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

BHT Branch History Table

BPU Branch Prediction Unit

BTB Branch Target Buffer

FPGA Field Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

PC Program Counter

PHT Pattern History Table

Chapter 1

Experimental results

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

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

1.1 Simulation

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

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

The following sections focus on the test strategies used for the Branch Prediction Unit (BPU), which is almost as important as a standalone design, and the top-level frontend.

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

1.1.1 BPU

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

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

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

Single loop

The first test case corresponds to the following simple loop:

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

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

Suppose that the instruction testing the loop condition is at address 10 and the beginning of the loop body (i.e. the target of the branch instruction) is at address 24. Figure 1.1 shows the simulation waveforms for this test case, with the predictions contained in the `pred_o` signal, occurring each time the PC 10 is read from the address file, and the resolutions read into `res_i` every time the `valid` signal is asserted.

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

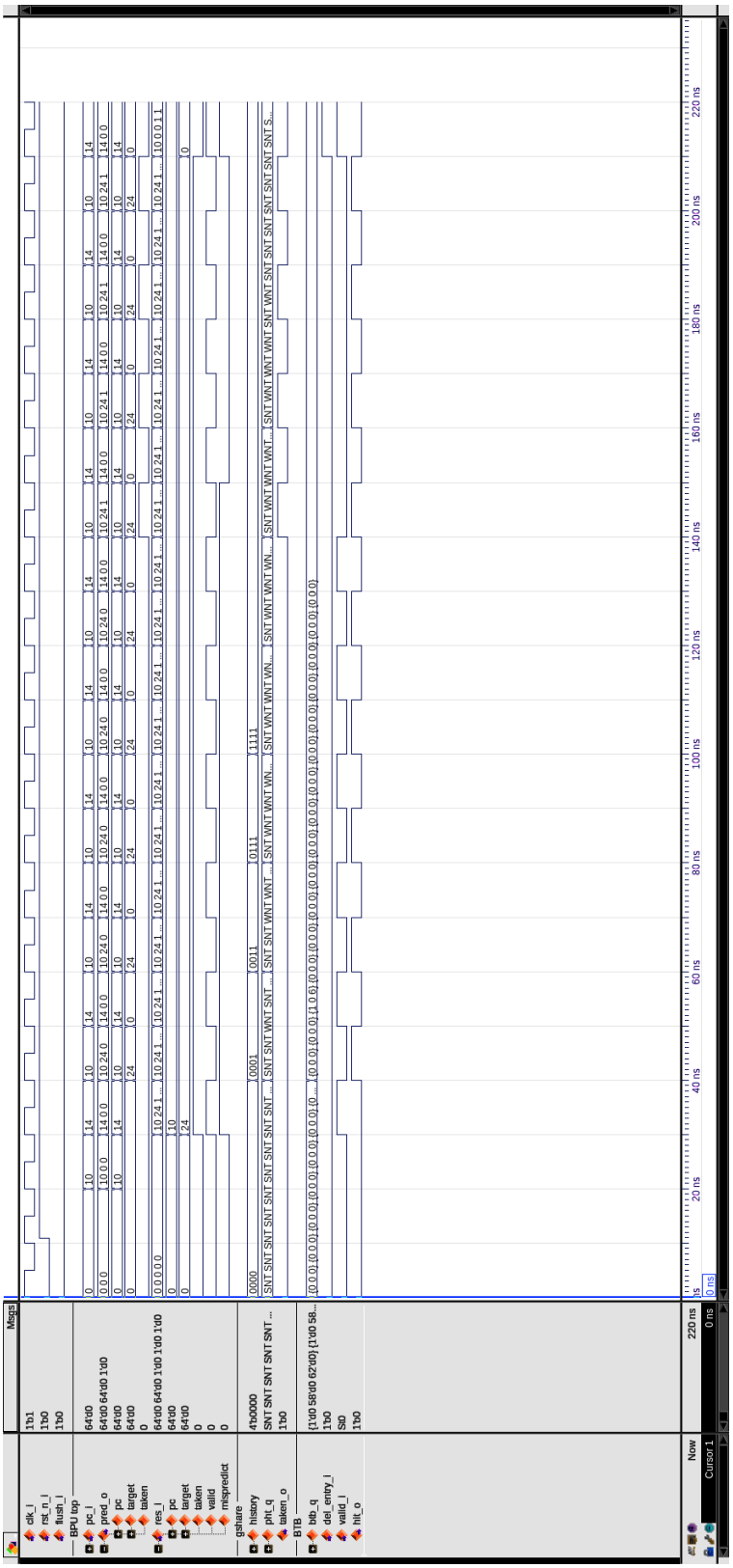

Figure 1.1: Single loop simulation

At the seventh iteration of the loop, the branch is predicted correctly as taken (the `mispredict` signal is deasserted) and this situation lasts until the the loop condition is tested false at the last iteration, leading to a new misprediction.

