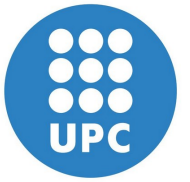

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Processor Design
(PD)

Insertion in pipelined CPU

Lab 5 Report

Jordi Solà, Ying hao Xu

December 18, 2018

Contents

1 Introduction

The course is about Processor Design and the initial goal of the project was to implement a module that extends the features of an already designed processor. This includes learning an HDL (Hardware Description Language) language, design the module and carefully reason about the design decisions we are taking.
In our case, we wanted to go further and implement a more complex accelerator that can be attached to any processor. Following this decision, we end up building a Systolic Array for accelerating the matrix matrix multiplication operation.

1.1 Motivation

Systolic Arrays are not new, they were introduced by Kung *et al.* in 1978[1], these systems are composed of several small processing elements (PEs) interconnected collaborating in order to perform different computations in a highly parallel way. The control is only involved in the injection timings of the data and the data flows through the computing units in rhythmic pulsations, hence the analogy with the human heart systoles.

Currently, the Systolic Arrays are becoming more and more popular as they fit really well with the Artificial Intelligence workloads. On the other hand, during a summer course given by *Yale Patt* in 2018, we presented during class the Laconic[2] deep learning accelerator, which its core unit is a Systolic Array. This motivated us to accept the challenge and design a systolic array from scratch in this project.

1.2 Objectives

The main objective of this project is to design an accelerator capable of computing a Matrix Matrix multiplication in the most efficient way. Therefore power, area and performance were considered too. In order to get the most of the accelerator, the design focused on reducing as much as possible the gap between two operations, letting the unit to start a new operation before the previous one has finished. On the other hand, we also wanted to document every decision we took and the discussions we had during the implementation of the unit.

From the point of view of RTL (Register-Transfer Level) design, one of the objectives was to write code that uses parameters and facilitates the code understanding. To accomplish this task, we leveraged on the `struct packed` type available in SystemVerilog and used macros when it was convenient (e.g. for the flip-flops).

We believe that this project can be really useful for other people, therefore one of our objectives was to make it public. For this reason we released all the source code in Github[1].

2 Methodology and tools

As part of the methodology, for every module we develop, there is a test bench that ensures the correct functioning of the unit. Following a bottom top approach, every module has been tested individually, meaning that we started testing the most basic elements (e.g. PE) and then building more complex units that will instantiate the smaller components and then test them again. The testing continues until we get to the top level entity that instantiates all the modules.

For the simulation, we used Verilator[2], which is an open-source HDL simulator capable of running synthesizable SystemVerilog code. By using verilator, we can generate waveforms in FST (Fast Signal Trace) format. The resulting waveforms can be visualized afterwards using GTKWave, another open-source tool for visualizing traces.

[1] `https://github.com/ying27/matrix-multiplication-accelerator`
[2] We used the version 4.008 of verilator that can be found in the official website.

In order to ensure that there is no problems running the same test we have done, we built a docker image that includes all the tools mentioned above. This docker is included in the source code and the details on how the tests can be run are explained in Section 5.

3 Module description

In the previous reports, we shown the structure of a single PE (Processing Element), how they are interconnected using the drain channel and the idea of using feed and drain arrays in order to abstract to the rest of components the pace in which the systolic array expects the data.

For this deliverable, we have implemented 4 new modules that interacts with the systolic array:

- **Dual Port RAM:** A very simple and ideal memory unit. It has 2 read ports and 1 write port. The access granularity is equal to one row of the systolic array.
- **Read Data Handler:** This unit handles the read accesses of the current operation. From the starting address, this unit will compute the address of the future accesses and passes the data to the systolic.
- **Write Data Handler:** This unit handles the write accesses. It gets the data from the output ports of the systolic array and writes them in to the memory at the correct address.
- **Control:** This unit decodes the instructions and generates the signals that controls the rest of the units (e.g. valid signal). The hazards (e.g. collisions in the drain channel) are also controlled in this unit.

