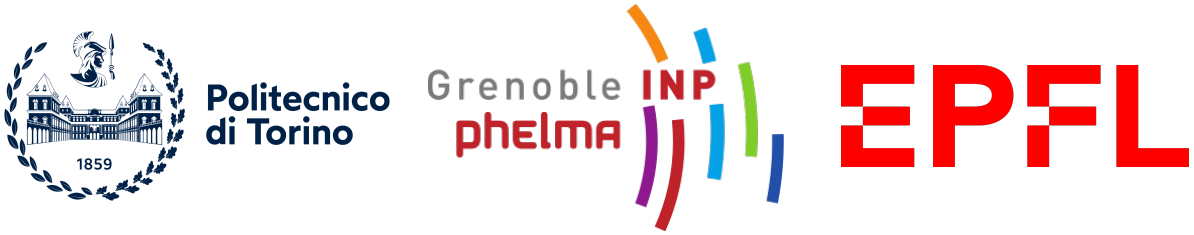

Master thesis report

NANOTECH 17 (2021-2022)

Implementation of a processor platform with IMC addons in 65nm technology

EGGERMANN Grégoire
gregoire.eggermann@grenoble-inp.org

From the 21st of February 2022 to the 12th of August 2022

Station 11 CH-1015 Lausanne

Supervisors:

- Dr. Alexandre LEVISSE, alexandre.levisse@epfl.ch
- Prof. Lorena ANGHEL, lorena.anghel@phelma.grenoble-inp.fr
- Prof. David ATIENZA, david.atienza@epfl.ch

Abstract

With the significant increase of computation throughput required by recent application such as machine learning, data processing or even internet of things, the reduction of calculation time and power consumption is more and more demanding for edge devices. Unfortunately, standard memory technologies can not stand the computing performances expected. In this context, new architectures are explored to face this issue. In-memory computing is one of the potential architecture that could enable to improve computing efficiency by reducing the data movement between the memory and computing cores. Convolution neural network is a potential application that could be run on this kind of memory. In the context of this study, a very efficient way to compute a single convolution layer have been explored starting from a first design of BLADE, an in-memory architecture developed in ESL. The design of the controller of this memory have then been improved in order to facilitate the communication between the central process unit and the memory. These new design optimizations have finally been tested with convolution application and have shown a 75% reduction of time to do convolutions compared to a RISC-V core processor.

Résumé

Alors que les applications actuelles telles que l'apprentissage artificiel, le traitement de l'information et l'internet des objets nécessitent un débit de calcul de plus en plus important. La réduction de la consommation d'énergie et du temps de calcul représentent des enjeux crutiaux pour les "edge devices". Malheureusement, les mémoires standards ne peuvent pas atteindre les performances de calcul espérées. Dans ce contexte, de nouvelles architectures sont envisagées pour résoudre ces problèmes. L'"in-memory computing" est l'une des architectures potentielles qui pourrait permettre d'améliorer la puissance de calcul en réduisant le déplacement de l'information entre la mémoire et les bloc arithmétiques. Les réseaux de neurones à convolution sont une application qui pourrait gagner à fonctionner sur ce type de mémoire. Dans le context de cette thèse de master, une manière efficace de calculer un noyau de convolution sur l'ancien design de BLADE a été étudiée. Le design du controlleur de BLADE a ensuite été amélioré afin d'accroître les performances de communication entre le processeur et la mémoire. Ces nouvelles optimisations ont enfin été testées sur des convolutions, démontrant 75% de réduction sur le temps de convolution comparé à un processeur RISC-V.

Sommario

Con il significativo aumento della potenza di calcolo richiesta dai recenti sviluppi del machine learning, dell'elaborazione dei dati o dell'Internet of Things, la riduzione dei tempi di calcolo e del consumo energetico è sempre più una necessità. Sfortunatamente, le tecnologie di memoria standard non possono sopportare le prestazioni di calcolo previste: per affrontare questo problema vengono utilizzate nuove architetture, come l'in-memory computing. L'in-memory computing è una delle potenziali tecnologie che potrebbero consentire di migliorare l'efficienza del calcolo, riducendo lo spostamento dei dati tra la memoria e le unità di elaborazione. Una potenziale applicazione che potrebbe essere eseguita su questo tipo di memoria è legata alle reti neurali convoluzionali. In questa tesi verrà analizzato un metodo efficiente per calcolare una singola operazione di convoluzione partendo da un primo progetto su BLADE, un'architettura in-memory sviluppata in ESL: il design del controllore di questa memoria sarà migliorato al fine di facilitare la comunicazione tra l'unità di processo centrale e la memoria stessa. Infine, il design ottimizzato sarà testato eseguendo l'operazione di convoluzione.

