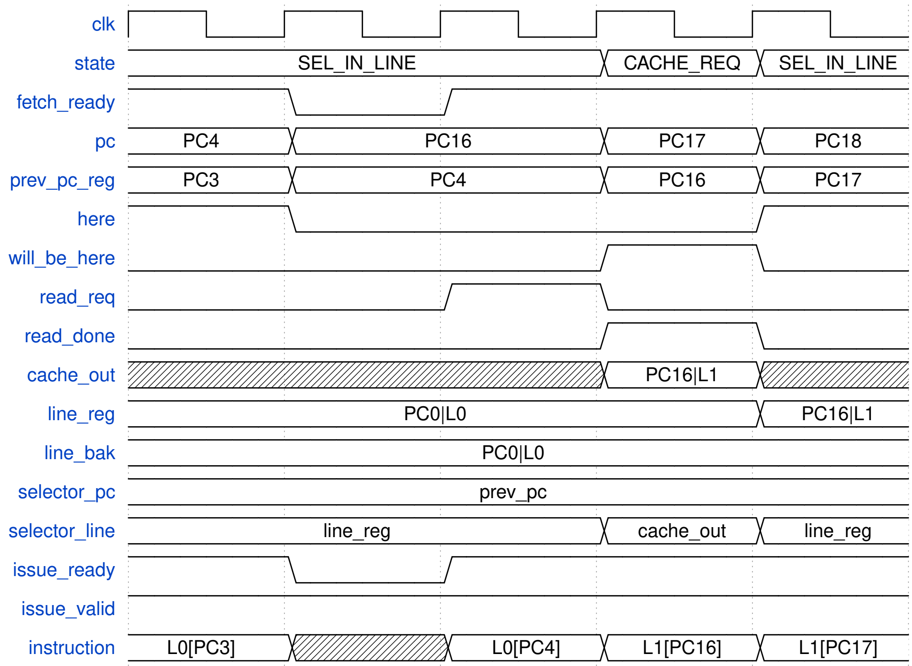


Figure 1.24: Issue queue not ready during request

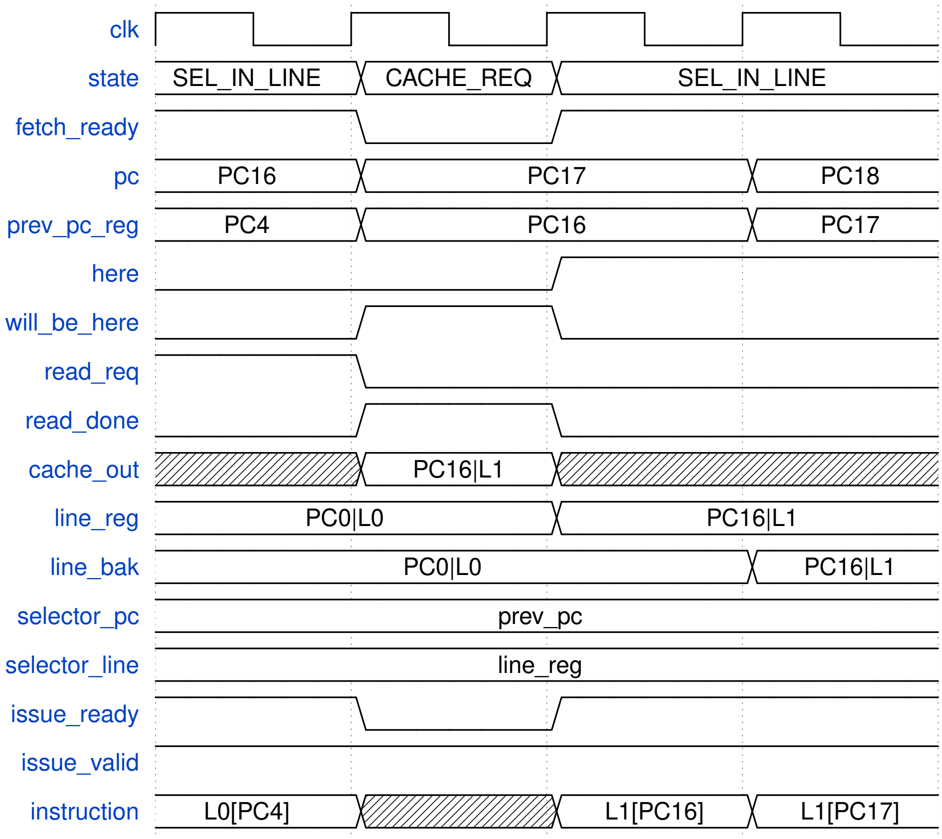


Figure 1.25: Issue queue not ready when cache line arrives

Figure 1.25 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 figure 1.26 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, figure 1.27 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 figure 1.23, here the stall does not cause any issues and the fetch resumes exactly as before when the queue turns ready again.