The system does not check for data dependencies amongst instructions, as the hardware for that would be pretty huge and take lot of timing, we would need to check all the stages of the control pipes (around 2 times the size of the systolic), potentially becoming our critical path and adding a lot of complexity to the control in order to compensate that. However, as this is an external accelerator, the driver is the one who will be in charge of making sure that such dependencies are satisfied.

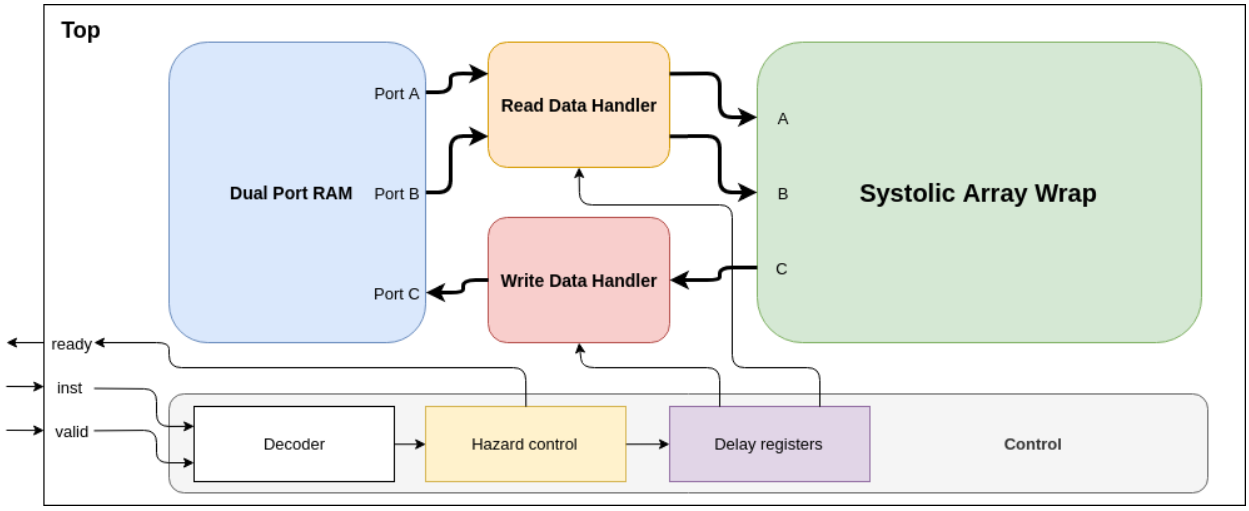

Figure 1: Top level entity block diagram.

3.1 Top module description

Our top module connects the 4 modules described above plus the systolic array presented in the report of the 3rd lab. This top entity has 5 ports:

- **clk_i (input):** This is the clock port.
- **rst_i (input):** This is the reset port. The port is active high, meaning that a value 1 resets the unit and a 0 does not.
- **inst_i (input):** This signal contains the operation code, the two source addresses and the output address where the result will be written.
- **inst_valid_i (input):** This signals specifies whether if the `inst_i` data is valid or not.
- **inst_ready_o (output):** This signal is used to synchronize with other units. If set to 1, the systolic array is ready to execute more instructions, if set to 0, it means that the unit is busy with the previous instruction and cannot start a new computation.

Figure 1 shows the block diagram of the top level entity and how the modules interact with each other. The clock and reset signals are not shown, but they are common signals for all the blocks. As part of the control module, we used delay registers that will control in which cycle the read or the write handler should start doing work.

4 Integration description

As we are designing a standalone accelerator, we did not have any integration with a core, however, we still had to design a testbench in order to be able to test the top level of our system.
As specified in section 3.1, our top module contains the systolic array, the control module, and an interface into a RAM memory. As it is a commanded accelerator, an external stimulus is required. In this case, an external module must feed the unit with commands in order to run a test. Also, as there is no DMA (Direct Memory Access) yet, we assumed that RAM memory is the system's main memory, and thus, we have to pre-load it.
Considering this requirements, we built an external instruction queue, inspired by UVM's queue for SystemVerilog [3], which is in charge of reading a source file, which we named `code.s` and issuing them to the accelerator. This file contains a sequence of parametrized instructions and the structure of the file can be seen in Figure 2. As this units must be synchronized with the top module, the internal instruction buffer uses a simple handshaking method to ensure the correct functioning.

