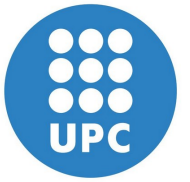


UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Processor Design
(PD)

Module Definition

2nd Report

Jordi Solà, Ying hao Xu

October 16, 2018

Contents

1 Description of the module

As mentioned in the previous deliverable, we are going to design an accelerator, specifically a systolic array. The target of the systolic array we are going to built is to compute a matrix matrix multiplication as fast and as efficient as possible.

The communication between the accelerator and the processor will be based on a custom non-standard protocol, therefore the accelerator can operate as a standalone device connected through a peripheral bus (e.g. PCI-Express) if it is needed.

The accelerator itself will have its own "cache" memory connected to a DMA (Direct Memory Access). The systolic array will only be fed from this cache memory. To fill the internal cache memory, the DMA will be used and it will be capable of accessing directly to the last level cache.

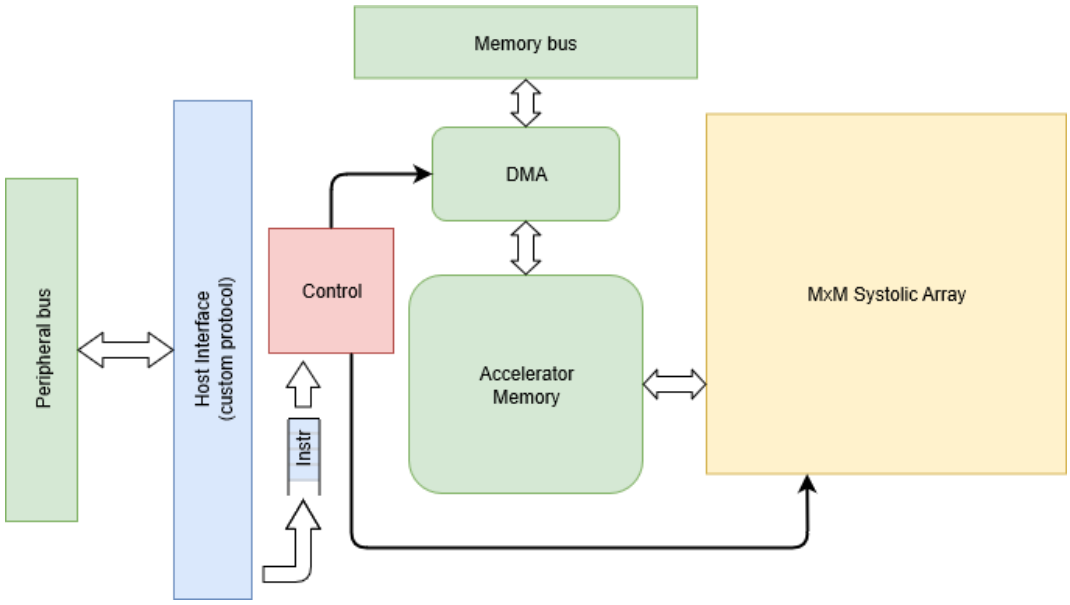


Figure 1: Accelerator block diagram

The systolic array will be an array of simple PEs (Processing Elements) that are capable of multiply and add 8 bit integers on a single cycle. Figure 2 shows the structure of a single PE. The way how these PEs work is to multiply and accumulate the result inside the PE and only pass the original matrix elements instead of passing the result too to other units. The benefit of this system is the simplification of the control system as the data that comes out of the systolic array is final (does not have to be accumulated with other results).

To output the result, we have designed a drain channel that is pipelined and shared between two consecutive columns of the array (Figure 3). The reason why we can share the drain bus between two consecutive columns of the systolic array is due the nature of how the data is computed. In the systolic array we propose, the data enters to the system in cascade and as every PE does the same number of operations, the final results are also available in cascade order. This property lets us take advantage of the drain channel by sharing it between two columns and save area as there will never be a collision between them.