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

This *warm-up* of the predictor is intrinsic of its design and cannot be avoided, but anyway figure 1.1 demonstrates the correct and expected behavior of the BPU.

Nested loops

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

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

This example, actually taken from [14], is intended to demonstrate that the gshare predictor, after the warm-up, can correctly identify taken branches in nested loops, where the outer loop is repeated many times.

Figure 1.2 shows the simulation results for this case, where the address of the condition instruction of the outer loop is 10, the one of the inner loop (i.e. the target of the outer loop) is 80 and the body of the inner loop starts at address 24.

At the beginning, the gshare continuously mispredicts the outer loop and the first iterations of the inner loop, due to the initialization of the PHT as mentioned in the previous case, but then after the warm-up it goes on to predict correctly both loops, until the exit of the outer loop. In particular, the steady state situation is shown in table 1.1, where each combination of address and history univocally determines the prediction outcome.

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

Table 1.1: Nested loops steady state predictions

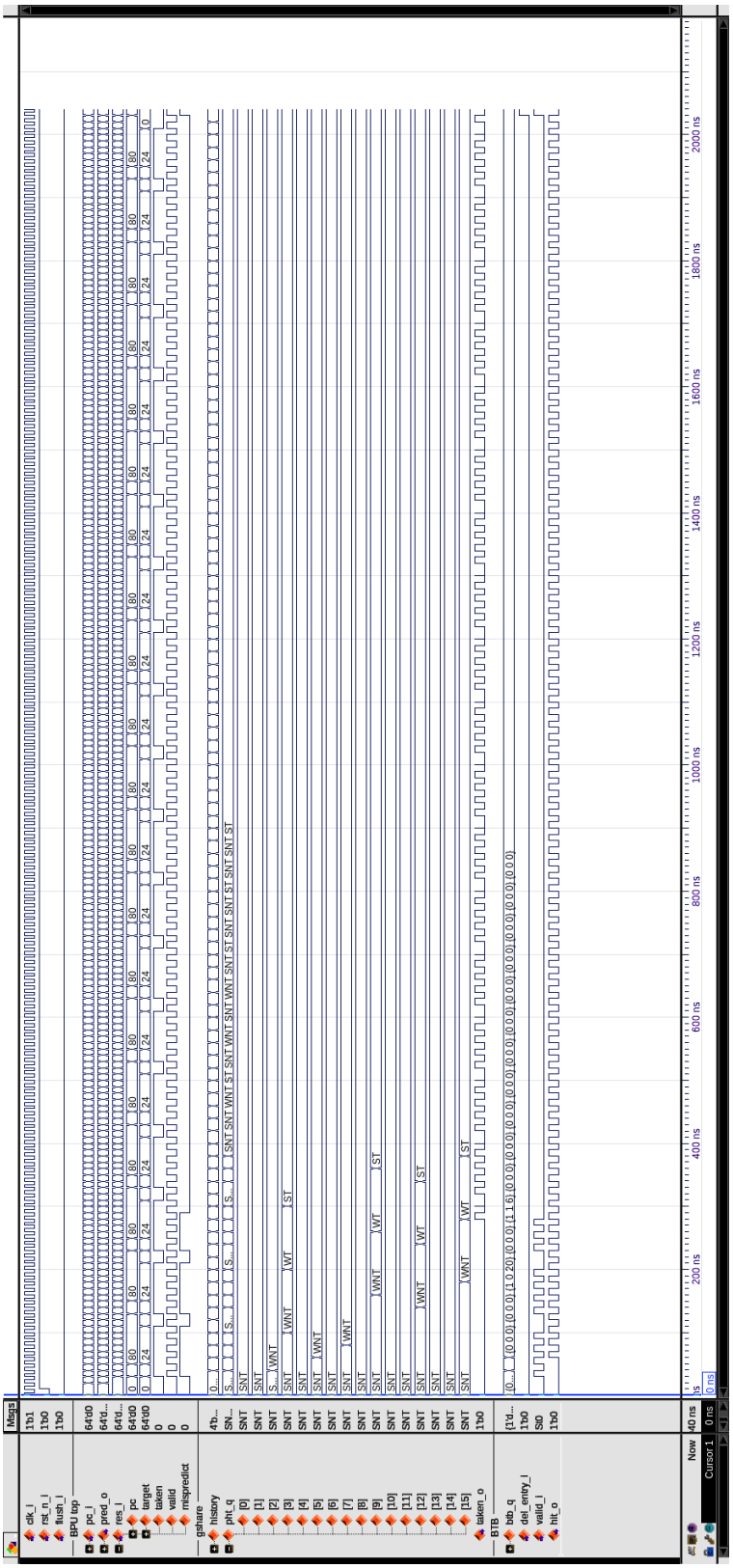

Figure 1.2: Nested loops simulation

1.1.2 Frontend

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

- A *PC jumper* used to simulate exceptions and branches by modifying the sequential generation of addresses.
- A *dummy instruction cache* which responds to memory access requests by simulating both hits and misses, with random delays. The fictional data line it provides always contains the PC that generated the request and N instruction fields with the number from 1 to N in order to track the movement of instructions.
- A *dummy issue queue* that simply simulates a busy issue queue by introducing random delays on the `issue_ready` signal.

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

Figure 1.3 shows the standard situation where subsequent instructions are selected among the same line, saved in the line register after the first memory access at startup (compare with **??**).

Figures 1.4 and 1.5 show the situation of a cache not ready to receive the address or a cache miss respectively, as in **??**. Note also the current states of the instruction cache interface FSM that correspond to the timing diagrams of **????**[2].

Figure 1.6 shows a sequence of cache reads in a pipeline fashion, just as in **??**. Note also how here there is a jump right after the boot address, which is correctly handled by the instruction cache interface.

Figure 1.7, like **??**, show the case when the instruction is selected among the line backup register, due to a jump back and forth to the same cache line.

Finally as an example, figure 1.8 shows the situation in which the issue queue is not ready during instruction selection (compare with **??**).