Valid	OpCode	Destination	Source1	Source2
0	nop	*	*	*
1	mmul_d	@dest	@src1	@src2
1	mmul_nd	@dest	@src1	@src2

Figure 2: Behaviour of the instructions for each situation.

In order to simulate that the RAM is the main memory of the chip, we had to pre-load it with the input data. In this case, the data comes from an external file named `random.list` and it contains a sequence of bytes and for pre-loading it, we used the `$readmemh` builtin function available in Verilog.
As soon as the testbench starts the contents of the RAM are loaded and the instruction queue starts pushing instructions into the top level, the tests can end either when the instruction queue runs out of instructions (plus some delay to let the systolic finish its computation), or when it timeouts. In Figure 3 we can see the structure of the testbench, including input files that are going to be loaded into the design.

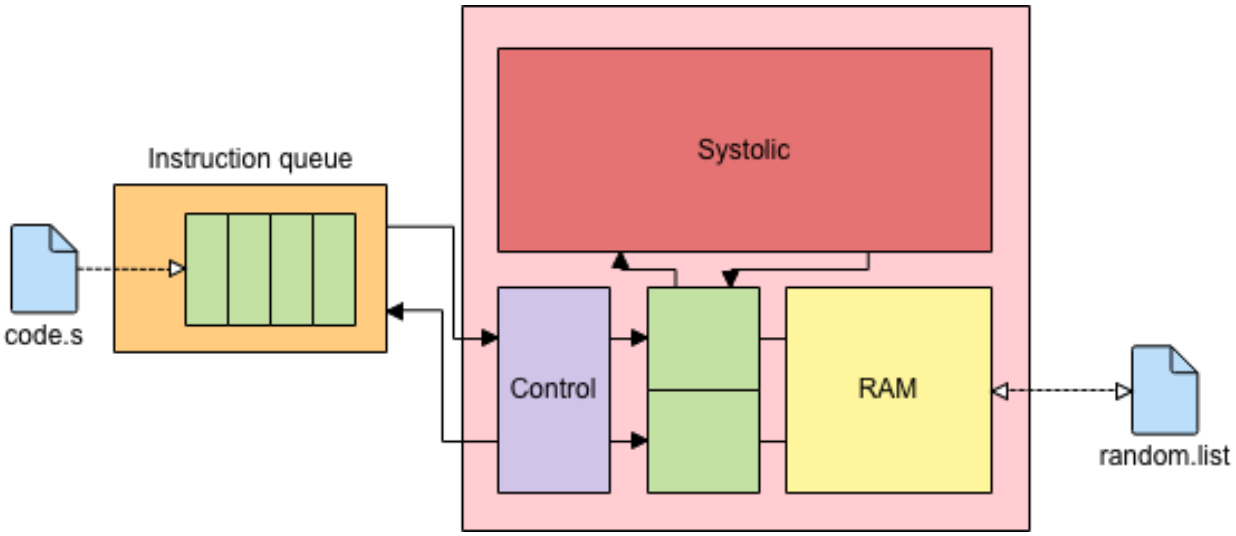

Figure 3: Structure of the top level testbench.

5 Tests and Results

Testing is a really important task in the development world because developers are known to not be bug-free, and thus in this section we will list and explain all the performed tests, as well as explaining some of the planned future tests and tools for the extension.

All tests were run using the `int8` data type and a systolic size of 4x4. All the tests output values have been checked by hand in order to ensure its correctness. We will not enter into the evaluation of the results, as this was already analyzed on the last report, we will rather focus on functionality and control.

5.1 Reproducibility of the tests

Given the basename of the test, all the tests can be reproduced by running the following commands at the repository's root folder:

```
cd $REPOROOT
source env.sh
compile --top top_tb --trace --test TEST_NAME
run top_tb --timeout 10000 --trace test.fst
```

This will produce the trace file `$REPOROOT/last_run/test.fst`, which can be opened using GTKWave. The signals used for this section can be visualized by loading the file `$REPOROOT/tb/top_tb/profiles/memory_interfaces.gtkw` inside GTKWave.