Contents

Introduction

At a time when the demand for connected devices is skyrocketing in our daily life, the need to delegate computations to edge devices is continuously increasing. The calculation performed by these devices are more and more demanding since they were previously completed by servers or Graphic Process Unit (GPU) [1]. This new use of edge devices requires to lower their energy consumption and improve their computation capacity. One limitation that have been encountered by edge devices is the Von Neumann bottleneck [2][3]. Indeed, the current computer architecture is limited by the data transfer rate. In order to solve this issue, the idea of performing computation inside the memory has emerged. This capacity to compute inside the memory in known as In-Memory Computing (IMC). This solution minimizes data movement between memory and the Central Process Unit (CPU), but at the cost of the need to encode calculation instructions to meet the CPU interface protocol.

BLADE (BitLine Accelerator for Devices on the Edge) is a cache memory developed in the Embedded System Laboratory (ESL), whose architecture enables to perform in-SRAM (Static Random Access Memory) computing [4]. It aims to put operations in parallel while reducing the power consumption and data movement. ESL has integrated BLADE for the first time on Rosetta chip (Figure 1) in 2019.

Figure 1: Rosetta [5]

Figure 2: Darkside [6]

This chip comes from a partnership between ETHZ and EPFL, it is a PULPissimo based architecture which uses the RI5CY core. The chip has been manufactured with the TSMC 65nm technology for a clock frequency of 190Mhz. BLADE has then been integrated in a second chip: Darkside (Figure 2), in 2021. It was the result of a new partnership between ETHZ's and EPFL's laboratories. This chip as the first one has been manufactured with the 65nm technology of TSMC up to a clock frequency of 200MHz.

This master thesis happens in between the receipt of Darkside and the preparation of a new chip integrating BLADE, also the majority of the contribution of this work insert in this context.

In the scope of this master thesis, applications had to be run on Darkside in order to validate first the proof of concept of BLADE and then to highlight its efficiency to run computations. Also one of the biggest challenge was to modify the controller of BLADE in order to make it more efficient for people to run complex applications on it. This controller was good for the proof of concept of BLADE, but with a view of running Convolution Neural networks (CNNs) algorithms, a more efficient way to transmit instructions had to be found out. The objective was to design a new controller for the next chip where BLADE will be integrated in November 2022. The future chip: HEEPocrates will be a HEEP (Heterogeneous Energy Efficient Platform) based architecture which use RISC-V core.

The goal of this work was first to show the energy and computation efficiency of BLADE compared to classical architecture using CPU, memory and ALUs (Arithmetic Logic Units). Then, some complex applications such as convolutions have been run on the current controller of BLADE to evaluate its performances compared to a CPU. In a second part of this study, a new controller has been designed in order to improve the efficiency of communication between the CPU and the controller. This new design has been compared to the existing one to prove its efficiency.

Chapter 1

Background

1.1 BLADE

Whereas the existing SRAM capable of in-memory computation are suffering from several issues such as their low density and their risk of data corruption, BLADE stands out from these memories because of its special architecture that solves these issues [4]. This architecture adds few constraints concerning the placement of data for computation. The specificity of this architecture will be explained in next section.

1.1.1 BLADE architecture

In HEEPpocrates chip as in the previous ones, the BLADE memory is divided in 16 SubArrays (SAs) of 2kB each, for a total capacity of 32kB. Each subarray is made of an array of 128 x 128 bitcells. The rows of the subarrays, called sets, are subdivided into 4 32-row Local Groups (LGs) with a dedicated I/O periphery that connects the local bitlines to the Global BitLines (GBLs). Horizontally, the bitcell columns are divided into 4 interleaved ways, so that every 4 bitcell column shares one BLADE bitline logic block. A Local Group Periphery (LGP) as depicted on Figure 1.2 is associated to each LG, it allows to read or write one of its 128 bitcells at once. This structure enables to store 512 words of 32 bits in each subarray and these words can also be divided in 2 or 4, leading to words of 1, 2 or 4 bytes. Figure 1.1 describes the structure of one subarray of BLADE.