1.2 BPU benchmarking

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

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

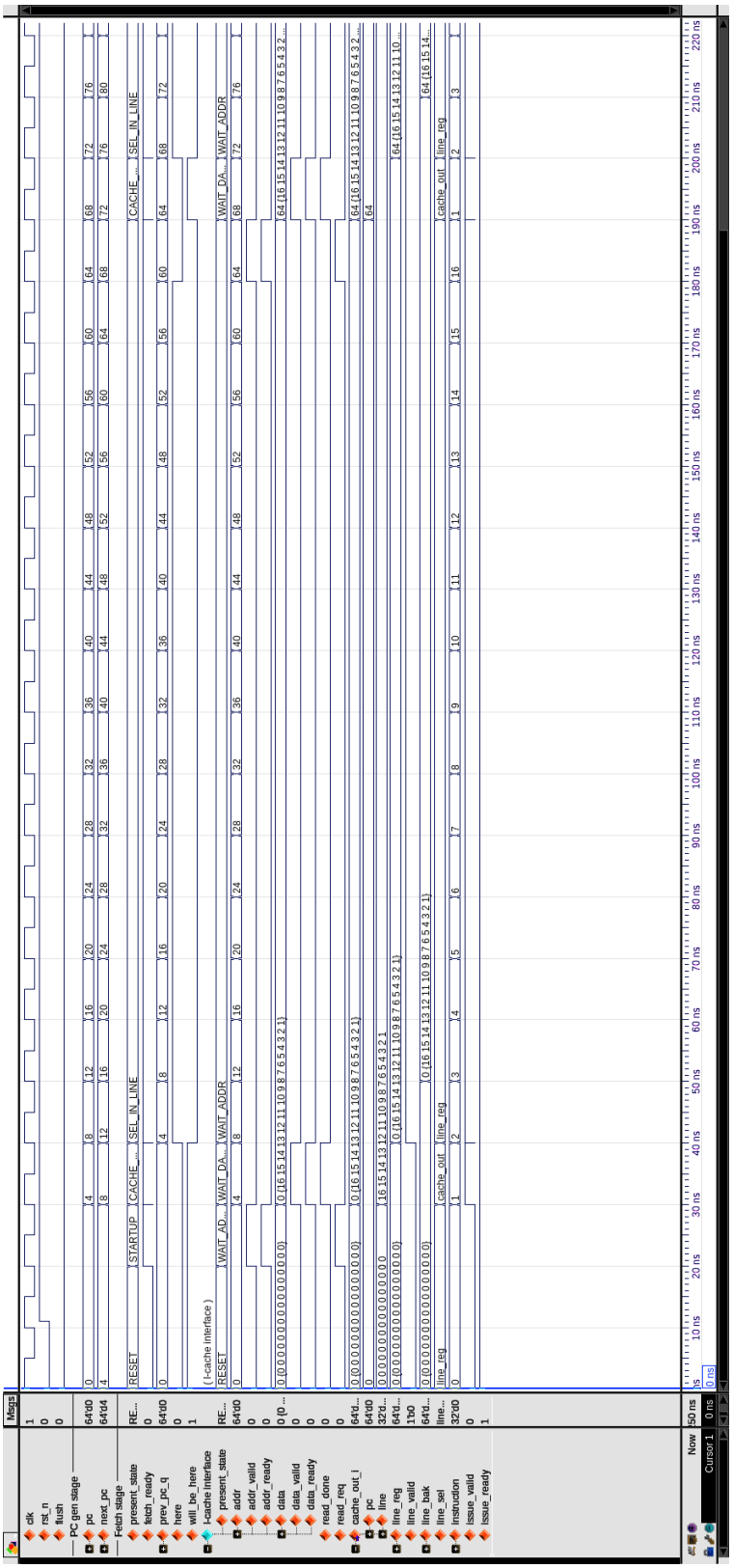

Figure 1.3: Sequential reads from the same line

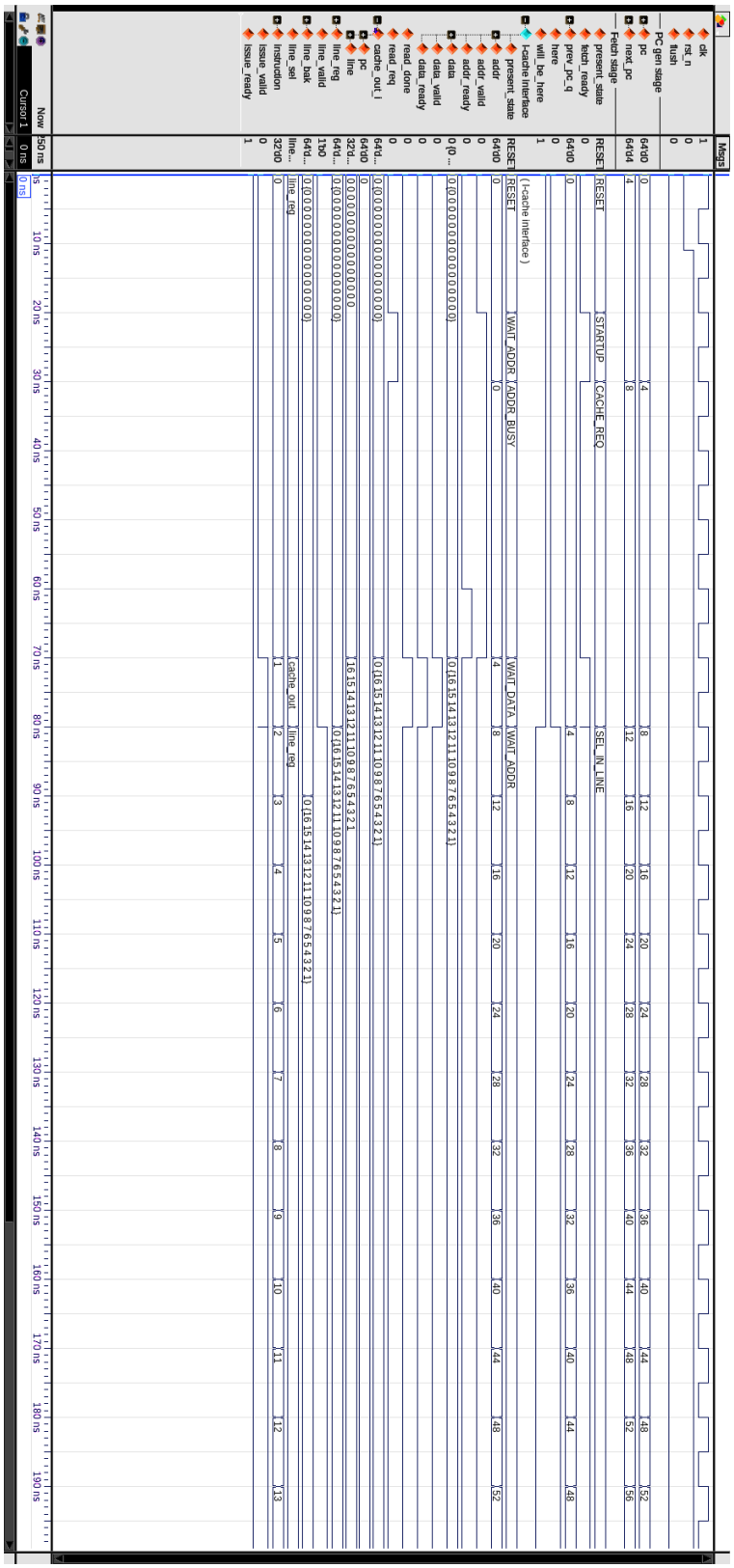