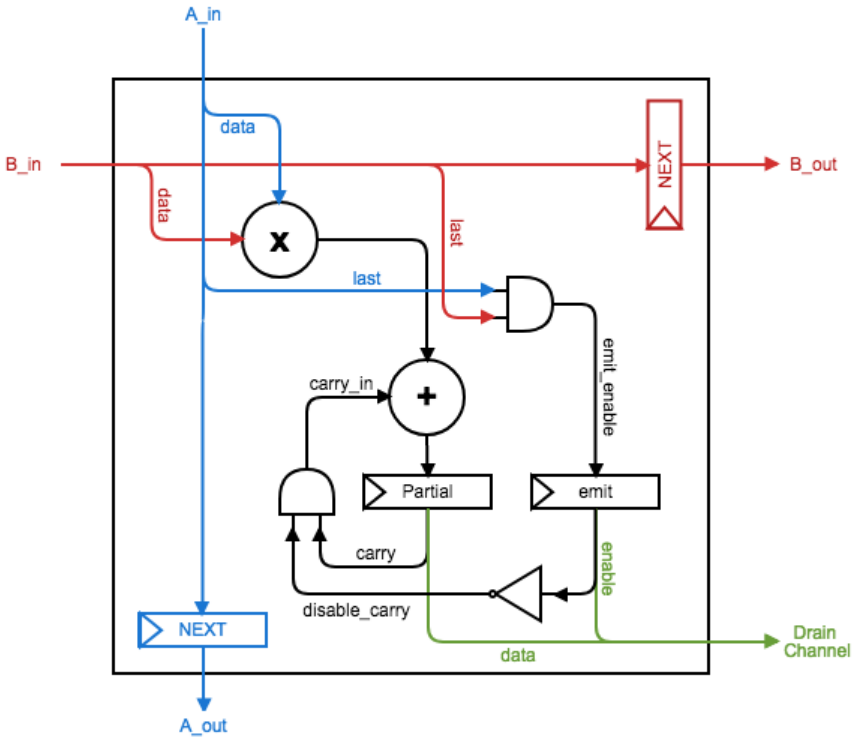


Figure 2: Processing Element structure

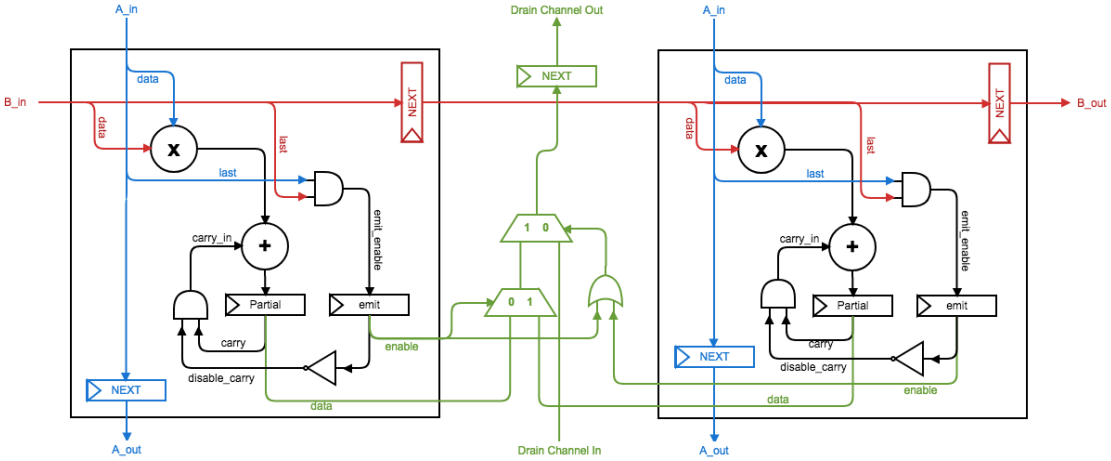


Figure 3: Two consecutive Processing Elements connected to a single drain channel

1.1 Processing Element Input and Output ports

- **Inputs:**
  - **a_data_i:** 8 bits element of the matrix A.
  - **b_data_i:** 8 bits element of the matrix B.
  - **a_last_i:** single bit to specify that it is the last element of the matrix A and therefore in the next cycle the result can be written into the drain channel.
  - **b_last_i:** single bit to specify that it is the last element of the matrix B and therefore in the next cycle the result can be written into the drain channel.

- **Outputs:**
  - **a_data_o:** 8 bits element of the matrix A. This is the A element that the PE received in the previous cycle and the idea is to pass it to the PE of the next row on the same column.
  - **b_data_o:** 8 bits element of the matrix B. This is the B element that the PE received in the previous cycle and the idea is to pass it to the PE of the same row but on the next column.
  - **a_last_o:** single bit that indicates if the *a_data_o* element is the last one of the matrix A.
  - **b_last_o:** single bit that indicates if the *b_data_o* element is the last one of the matrix B.
  - **drain_data_o:** 8 bits element which is a final element of the results matrix.
  - **drain_en_o:** single bit to control the drain channel. If set to one, the drain channel will be fed with the *drain_data_o* instead of the element coming from the previous row.