The current architecture of the macro of BLADE enables mainly 5 basic operations, including 3 bitwise operations (AND, NOR and XOR) and 2 wordwise operations (addition and the shifting of bits). These computations are possible by means of the global bitline logic that are replicated for each bit of the words and allow operations only between different LG to avoid data corruption within the bitcells. One LG being 128 addresses, one word stored in an address can be implied in an operation with the 384 words of the 3 other LG. There are in total 32 global bitline logic blocks that are connected in a carry chain. The carry chain for the addition can be cut so that these computations can be performed on 1, 2 or 4 bytes words. It also contains a write back circuit in order to write the result back in the memory after performing an operation. This feature enables to keep the IMC result in memory and allows to do more complex operations, such as the multiplications. Figure 1.3 presents a global bitline logic block.

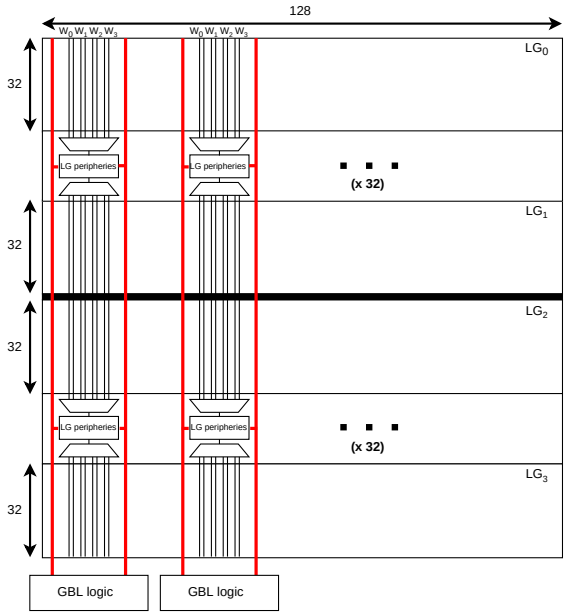

Figure 1.1: BLADE subarray architecture

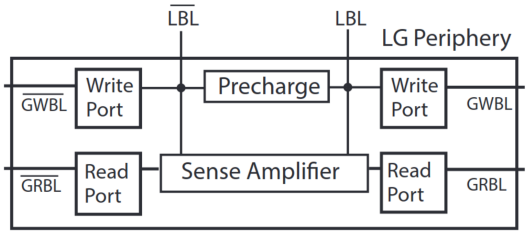

Figure 1.2: Local group periphery [7]

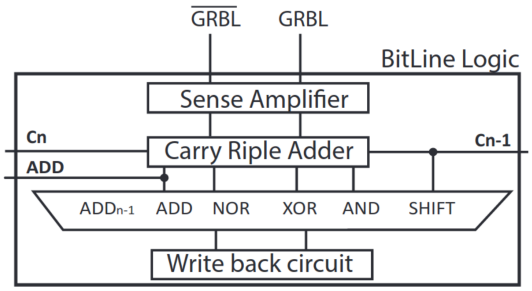

Figure 1.3: Global bitline logic [7]

1.1.2 BLADE controller [1]

1.1.2.1 Read/Write and IMCs

In HEEPpocrates, BLADE controller receives instructions from the CPU through the bus. It uses the address and data sent by the CPU not only to read or write addresses but also to decode instructions about the type of operation to be performed. Figure 1.4 explains the way the instructions are encoded in the address and the data sent by the CPU. A first observation that can be made, is that the actual memory size of BLADE is 32kB and every address could be accessed within only 15 bits. However,the range of addresses necessary for BLADE is multiplied by 8 compared to the real size of the memory. Indeed, 3 bits are used to encode the operation to be performed. On Figure 1.4, one can notice that the addresses are not increasing with the order of the memory in the macro. This is a choice that have been made at the moment to encode the address that will impact the way to run complex application.