Figure 1.4: Cache not ready on address

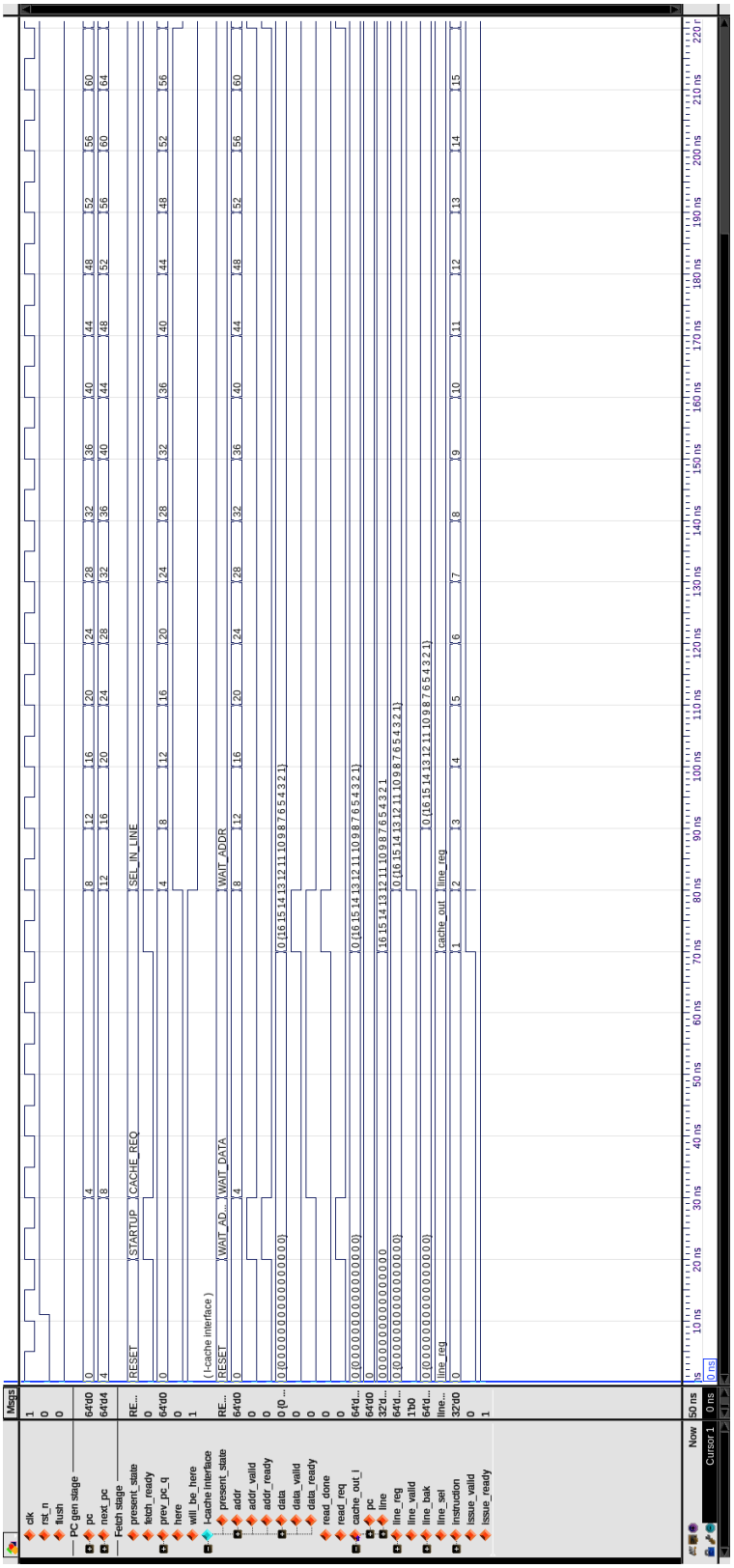

Figure 1.5: Cache miss

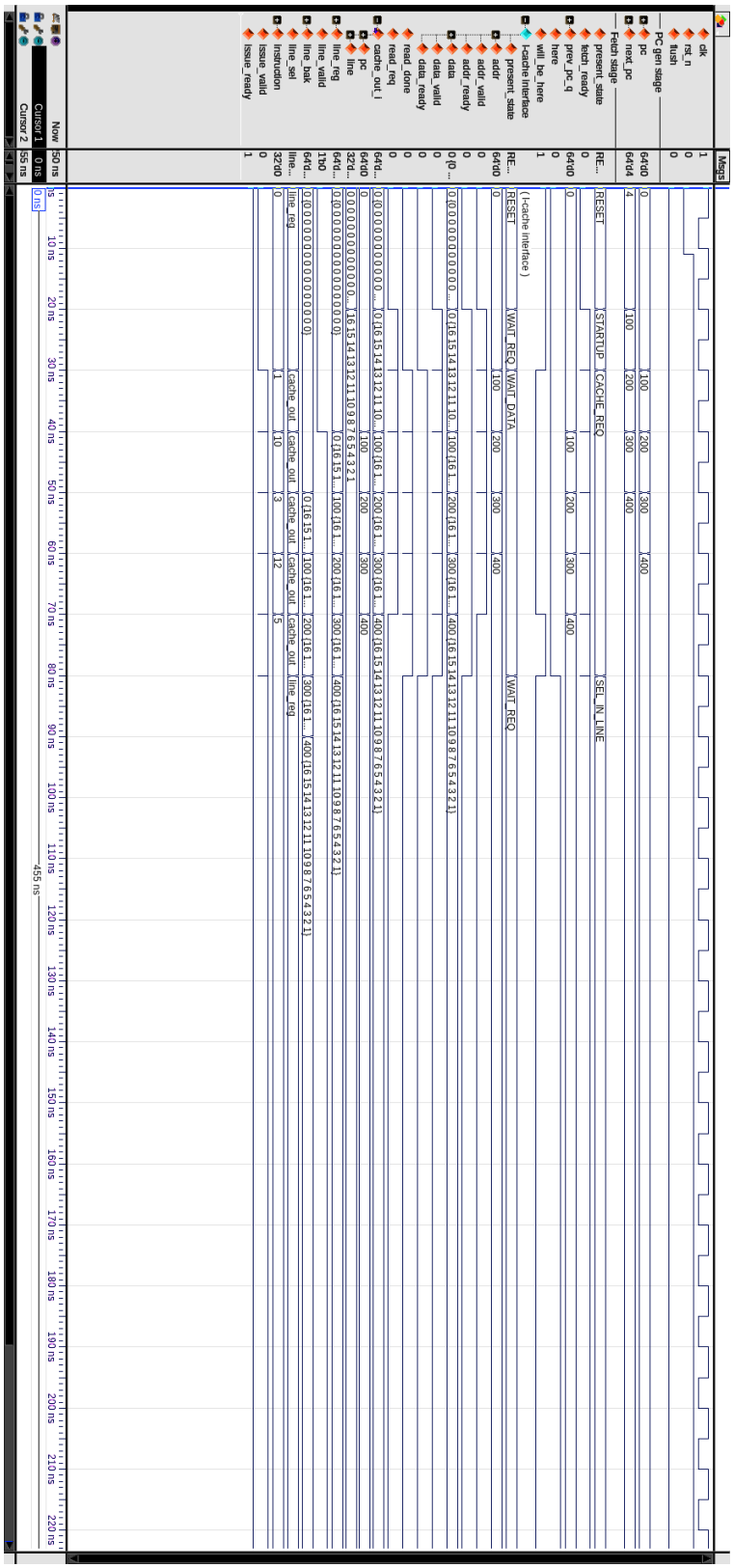

Figure 1.6: Cache read pipeline

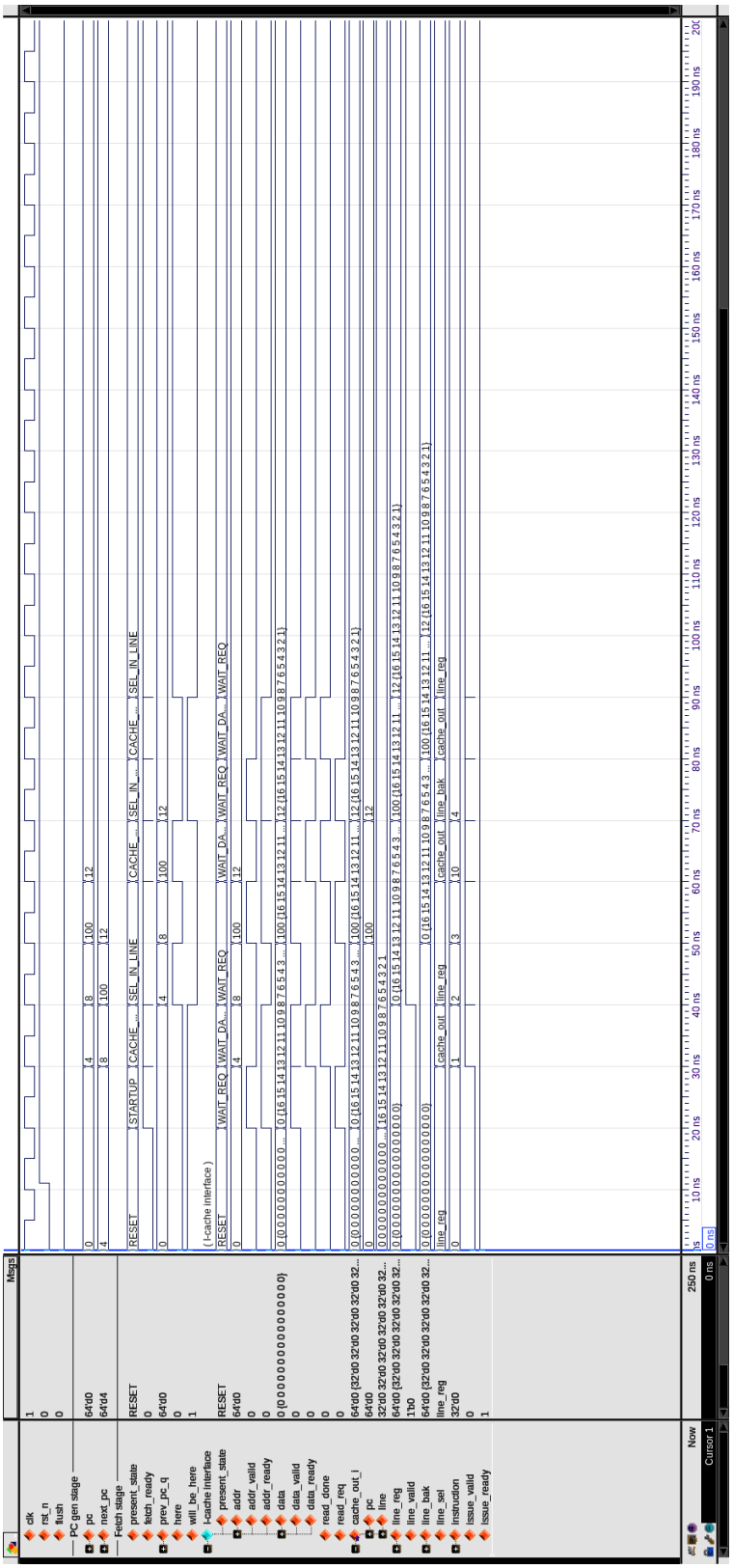

Figure 1.7: Line backup register selection