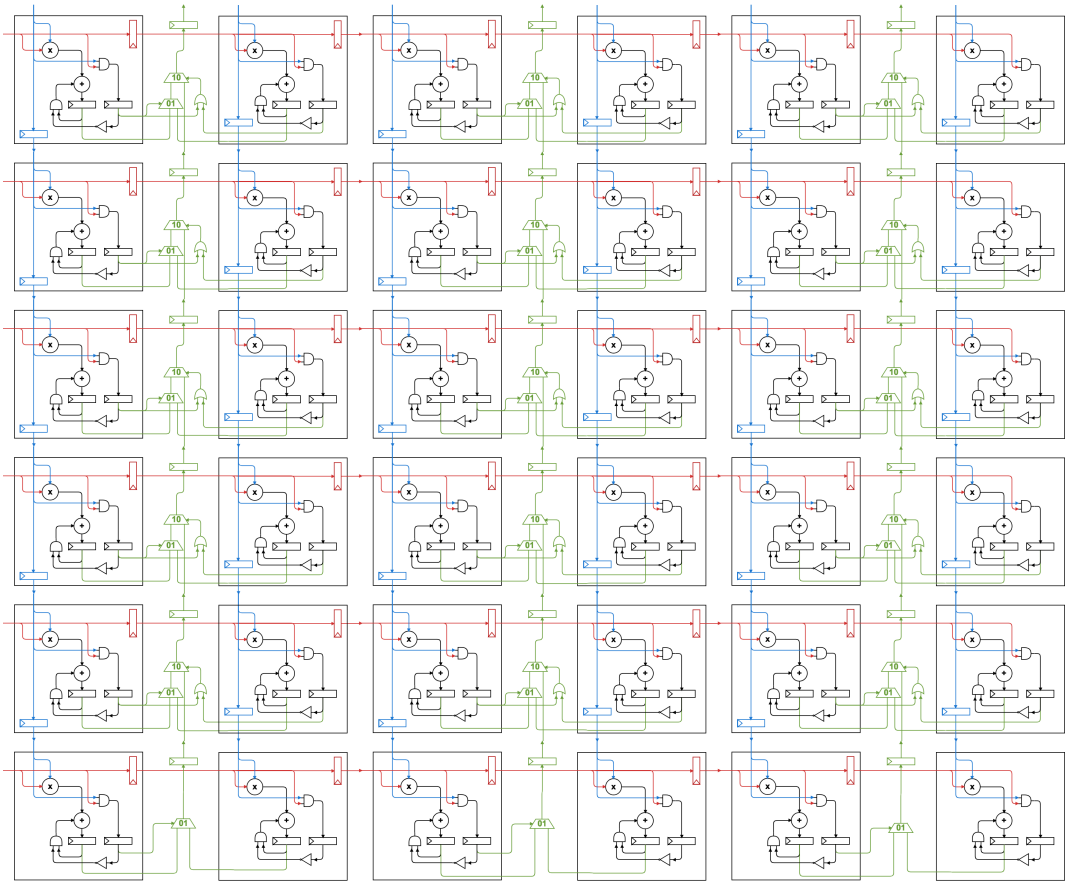


Figure 4: Schematic of a matrix of 6x6 PEs interconnected

1.2 Entire Systolic Array Input and Output ports

- **Inputs:**
  - **a_data_i[N]:** array of 8 bits ports of length N. Each element of the array is connected to a row of the matrix A stored in the internal memory, and every cycle each port is fed with the element of the same row next column.
  - **b_data_i[N]:** array of 8 bits ports of length N. Each element of the array is connected to a column of the matrix B stored in the internal memory, and every cycle each port is fed with the element of the next row same column.

- **a_last_i[N]:** array of single bit ports of length N. Each element of the array specifies if the element in *a_data_i* is the last element of the row in the matrix A.
- **b_last_i[N]:** array of single bit ports of length N. Each element of the array specifies if the element in *b_data_i* is the last element of the column in the matrix B.

- **Outputs:**
  - **drain_data_o[N]:** array of 8 bits ports of length N. Each element of the array is a final element, part of the results matrix.

2 Accelerator ISA

The control of the accelerator is splitted into two parts, first, the control of the DMA in order to manage the data flow, and second, the control of the systolic array in order to do the correct operations.

Regarding the style of the ISA, we wanted to minimize the complexity of the instructions and the operations, so we went for a RISC-like approach.

2.1 Memory model

As the systolic array is designed to compute matrices of a fixed size $N \times N$, the DMA assumes that the data is stored in memory following a blocked approach. This is, consecutive memory will contain only elements of the same block of dimension $N \times N$. Important to clarify that these blocks refer to the blocks used in a blocking algorithm for computing matrix matrix multiplications. This decision was taken in order to simplify the operations and reduce the required variables to operate the array.

Even though this might introduce a limitation, in the end this mapping has a huge positive impact on the performance, as all the data will be stored contiguously, we will take full advantage of the data locality during the fetch of the data. On the other hand, this also let the outputs to be fed-back to the array without any transformation or overhead. In Section 3.2 we will enter in further detail on how to compute matrix multiplications of bigger sizes than the systolic array in an efficient way.

2.2 Instructions

In this section we are introducing the instructions of the accelerator, however, the precise encoding of them will be decided and parameterized later on during the implementation, based on the results of execution.

Given that the accelerator is memory mapped on the global memory, only one instruction is needed to control the DMA:

```
blkcpy source_addr, target_addr
```

Which will copy a matrix block from `source_addr` to `target_addr`, and as the accelerator memory is global, this operation can go both ways, copying data to the accelerator, or sending results back to main memory, depending on the location of both addresses.

As for the computation control, two instructions will be implemented:

```
mmul-d source_block_c, source_block_a, source_block_b
```

Which will take the matrix blocks at addresses `source_block_a` and `source_block_b`, compute their multiplication, accumulate the result with the values currently stored in the array, drain the systolic array, and store back the results at `source_block_c` address.

```
mmul-nd source_block_a, source_block_b
```

Which will take the matrix blocks at addresses `source_block_a` and `source_block_b`, compute their multiplication, and accumulate the result inside the PEs with the values currently stored in the units.