5.2 Selected signals for the traces

For the test traces we have selected a subset of signals that we think are relevant to understand the behaviour of the system. In this subsection we will try to introduce them, as well as to explain their functionality. First, we will see that the traces have the signals split in several sections. We try to take look at the control module, as well as to the buses going into and out of the memory.

- `clk_i`: This signal is the clock of the system, given as a reference for cycle period.
- Control: This section tries to give an overview of what is happening inside the control of the accelerator.
  - `inst_valid_i`: Signal specifying whether the instruction at the input of the control is valid or not, it is used for the handshaking with the instruction queue.

- `dest[9:0]`: Destination's address in memory for the instruction that the instruction queue is sending us this cycle.
- `op`: Operation command of the current instruction. It is a 0 for a `mmul-d` instruction and one for a `mmul-nd` instruction, it's size will grow as more instructions are added.
- `src1[9:0]`: Memory address of the first source for the current instruction.
- `src2[9:0]`: Memory address of the second source for the current instruction.
- `inst_ready_o`: Signal that the control module sends to the instruction queue in order to signal that he is ready to receive an instruction. It becomes 1 after an instruction is issued until a new one is fed into the control.
- `stall_decode`: This signal is driven by the hazard control unit after decoding an instruction. It controls whether the instruction is issued or if the decoder stalls. The ready signal is directly related to it.

- Memory read port A: This block shows the activity of the first read port of the RAM. Its signals conform the first data bus between the memory and the read handler.
  - `en`: Enable read signal, it is set to one when a request is issued.
  - `addr[9:0]`: Address of the request whenever it is issued.
  - `row[31:0]`: Data row returned after the request. It grows dynamically depending on the systolic array's size, for this case it is $8bits \times 4elements = 32bits$.
- Memory read port B: This block shows the activity of the second read port of the RAM. Its signals conform the second data bus between the memory and the read handler.
  - `en`: Enable read signal, it is set to one when a request is issued.
  - `addr[9:0]`: Address of the request whenever it is issued.
  - `row[31:0]`: Data row returned after the request. It grows dynamically depending on the systolic array's size, for this case it is $8bits \times 4elements = 32bits$.
- Memory write port C: This block shows the activity of the single write port of the RAM. Its signals conform the data bus between the memory and the write handler.
  - `en`: Enable write signal, it is set to one when a request is issued.
  - `addr[9:0]`: Address of the request whenever it is issued.
  - `row[31:0]`: Data of the issued write request. Contrarily to the read channels this flows from the write handler to the memory. It grows dynamically depending on the systolic array's size, for this case it is $8bits \times 4elements = 32bits$.

5.3 Base operation tests

5.3.1 Target and code

This test consists on a simple matrix multiplication in order to check if the main unit of the system is working correctly. This includes checking if the data enters and leaves the array as expected and if the control interacts correctly with the instruction buffer.
The test's name is `base`, and its code is the following:

```
1  0 nop       0   0   0
2  1 mmul_d    40  0   10
3  0 nop       0   0   0
```

Which means it will compute a matrix multiplication between the 4x4 matrices stored starting at addresses `0x0` and `0x10`. The contents of the memory are the following ones:

$$M0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 0 \end{bmatrix} M1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 0 \end{bmatrix}$$