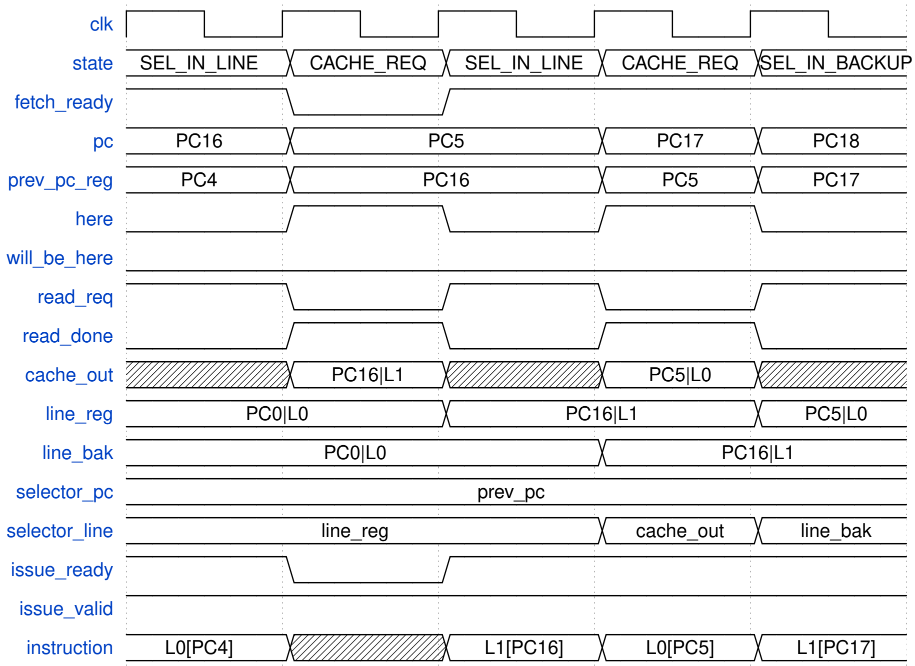


Figure 1.26: Issue queue not ready and causing loss of backup

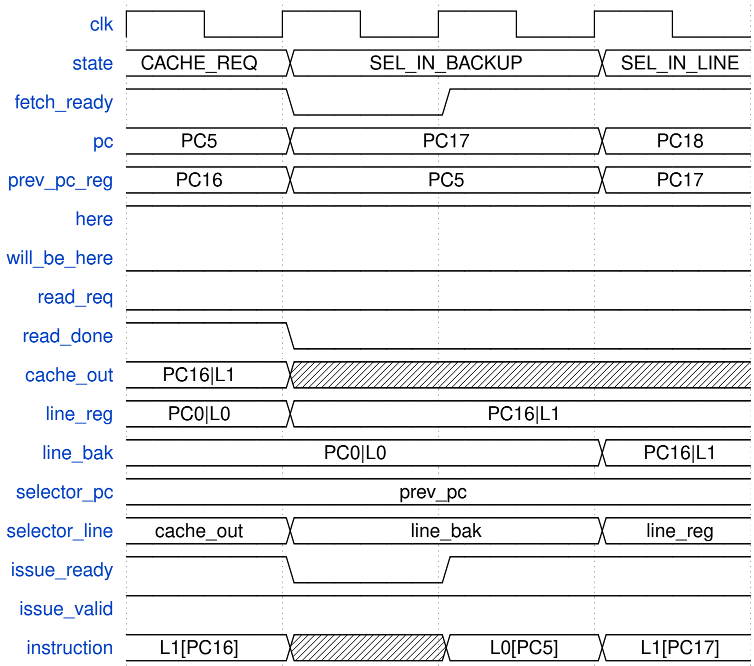


Figure 1.27: Issue queue not ready during backup

1.5 Branch Prediction Unit (BPU)

As seen in figure 1.1, 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 figure 1.28, is composed of two main blocks:

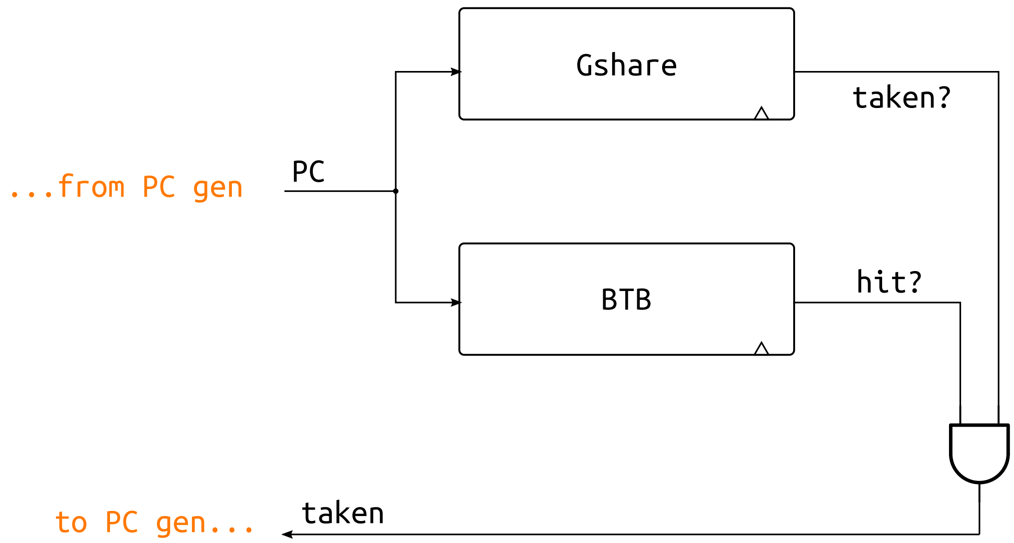


Figure 1.28: BPU general idea

- 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 section 1.5.2. 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 **Branch Target Buffer (BTB)**, described in section 1.5.3, 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.

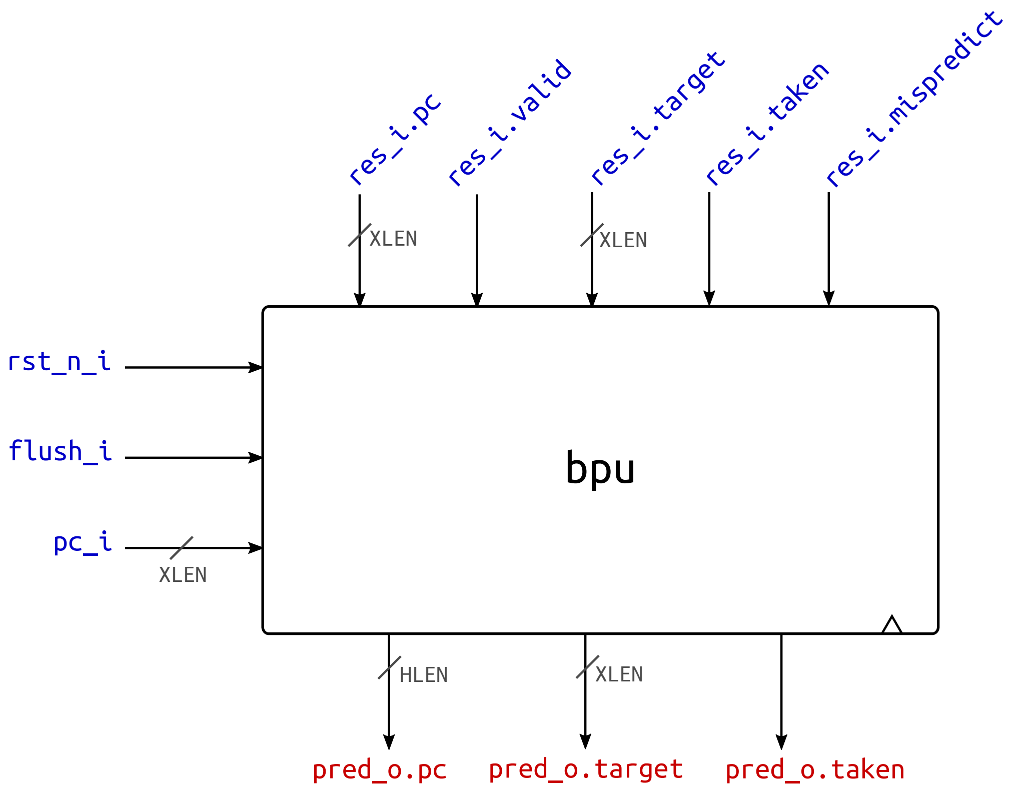


Figure 1.29: BPU module ports

1.5.1 Top-level block diagram

Figure 1.29 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 table 1.1. 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 1.1: 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.

Figure 1.30 shows the internal organization of the BPU which is simply composed of the gshare and BTB blocks, along with the AND gate of figure 1.28 and a couple of other gates to decide the correct update action based on the results, as in table 1.1.