3 Array usage

Due to the fact that our systolic array has a fixed number of PEs, depending on the size of the matrices, there are three different situations that must be considered:

- **Same size:** In the case where the matrices has the same size as the systolic array, only one computation needs to be scheduled and the final result is obtained straightforward.
- **Bigger:** In the case where the matrices to be operated are bigger than the systolic array, we have to apply blocking and feed the systolic array with different chunks in a specific order.
- **Smaller:** In the case where the matrices are smaller than the systolic array, several optimization on power and latency can be performed, turning off the non-used cells.

For the first two situations, we will provide examples of the control and the behaviour of our proposed systolic array, while the third will be treated as an optimization for the accelerator, and thus will be explained at Section 6. For both experiments, we will use an array of size $2x2$ (Figure 5) for illustrative purposes.

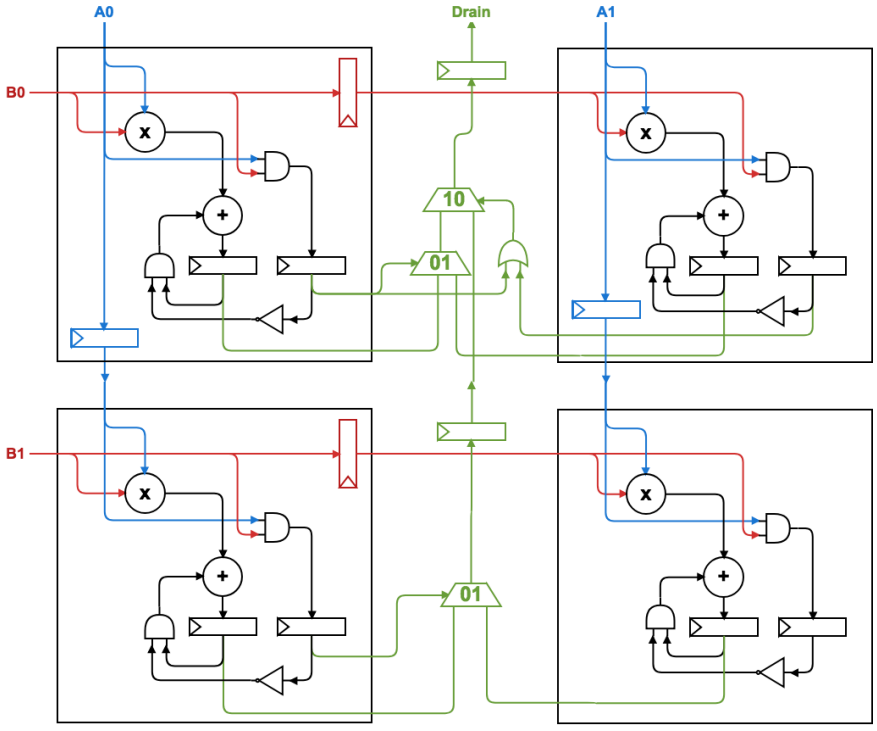


Figure 5: Schematic of a matrix of 6x6 PEs interconnected

3.1 Matrix of the same size

In order to explain what happens when we are operating on matrices of the same size, let us consider the following square matrices of size 2:

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} C = \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix}$$

And the following operation:

$$C = A \times B$$

For the case of 2x2 systolic array (Figure 5), the array would be fed as shown in Figure 6, which would be issued using the following sequence of instructions:

```
# Fetch the input data
blkcpy   @main_mem_A, @acc_A
blkcpy   @main_mem_B, @acc_B
# Compute the matrix
mmul-d   @acc_C,      @acc_A,        @acc_B
# Send the result to main memory
blkcpy   @acc_C,      @main_mem_C
```

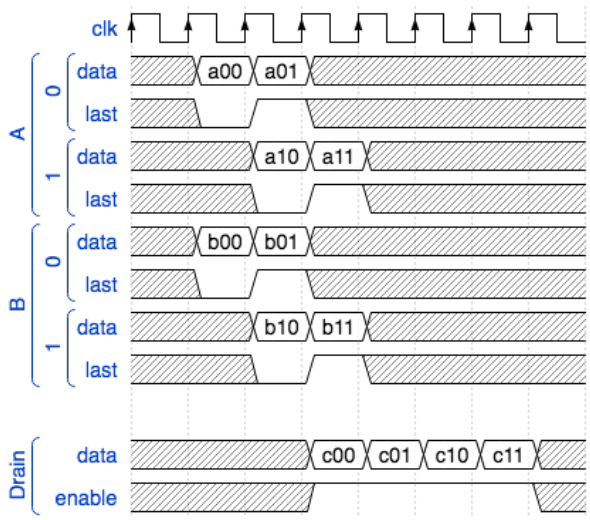


Figure 6: Waveform of the inputs and outputs of the systolic while performing single block matrix multiplication.