5.3.2 Trace analysis

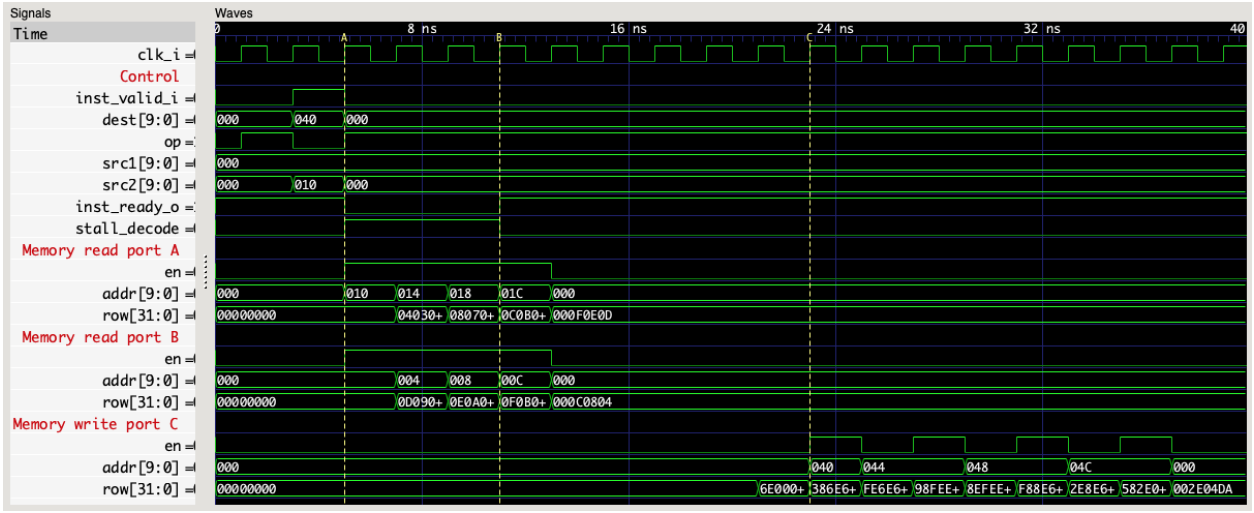

Figure 4: Trace of the execution of the base test.

At Figure 4 we can see the traces of the execution of this test. At marker `A`, we can see the instant at which the control issues the `mmul_d` instruction and how the control stalls for several cycles in order to give some time to the read-handler to load the data until marker `B`. Looking at the read ports of the memory we can also see who the data is diligently started to fill into the array one cycle after marker `A`. And looking at the write port, we can see how the data is successfully written at marker `B` after all the computation.
With this, it looks like the control timings are set up appropriately in order to operate the array.

5.4 Hazard test

5.4.1 Target and code

This test consists on a small sequence of instructions, a couple without draining, a couple with drain, in order to see how the control takes care of the control takes care of the timing constraints of the system. The code tries to check all the possible combinations of instruction after instruction.
This test's name is `hazard`, and its code is the follwing:

```
1  0 nop        0   0   0
2  1 mmul_nd    40  0   10
3  1 mmul_nd    40  0   10
4  1 mmul_d     40  0   10
5  1 mmul_d     40  0   10
6  1 mmul_nd    40  0   10
7  0 nop        0   0   0
```

With this, we check the following sequence cases:

- Non-drain after Non-drain

- Drain after Non-drain
- Drain after Drain
- Non-drain after Drain

The contents of the memory are the same of the last test, as the target of this test is not computation, but functionality.