In addition to this extended range of address, an other BLADE controller specificity is that it has 2 completely different behaviours depending on the kind of operation to be performed. Indeed, in the case of a read or write the controller becomes transparent by adopting a combinational behaviour. In this case the data signal is directly linked to the data input of the macro and the data is not interpreted by the controller. But when performing In-Memory Computing (IMC), the controller becomes sequential and it needs 4 cycles of instructions to request an IMC. This change of behaviour is set when receiving the first instruction. The controller will first check the bits 17 down to 15, if they are all zeros, it means it is a read or a write depending on the write enable signal. Then, in this case, it will consider the bits 14 down to 2 that contain the information

[1]This section was largely inspired from [8], this document is internal to ESL and has been written especially for me by a former PhD student.

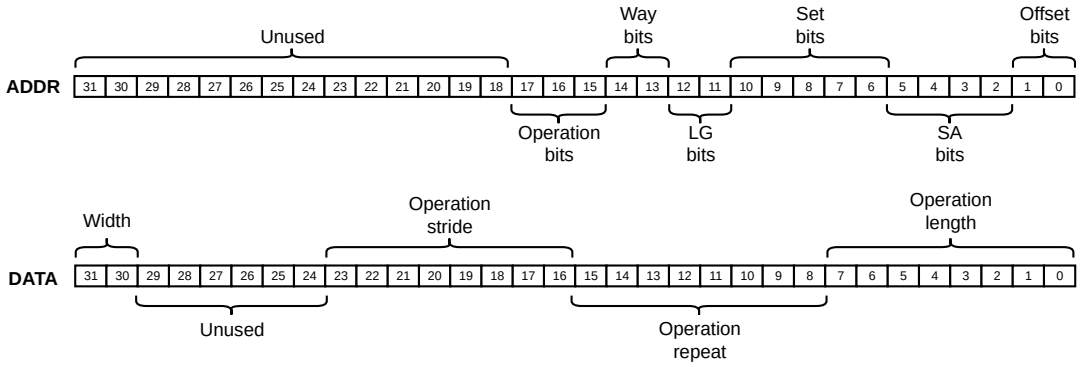

Figure 1.4: Address and data encoding

about the address to read or write. In the case of an IMC, the only important bits when receiving the first instruction are bits 17 down to 15. The operation is encoded with these 3 bits and the write enable signal. Table 1.1 sums up the encoding for all type of operations that are currently possible with BLADE. In the 3 following cycles, the information about the addresses where to perform the IMC are sent. Each cycle will enable to store the address of an operand or the address to store the result of the IMC. Moreover, it is also convenient to repeat several times the same IMC in different addresses when the data is already wisely placed in BLADE memory. Therefore hardware loops have been implemented in BLADE controller (1.1.2.2). In order to use this feature of the controller, the data that are sent in the 3 last cycle of configuration are used to tell the controller how many operations in adjacent addresses are about to be executed. The 2 Most Significant Bits (MSB) of the data signal are used in the case of an addition or a multiplication to determine the size of the carry chain. This encoding is summarized on Figure 1.4.

Operation interpretation	Operation bits	Write enable signal
Write	000	W
Read	000	R
XOR	001	W
Multiplication	001	R
Shift	011	W
Load BLADE register	011	R
Write from result register	100	W
NOR	101	W
AND	101	R
Add	111	W

Table 1.1: BLADE operations

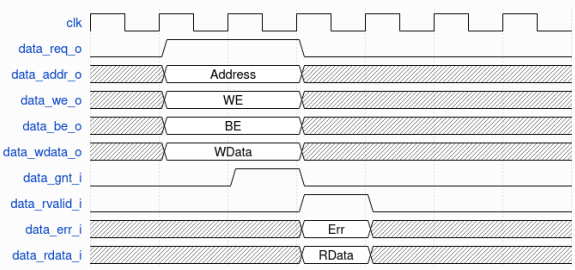

Figure 1.5: CPU interface protocol [9]

The way BLADE controller receives information enables to transmit complex instructions at the cost of several cycles of CPU clock. Indeed, the interface protocol of the CPU (Figure 1.5) makes that it can not send a new instruction until it receives the r_valid signal from the controller, which means the instruction has been performed. Thus, sending 4 different instructions to request an IMC results in 7 clock cycles: the computation in BLADE only start when the fourth instruction is received and and each of the 3 first instructions take 2 clock cycles (one for the request and one for the r_valid).
The number of cycle to do an operation also depends on the operation type. For a read or a write, the operation happens in only one clock cycle. For an IMC, the operation

take 2 clock cycles: one for the operation and a second one to write the result back in the memory from the result register. The multiplication is an exception since it uses the addition and shift features of BLADE. It requires 1 cycle to load the multiplier in some specific register and then it repeat the add and shift operation and it writes back the result in the result address.