3.2 Bigger matrix with blocking

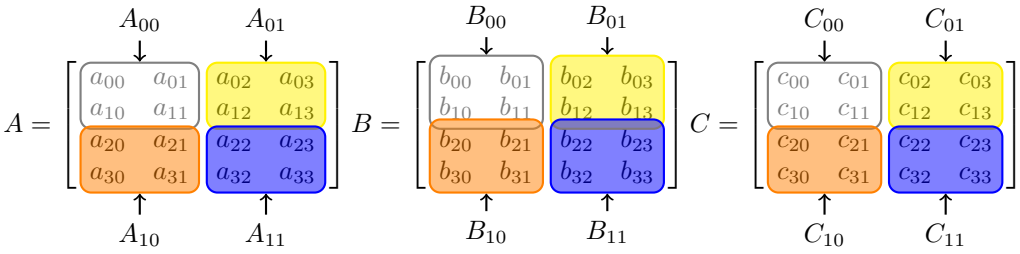
In order to explain what happens when we are operating on matrices with a size greater than the systolic array, let us consider the following square matrices of size 4:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} B = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} C = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

And the following operation:

$$C = A \times B$$

Considering that our systolic array has a size of 2, we will have to apply a blocking algorithm such that the block granularity matches the dimension of our systolic array.



The blocking algorithm to compute the matrix matrix multiplication is defined as $C_{i,j} = \sum_{k=0}^{n/N} A_{i,k} \times B_{k,j}$, where i and j are block indices of the matrices, n the size of the original matrix and N the size of the systolic array. This strategy allows us to schedule all the component products of a block sequentially, with the advantage of re-using the intermediate results that are stored in the PEs without having to do a drain. Figure 7 shows a waveform of how the blocks are scheduled to compute the blocked operations and how the results are outputted. This computation can be issued using the following operations:

```
1   #Fetch initial data
2   blkcpy   @main_mem_A00, @acc_A00
3   blkcpy   @main_mem_B00, @acc_B00
4   #Pre-fetch next iteration data
5   blkcpy   @main_mem_A01, @acc_A01
6   blkcpy   @main_mem_B10, @acc_B10
7   #Compute first product of C00
8   mmul-nd  @acc_A00,      @acc_B00
9   #Pre-fetch next iteration data
10  blkcpy   @main_mem_B01, @acc_B01
11  blkcpy   @main_mem_B11, @acc_B11
12  #Compute final result for C00
13  mmul-d   @acc_C00,      @acc_A01,      @acc_B10
14  #Pre-fetch next iteration data
15  blkcpy   @main_mem_A10, @acc_A10
16  blkcpy   @main_mem_A11, @acc_A11
17  #Compute first product of C01
18  mmul-nd  @acc_A00,      @acc_B01
19  #Send C00 to main memory
20  blkcpy   @acc_C00,      @main_mem_C00
21  #Compute final result for C01
22  mmul-d   @acc_C01,      @acc_A01,      @acc_B11
23  #Send C01 to main memory
24  blkcpy   @acc_C01,      @main_mem_C01
25  #Compute first product of C10
26  mmul-nd  @acc_A10,      @acc_B00
27  #Compute final result for C10
28  mmul-d   @acc_C10,      @acc_A11,      @acc_B01
29  #Send C01 to main memory
30  blkcpy   @acc_C10,      @main_mem_C10
31  #Compute first product of C11
32  mmul-nd  @acc_A10,      @acc_B01
33  #Compute final result for C11
34  mmul-d   @acc_C11,      @acc_A11,      @acc_B11
35  #Send C11 to main memory
36  blkcpy   @acc_C11,      @main_mem_C11
```

We can see that with this specific case the utilization of the array is really high, as we can interleave the operations that are executing on the array. Also, the fact the matrix multiplications take several cycles, gives us a window in which, if utilized, we can perform pre-fetches of future needed data, effectively hiding the memory accesses latency.

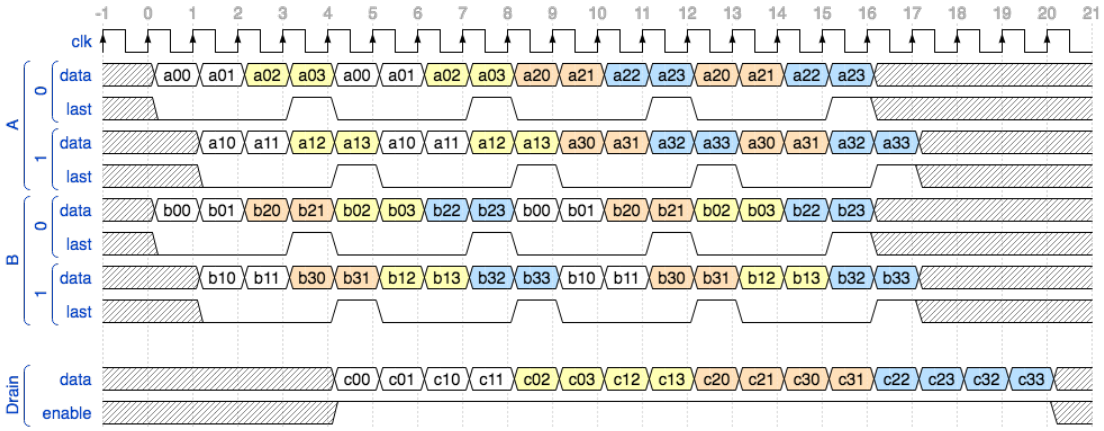


Figure 7: Waveform showing the computations of a 4x4 matrix on a 2x2 Systolic Array.