5.4.2 Trace analysis

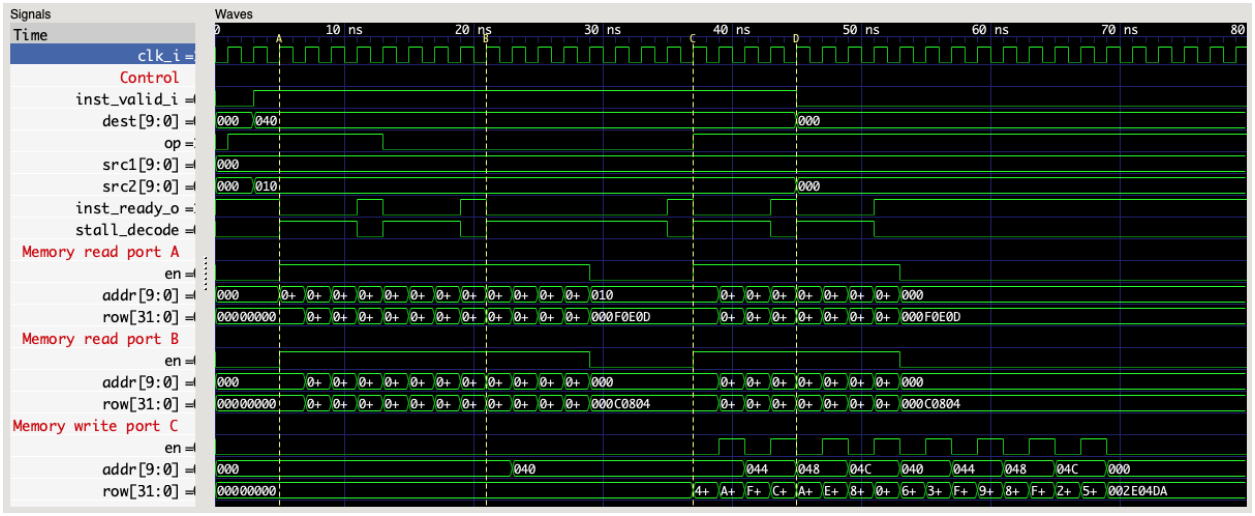

Figure 5: Trace of the execution of the hazard trace.

At Figure 5 we can see the trace of the execution of the test.
In this test we can see how the several instructions interact nicely. First, between marker `A` and marker `B`, we can see how the first three instructions are issued. As the only one using the drain channel is the last one, the only bandwidth limitation they have amongst them is that of the feed channels, and thus, we can see how they are issued with an interval equal to the size of the array, using the feed channels at their full capacity. Then, between marker `B` and marker `C` we can see the interaction amongst the two drained instructions. As they both use the drain channel, which has only half the bandwidth of the feeding ones, they have to keep themselves separated by two times the size of the array, which results on the drain channel being utilized at full capacity (one line every two cycles) without any collision amongst them, which can be seen at port c right after marker `C`.
And lastly, at marker `D` we can see how the last non-drained instruction is issued. As this instruction does not use the drain channel, its only limitation with the previous instruction is the feed channels' bandwidth, thus it only has to wait the same as if the previous one was a non-drained instruction and, as we can see, results on the feed channels being used at their fullest.
After looking at this behaviour, we can assert that the hazard timings set up between instructions allow us to utilize the system at its maximum possible bandwidth at any given time, without any collisions or inter-instruction interference.

5.5 Long computational base test

5.5.1 Target and code

This test consists on a large sequence of drained operations, all consecutive, multiplying again and again three different matrices against the identity matrix, to fully ensure that there is no interference amongst independent computations and that we are able to consitently mantain the system at maximum bandwidth. This test's name is `identity`, and its code is the following:

```
0 nop       0   0   0
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
1 mmul_d    10  0   10
1 mmul_d    20  0   20
1 mmul_d    30  0   30
0 nop       0   0   0
```

The identity matrix has ben placed in memory at address `0x0`, while the other matrices are at adresses `0x10`, `0x20`, and `0x30`.

$$M0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M1 = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{bmatrix} M2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} M3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.5.2 Trace analysis

At Figure 6 we can see the resulting trace of this test.
This test allows us to observe the regularity at which the instructions are issued, and that the data is fed and retired to memory. So far, the system looks to be limited by the drain channel, as we expected, but the system keeps it constantly completely saturated, meaning that we are able to operate the system at its maximum write throughput.