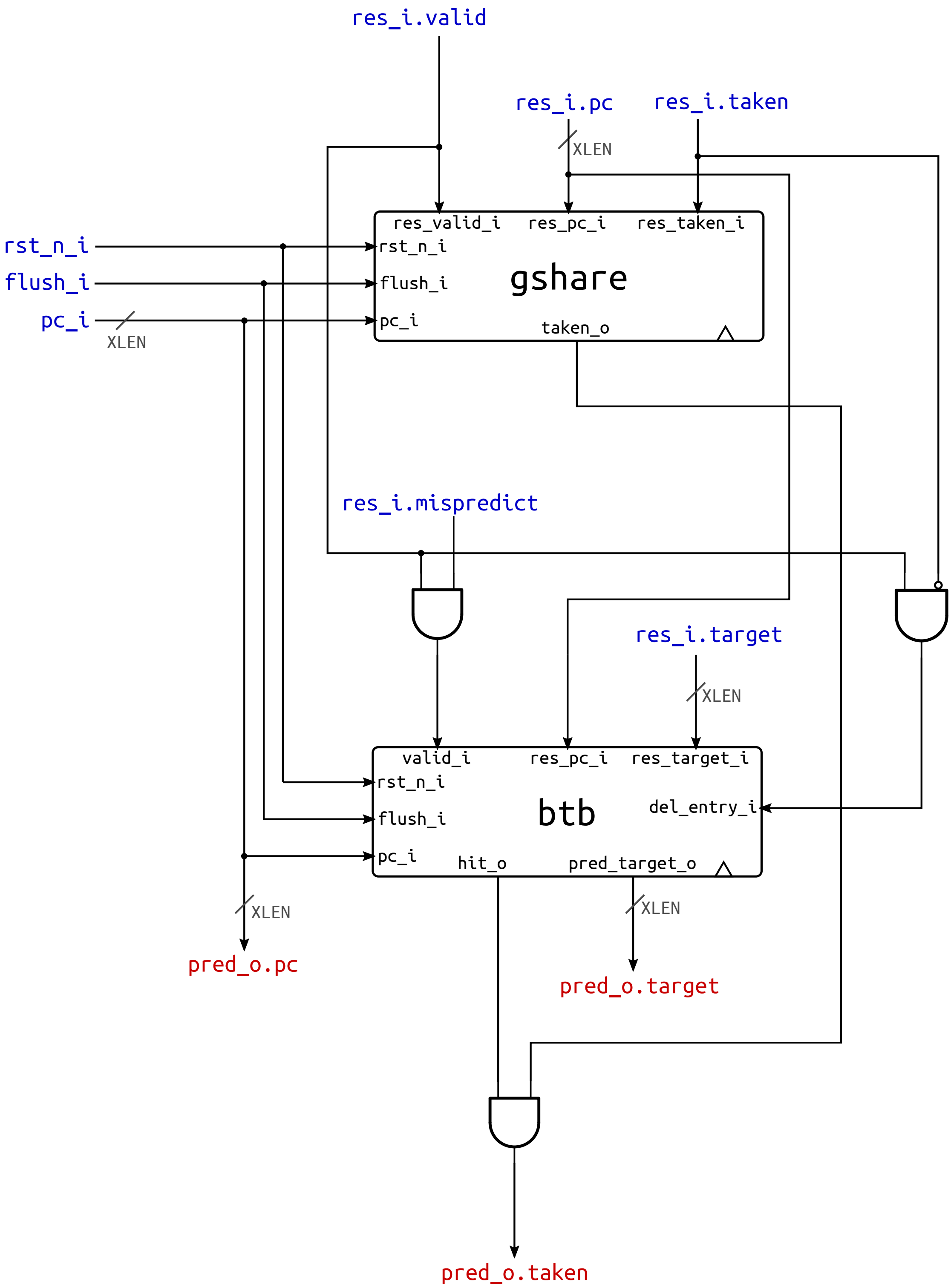


Figure 1.30: BPU block diagram

1.5.2 Gshare branch predictor

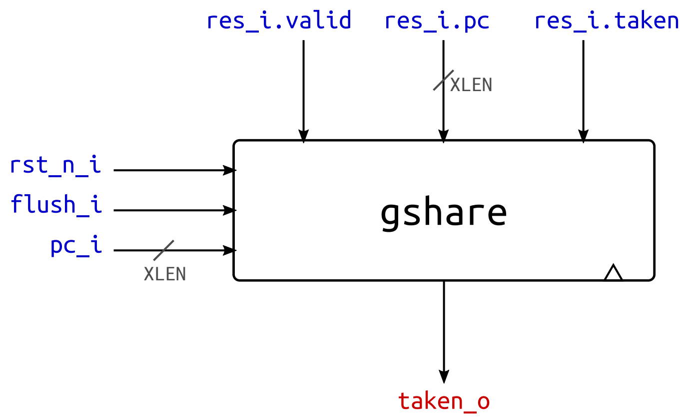


Figure 1.31: Gshare module ports

The idea of the gshare predictor, whose interface is shown in figure 1.31, was first proposed by Scott McFarling in a 1993 paper [14] 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 figure 1.32 and is composed of a Branch History Table (BHT) composed of a single shift register where branch resolutions are left shifted in and a Pattern History Table (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