4 Design limitations and bottlenecks

Due the cascade nature of feeding the system and because each PE does the same number of operations, the upper PEs will finish sooner than other PEs. In order to exploit the utilization of the system as much as possible, we would like the system to start a new computation as soon as the last instruction has been issued. However, there exists an imbalance between the input and output channels. Due to the drain channel being shared among two columns, it has half of the wires, thus causing the drain channel to have half the maximum bandwidth compared to the input channels.

In fact the critical path for the drain channel is what limits the latency between the drains (see Figure 8) of two independent operations per each pair of columns (and thus for the whole systolic). Because the systolic is fed in cascade, the results will be put in the drain channel also in cascade, therefore this latency (T_d) can be defined as two times (because both columns writes into the channel) the number of rows of the systolic array.

$$T_d = 2N$$

Another concept to consider is the computation throughput (t_c) which can be seen as how many operations a PE does before putting the final result on the drain channel. In this case, as each PE computes an element of the results matrix, the computation throughput is the number of columns of the matrix A or the number of rows of the matrix B.

Considering this two concepts, the assumption of being able to overlap different computations remains true if the following condition is fulfilled $t_c \geq T_d$, as we will put data on the drain channel in a lower pace than the capacity of the channel to drain the data.

Figure 8 shows a waveform when we try to compute two 2x2 matrix matrix multiplication on a 2x2 systolic array. In this case, between *a00* enters an *c00* flows out, there are 2 cycles (t_c). If we try to start a new computation when *c00* is done, after 2 cycles, we will have a new result which will collide with *c10*, therefore we have to wait until the computation is done in order to start a new computation (yellow).

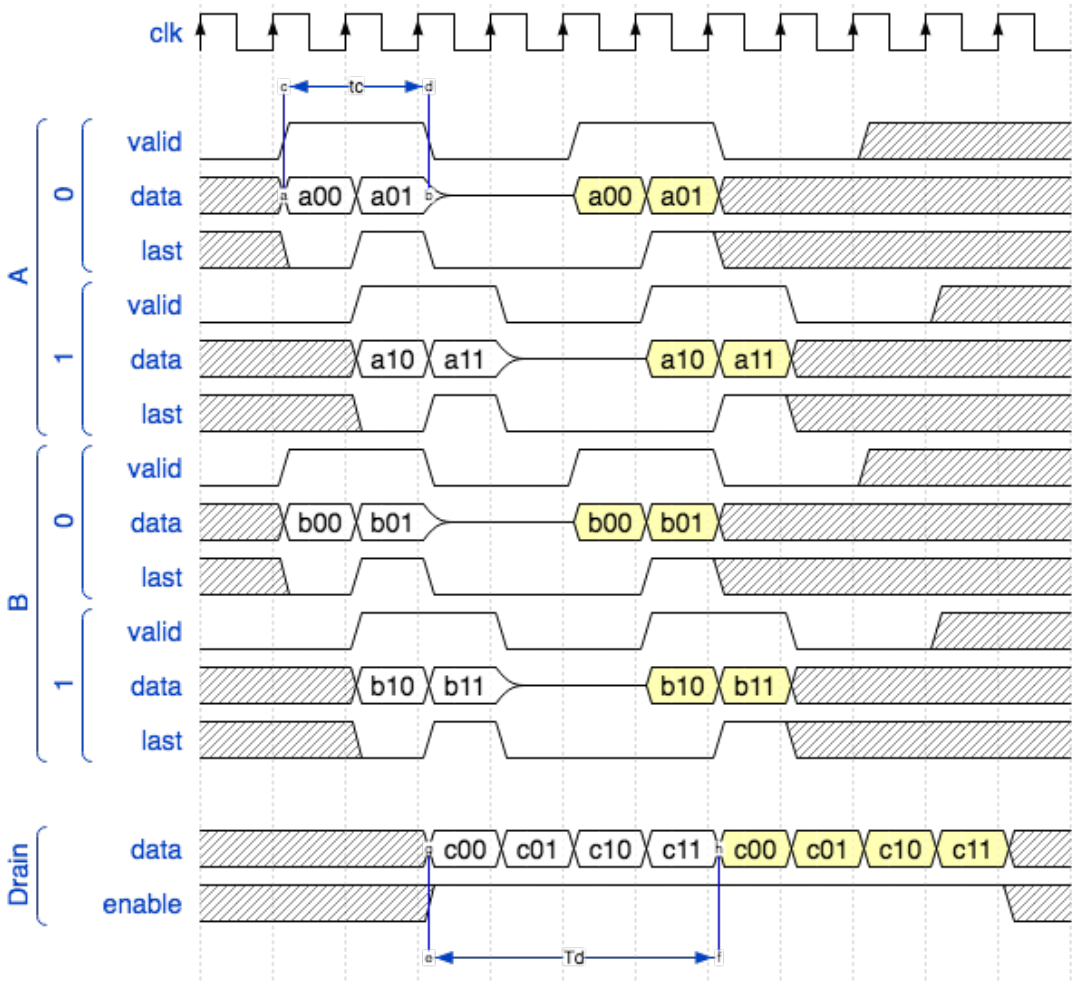
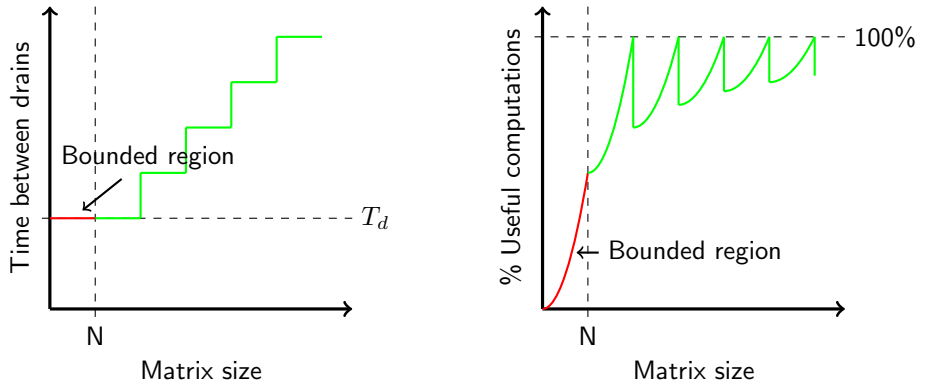