1.1.2.2 Hardware loop

The controller of BLADE allows to repeat the same operation among a range of successive addresses. These hardware loops articulate around 3 parameters that are set once for each operand and that control different features:

- *Length*: The length value specify the number of adjacent addresses to which the same operation will be performed. It should be noted that operands containing shorter lengths will loop back to their initial address when their maximum length is reached.

- *Stride*: In direct contrast to the immediate previous sentence, sometimes the operand are have shorter length are expected to loop not at their initial address but at the address added to a certain stride.

- *Repeat*: The repeat parameter enables to loop on the same address before incrementing to the next one.

1.1.2.3 Multiplications

Multiplication in BLADE is a special operation that is more complex than other IMCs since it takes more than 2 cycles to be performed and it uses a sequence of additions and shiftings. The number of cycles to complete a multiplication is linearly proportional to the size of the multiplier. It takes in fact 2 cycles per bits of the multiplier, one to do the add and shift operation and the second one to write the partial result back in the memory. An extra cycle is necessary at the beginning of the multiplication to store the multiplier in a special register (*bladereg*) of BLADE present in each SAs. At each operation cycle, the MSB of the *bladereg* is considered. If it is a '1', the multiplicand is added to the partial product. The sum is then left shifted by one bit and stored back to the memory, and the multiplier is left shifted by one bit in *bladereg*. This loop continues until all bits in the *bladereg* have been consumed. The number of cycles to perform a multiplication is therefore 17, 33, or 65 for 8, 16, and 32 bits multiplier values, respectively. Finally, the last cycle of the multiplication is an add without shift. As previously, the add occurs only if the MSB of the *bladereg* is '1'. Figure 1.6 illustrates this procedure for a 4-bits multiplication.

When performing a multiplication, the *Repeat* parameter takes a special function. Since the multiplier is store in *bladereg*, it should be noted that the multiplicand and the multiplier are not necessarily of the same size. The multiplier size determines the number of recurrences of the process, whereas the size of the multiplicand sets the carry chain. Thus in one word up to 4 multipliers can be stored in the *bladereg* and the *Repeat* parameter can enable to do multiplication with the 4 adjacent addresses of the multipliers.