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

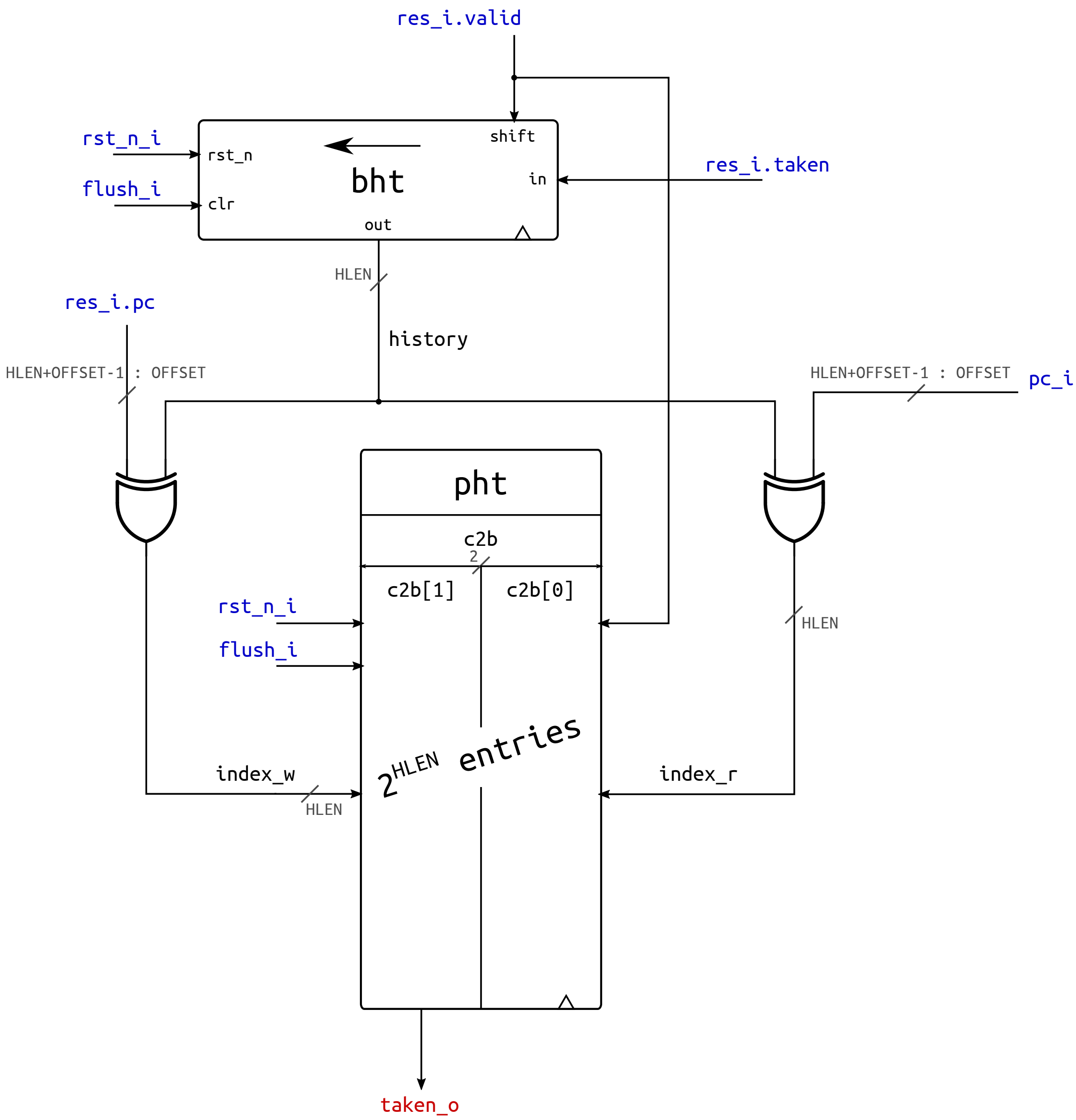


Figure 1.32: Gshare branch predictor

register. This poses no timing issues, as demonstrated by figure 1.33, 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