Figure 8: Waveform showing two computations of a 2x2 matrix on a 2x2 Systolic Array.

In Figure 7 from the previous section, we showed the case when a 4x4 matrix is computed on a 2x2 systolic array. In this case, because we compute the matrix in blocks, the intermediate results are stored inside each PE and only when the last bit is set, the result is put in the drain channel. Given that $t_c = 4$, we can successfully overlap multiple computations with no-collisions on the drain channel as $T_d = 4$ and the $t_c \geq T_d$ condition is fulfilled.

Summarizing the two cases analyzed above, we can conclude that the drain time (T_d) remains constant, which has a considerable negative impact on the performance. As explained before, this is because the drain channel has a fixed drain pace while the PE throughput (t_c) varies depending on the matrix size. When the matrix size is small, we are limited by the drain channel drain rate (bounded region on Figure 9a) while for matrices greater or equal than T_d, we are limited by the rate of producing a new final result from the PE.

As mentioned in the Section 3.2, when the matrices are bigger than the systolic array, we have to apply blocking. As the systolic array size is fixed, we have to fill with 0s the empty positions when we have to compute blocks smaller than the systolic array size. Filling the blocks with 0s implies that for that specific block, the useful computation made by the systolic array is low and the t_c will not vary for sizes between K and K+N (being K a multiple of N, and N the size of the systolic) that is the reason why the trend on the Figure 9a is stepped.



(a) Time between drains per different matrix sizes, being N the systolic array size.

(b) Average occupation percentage of all PEs given a matrix size, being N the systolic size.

Figure 9

Figure 9b shows the percentatge of useful computations given a matrix size. For matrices smaller than N (systolic array size), most of the PEs are computing the filled with 0 elements, therefore the percentage of useful computations is really low. Once the matrix size is equal to N, the maximum percentage of useful computation we can achieve is 50%. This is because we are limited by the drain channel output rate (as seen in Figure 6). As we increase the matrix size, the percentage of useful computation increases quadratically, reaching the 100% when the matrices size is a multiple of N. Once past this point, the useful computation drops again whenever the size of the matrices are not multiple of N, however, the percentage drops less and less as we increase the matrix size. This is because the number of blocks filled with 0s increases linearly while the number of blocks only containing data grows quadratically.

5 Verification

For the Verification we are going to use Verilator, a free Verilog HDL simulator that compiles synthesizable Verilog/SystemVerilog code[1]. Each of the modules will be tested independently and the complete design will be tested using a *checker*.

The units we will check are:

- **Processing Element:** to test this unit, we will check that the multiplication and the accumulation is done correctly and that the final result is put on the drain channel in the correct cycle.
- **Drain channel:** to test the drain channel, we will check and ensure that data flows out following the expected pace and that it behaves as expected when the enable bit is set.
- **Memory Unit:** to test the internal memory of the accelerator, we will check the following operations:
  - Load data from the main memory
  - Store data back from accelerator memory to the main memory
  - Provide the data to the systolic array following the correct rhythm.

For the full system simulation, the Verilator simulation will be driven and fed by a *checker*. This *checker* will be capable of generating random N by N matrices that will be passed to the simulation as an input, and detect when the accelerator emits the results of each computation in order to compare it to the expected value. This method let us automatize as much as possible the verification while at the same time it tries to cover all the potential corner cases.