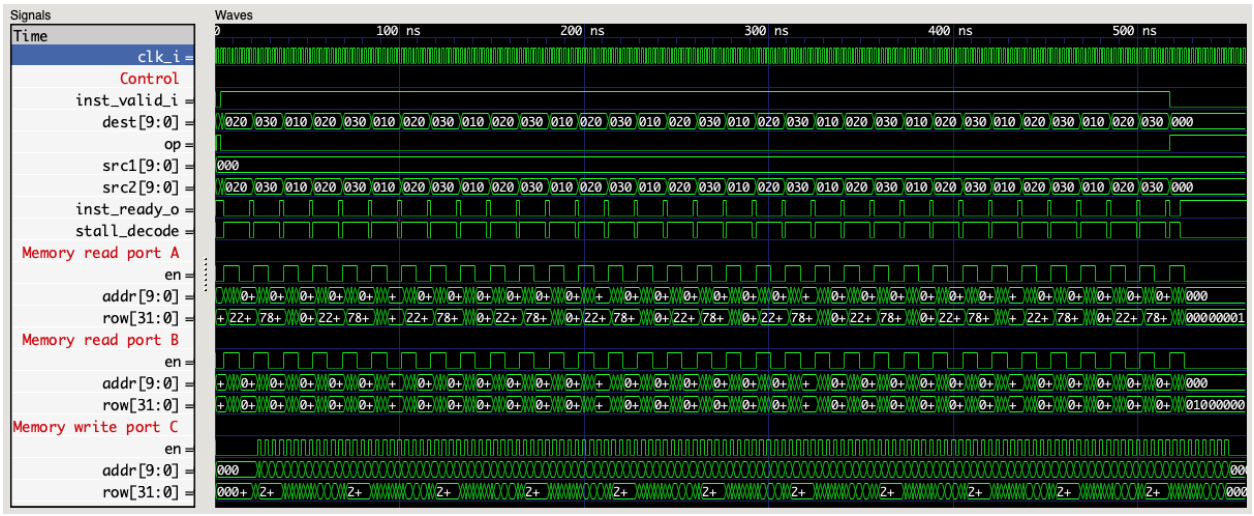

Figure 6: Trace of the execution of the identity test.

5.6 Future tests

As future work, as an extension, it would be great to automate the validation of the results and put more emphasis on real computation. From the validation point there are still several issues that have to be tackled, one of them would be performance analysis of the system. It would be interesting to have a whole analysis of how different elements of the design would operate given different problem sizes or structures. From the functionality standpoint we are still lacking the validation of operations with bigger matrices using blocking or with a bigger systolic array. This has not been tested yet due to the lack of automation on test generation and evaluation.

6 Conclusions

This project covers all the challenges that one typical faces when starts designing a processor, in our case an accelerator that works in conjunction with a processor. First we learned the basis of a Systolic Array and the reason why they are so good for certain workloads (Lab1 report). This let us understand the problem we were trying to solve (acceleration of the Matrix Matrix Multiplication operation) and to explore different existing design strategies. The first bottleneck we detected during the design stage is that PEs finishes the computation at different rates.

In order to tackle this issue, we designed a drain channel that is pipelined and because area is also important, we managed to make the drain channels to be shared between two consecutive PEs columns. As the drain channels were shared we also had to design a drain array that collects the data at the correct pace.

From the design to the implementation stage, as we carefully described the behaviour of each module during the design phase, we did not face too many problems during the implementation. However, we failed in foreseeing the complexity behind some tasks, which forced us to not fully implement all the ideas that we initially proposed.

Finally, from this project we learned the advantages behind a Systolic Array for certain workloads and how difficult is to design and implement one from scratch. Luckily we spent enough time during the design phase, and the implementation stage went out smoothly. We believe that doing the exercise of designing and implementing a processor module (in this case an accelerator) is really interesting as it forces you to understand and experience from first hand the importance of defining a good specification during the design phase.

References

[1] H.T. Kung and C.E. Leiserson. ‘‘Systolic Arrays for VLSI’’. In: *Sparse Matrix Proceedings* (1978).

[2] Sayeh Sharify et al. ‘‘Laconic Deep Learning Computing’’. In: (2018). URL: https://arxiv.org/pdf/1805.04513.pdf.

[3] Janick. Bergeron. *Verification methodology manual for SystemVerilog*. Springer, 2006, p. 503. ISBN: 9780387255569. URL: https://books.google.es/books?hl=ca&lr=&id=-bNJAAAAQBAJ&oi=fnd&pg=PA1&dq=unified+verification+methodology&ots=DDLM-ryROo&sig=jtny2dfzd4l89I0O0Z6VqiA-ZLI#v=onepage&q=unified%20verification%20methodology&f=false.