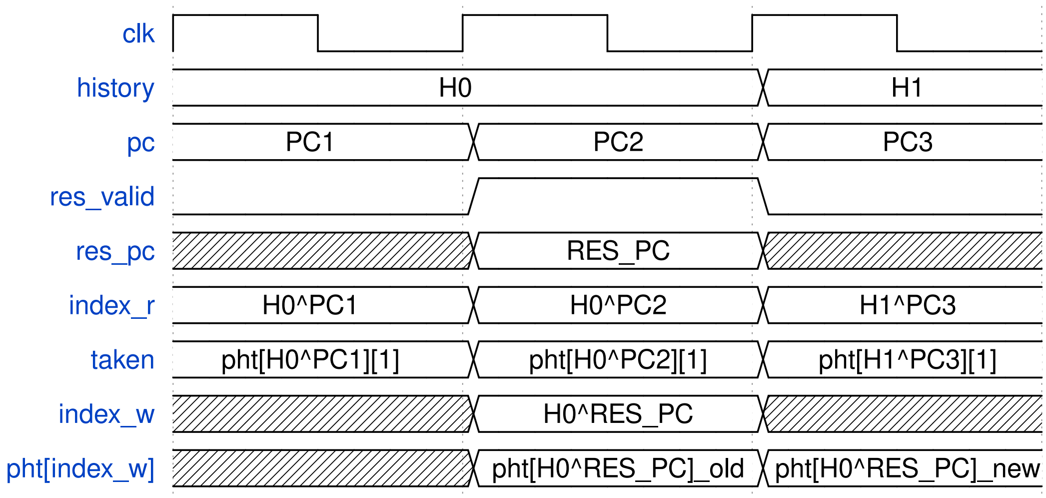


Figure 1.33: Update timing diagram

of data structures like the issue queue and the branch reservation stations by not 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.

1.5.3 Branch Target Buffer (BTB)

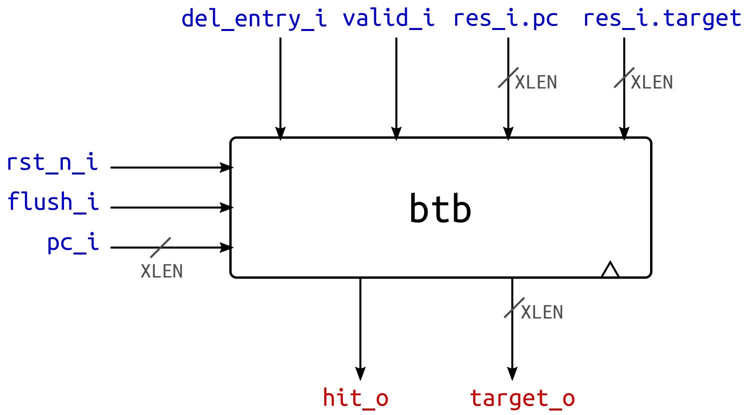


Figure 1.34: BTB module ports

The purpose of the BTB (interface in figure 1.34) 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 [15] and was since optimized and employed in a great number of processor designs.

Its structure shown in figure 1.35 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.

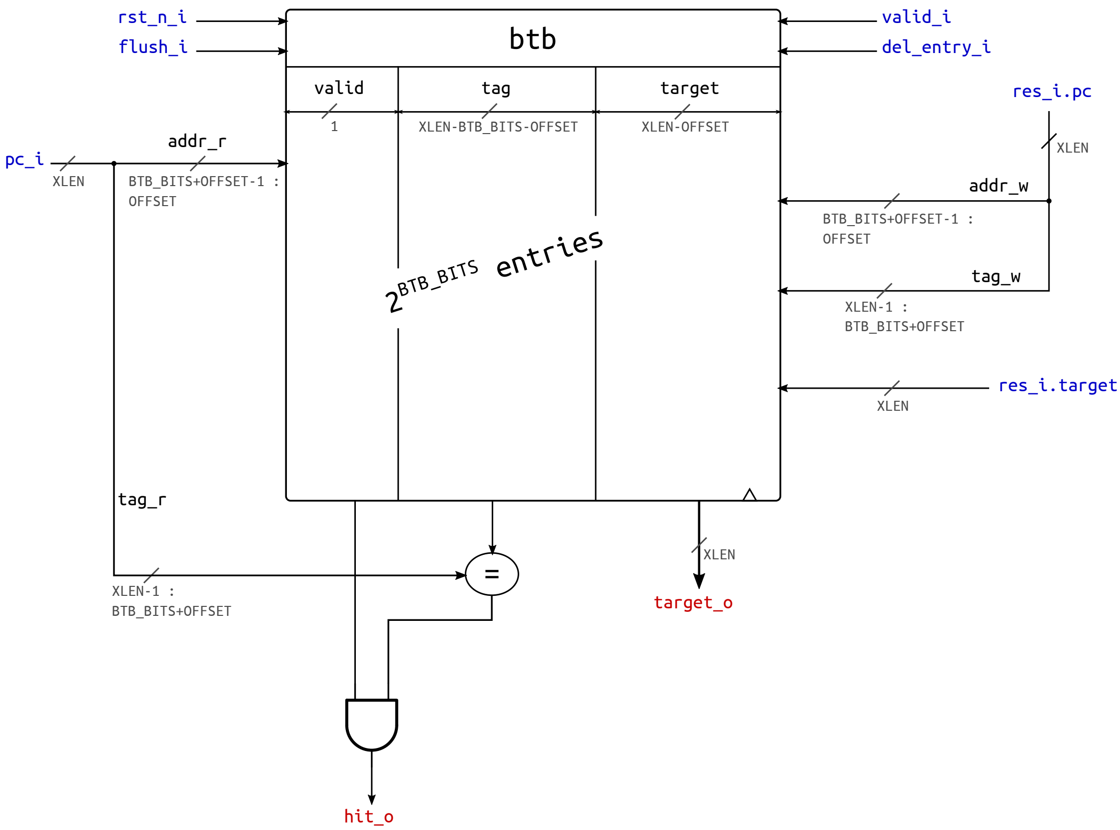


Figure 1.35: BTB diagram

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 [16]. 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 section 1.5.2 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.