[1]For more information see Verilator's website

In order to detect what is happening in our unit, we will add SystemVerilog DPI (Direct Programming Interface) calls. DPI is basically an interface between SystemVerilog and a high-level language (e.g. C). Using DPI let us build the *checker* in a traditional language, allowing him to control the simulation, injecting instructions, memory data, and performing the same computations as the systolic array but on an alternative code. This allows us to constantly validate any possible result produced by our model.
We are considering to build two different testing setups:

- **Parameterized tests:** We should be able to run simulations with static tests in order to test specific cases or replicate detected bugs.
- **Random feed:** One possible way of making an exhaustive validation of our system is to constantly feed it with random instructions and data while supervising it with our *checker*, this way we will be able to detect potential errors due to non-considered corner cases, and potential non-appreciable errors when performing a single operation propagated into the next ones (e.g. collisions on drain, partial results not completely erased, non used PEs modifying their state when they should not, etc.).

6 Potential Extensions

As part of the extensions, we propose the following upgrades as potential improvements:

- Clock gate the unused flip-flops.
- Dynamically adapt the size of the systolic array.
- Add support for an AXPY operation ($Y = A \times X + Y$).
- Accelerator memory model.
- Advanced scheduling for packed small operations.

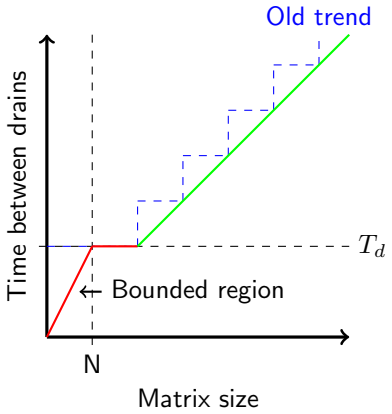
6.1 Clock gating unused flip-flops

The idea behind this extension is to be closer to a real world environment, where energy consumption is really important. Therefore, all the flip-flops will be clock gated giving the possibility to fully turn-off the accelerator when it is not used.

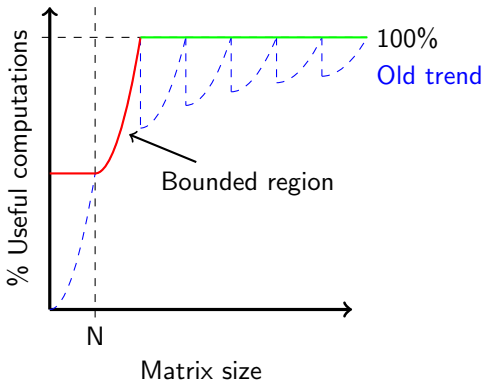
6.2 Dynamically adapt the size of the systolic array

As shown in the Figure 9b, the useful computation drops when the matrices size are not multiple of the systolic array size. This extension pretends to dynamically adapt the size of the systolic array by disabling the non-used PEs. By disabling the PEs, the control will also have to adapt to the new size.

As result, for matrices which size are greater than T_d, the percentage of useful computation will be 100% even if the matrices size are not multiple of the systolic array size (Figure 10b), as for computing the remaining matrix blocks that are smaller than the systolic array size, we will only activate the required PEs. This optimization in fact reduces dynamically the T_d and the t_c, letting other operations to start earlier and therefore improve the throughput as seen in Figure 10a.



(a) Time between drains per different matrix sizes, being N the systolic array size.



(b) Average occupation percentage of all PEs given a matrix size, being N the systolic size.

Figure 10

6.3 Add support for an AXPY operation

Currently the PE of the systolic array only supports multiply and add, considering 0 as the initial value to accumulate on.

In this extension, we are thinking about adding the possibility to use a user defined value to accumulate the operations instead of using 0. This operation is defined as $Y = A \times X + Y$, being Y a value defined by the user. Thus from the point of view of a matrix operation, this will be translated into $C = A \times B + U$, where C is the result, A, B are the original matrices and U the matrix to accumulate the results.

6.4 Accelerator memory model

Even though systolic arrays provide huge computation throughput, the system cannot take full advantage of it if it is unable to feed the systolic array with a comparable data flow. Given that, it makes sense to say that its built-in memory plays an important role on the viability of the system.

This extension is about designing, from a theoretical point of view, the internal structure of this component. It would imply studying the different technologies on the market and their limitations, and the design of different problem-dependant configurations, analyzing and justifying their use-cases.

6.5 Advanced scheduling for packed small operations

It might be the case that for some applications, the matrices or vectors are packed in a way in which it is possible to pack several independent operations together, increasing the throughput of the system. One possible extension would be to define an algorithm in order to pack computations more effectively, defining new instructions if necessary.

As an example, for the case of matrix-vector multiplication, one could pack several column vectors together in the form of a matrix, in order to later retrieve each of the columns of the resultant matrix as the resultant vectors.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_0 & c_0 & d_0 & e_0 \\ b_1 & c_1 & d_1 & e_1 \\ b_2 & c_2 & d_2 & e_2 \\ b_3 & c_3 & d_3 & e_3 \end{bmatrix} = \begin{bmatrix} b'_0 & c'_0 & d'_0 & e'_0 \\ b'_1 & c'_1 & d'_1 & e'_1 \\ b'_2 & c'_2 & d'_2 & e'_2 \\ b'_3 & c'_3 & d'_3 & e'_3 \end{bmatrix}$$