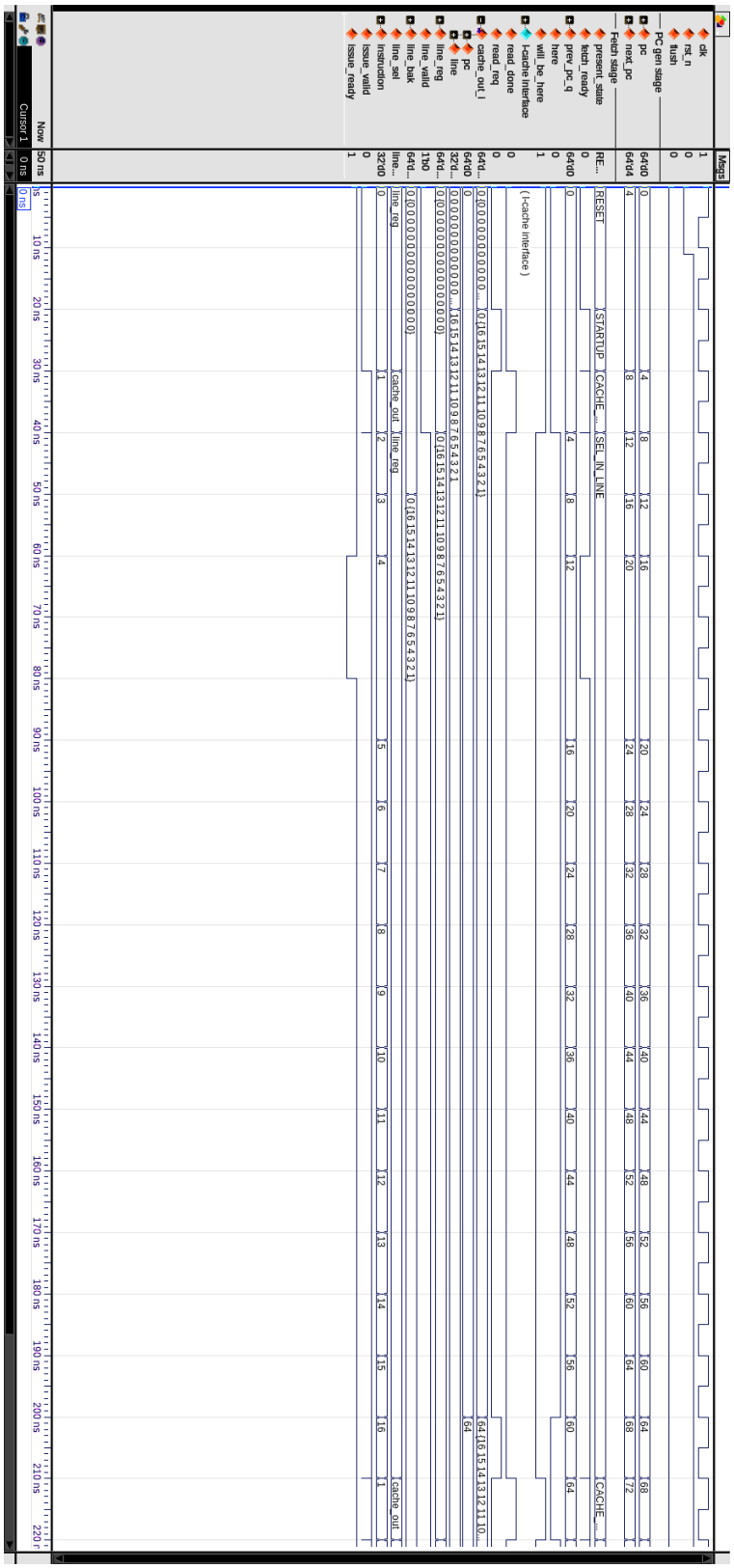

Figure 1.8: Issue queue busy

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

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

After significant efforts spent to find a way to extract *branch traces* (i.e. the list of branch instructions and their result) from a compiled program, no feasible solution was found and so a decision was made to rely on the trace files provided by a laboratory exercise of the course *Principles in Computer Architecture* held by Prof. Dean Tullsen of the University of California San Diego, available on GitHub[3]. These traces come from a series of benchmarks taken from the SPEC suite, listed in 1.2.

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

Table 1.2: BPU benchmarks

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

1.2.1 Gshare

1.2.2 BTB

1.3 Synthesis

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

Chapter 2

Concluding remarks

2.1 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.

[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.