1.6 Branch unit

A unit that was not talked about until now and that is not shown in figure 1.1 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.

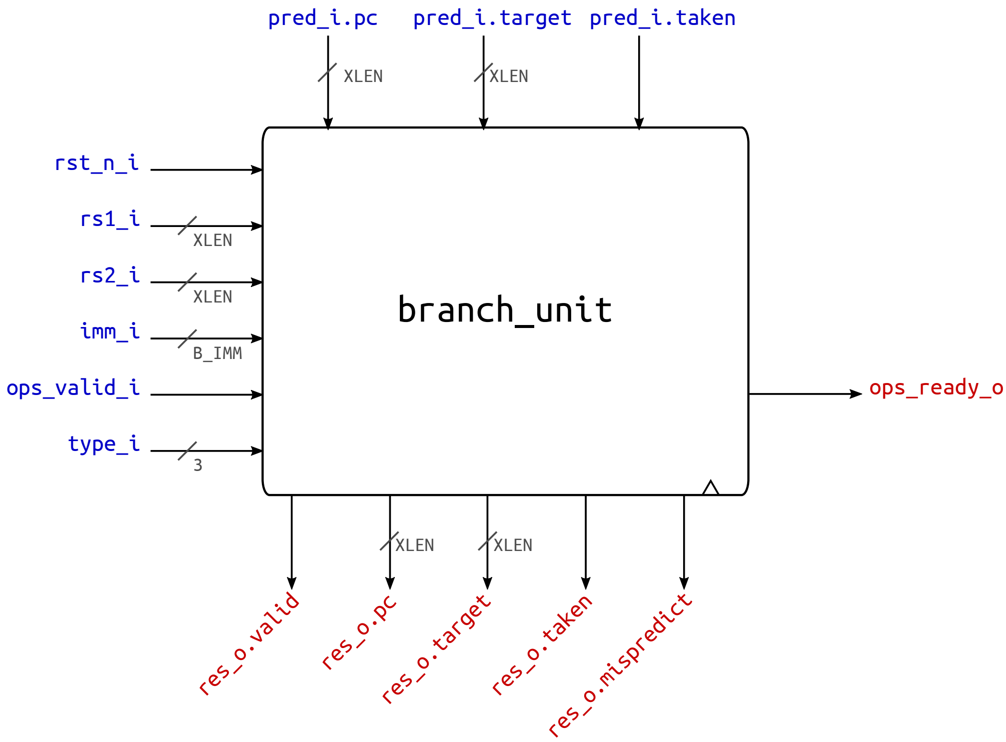


Figure 1.36: Branch unit module ports

Figure 1.36 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.

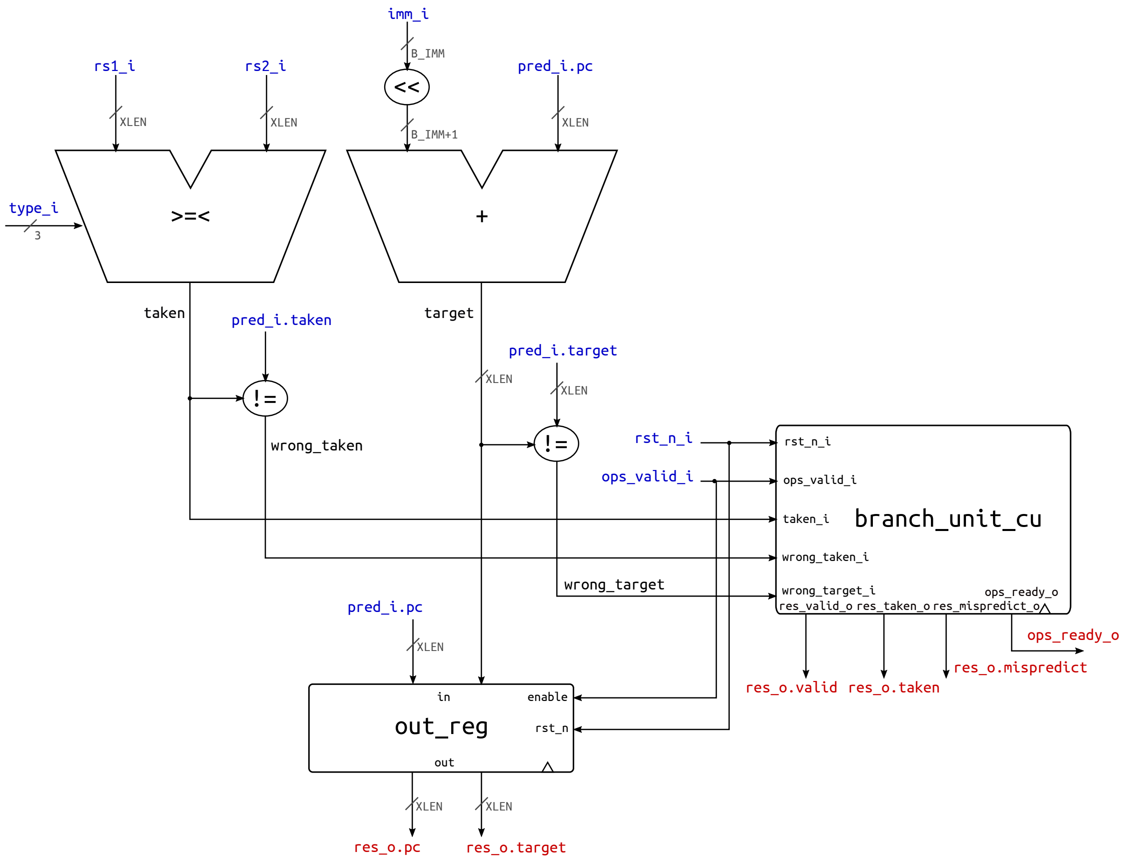


Figure 1.37: Branch unit datapath

1.6.1 Datapath

The datapath of the branch unit is shown in figure 1.37 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 Instruction Set Architecture (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.

1.6.2 Control unit

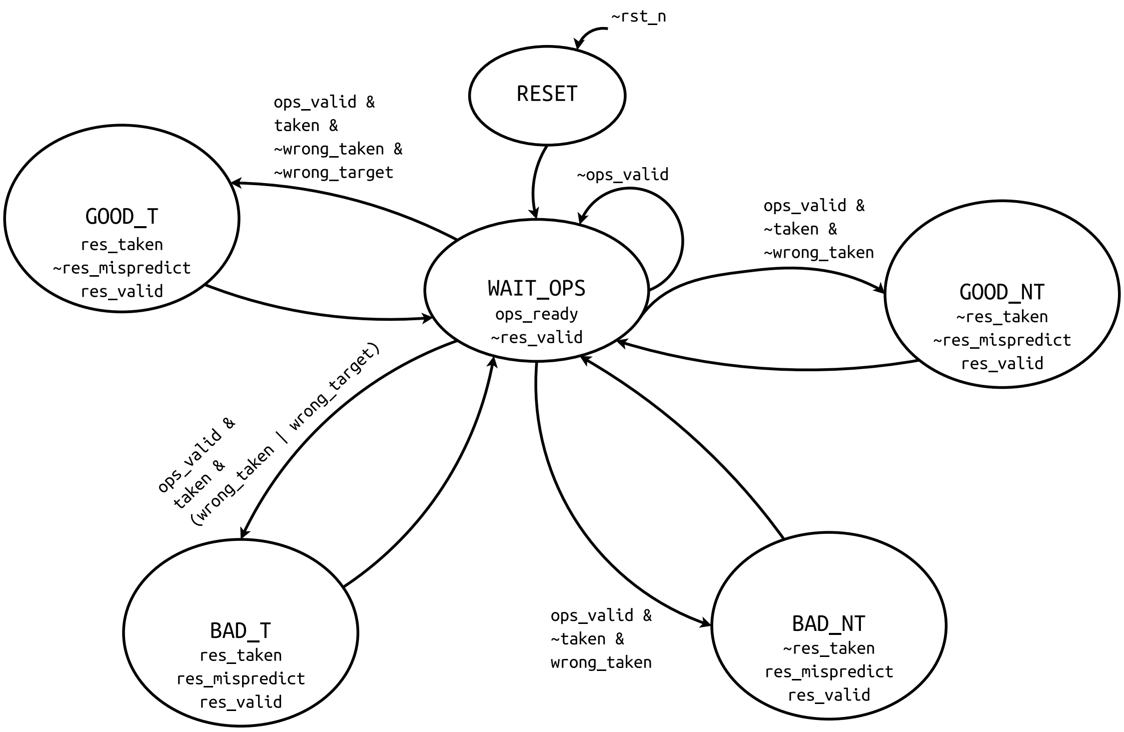


Figure 1.38: Branch unit control unit

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.

Aggiungere timing dopo modifica

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.