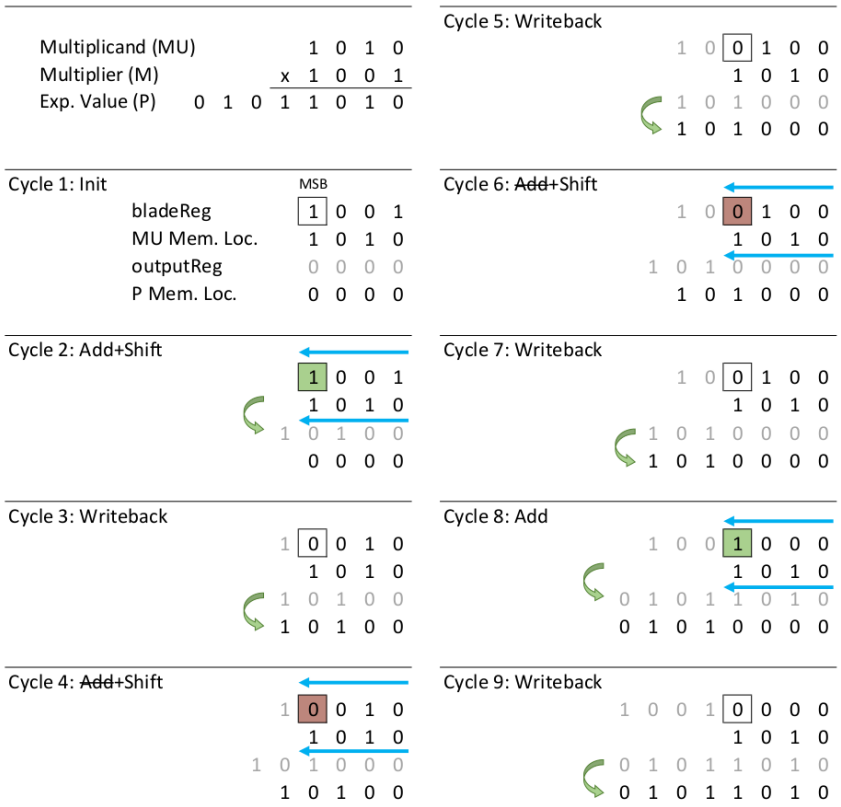

Figure 1.6: Multiplication example [8]

Chapter 2

Programming applications for BLADE

2.1 Basic operations

A simple way to test BLADE is to write commands in C. The test script of BLADE is mainly composed of a sequence of command to tell the CPU to perform store or load in the range of addresses of BLADE. Since the way to do an IMC is always the same and the only parts that change are the addresses, the number of IMC to perform and their type ; it was easy to create functions in C that were able to request IMCs. At first, 7 functions were coded in C to enable to do the read and write operations along with the 3 bitwise IMC, the addition and the multiplication. For the sake of easiness, the spirit of the code optimizations will be described in the case of a read (all the functions are available in the Annexes). The IMCs code was containing more variables and was a bit more complex. Indeed, due to high flexibility of the controller many parameters had to be transferred. Figure 2.1 shows the C code written to perform a read. In this code, it should be also highlighted that a lot of computations are performed on the address and data to be sent to BLADE. This will result after compilation in extra instructions for the processor to accomplish. Some improvements in term of efficiency can be still done by changing the power and multiplications by shiftings, but these changes are not solving completely this issue. The main problem come from the fact that we encode both the addresses and the data to request an IMC on BLADE.

```
1 unsigned int Read (unsigned int addr, unsigned int sa)
2 {
3     unsigned int addr_offset = pow (2, 6) ;
4     unsigned int sa_offset = pow (2, 2) ;
5     unsigned int *ptrBlade = (unsigned int *) 0x80000000 ;
6
7     ptrBlade = (unsigned int *) ((char *) ptrBlade + addr * addr_offset +
    sa * sa_offset) ;
8     return *ptrBlade ;
9 }
```

Figure 2.1: C code to perform a read with BLADE

This C code is translated into assembly instructions after the compilation. The result

is depicted on Figure 2.2. These assembly lines correspond only to the read function, but extra lines in the main script to load the variables used in the function must also be considered. Here, the line 564 is the one that corresponds effectively to the load instruction. The other lines are just enabling computations to encode the information in the address. When several reads are realized in a row, the simulations show that these parasitic instructions can take until 20 clock cycles, thus it is not easily possible do a serie of load in a row. The same issue is encountered with the IMCs.

```
00000556 <Read>:
 556:   051a                    slli    a0,a0,0x6
 558:   058a                    slli    a1,a1,0x2
 55a:   95aa                    add     a1,a1,a0
 55c:   962e                    add     a2,a2,a1
 55e:   800005b7                lui     a1,0x80000
 562:   95b2                    add     a1,a1,a2
 564:   4188                    lw      a0,0(a1)
 566:   8082                    ret
```

Figure 2.2: Assembly instructions for the read operation

These extra instructions and cycles of clock are usually not a problem in the case of a small application or for functional tests, but as soon as we want to write more complex code or to measure the efficiency of BLADE as an accelerator, it can be an issue. For these reasons, it is necessary to find a way to circumvent these instructions. 2 solutions have then been found to avoid any unwanted instructions in the assembly code. The first idea was to insert some assembly instruction directly in the C code. The big advantage of writing in assembly is that there is a strong correspondence between the instruction in the language and the machine actions. This way, it allows to be very specific about the instructions that the machine should follow.

An other important idea to gain efficiency in the processor actions, is to write a script in python that enables to do all the computation necessary to encode the address and data. Then, the python script will generate a main.c file containing mainly assembly commands reducing greatly the number of instructions processed by the CPU. Figure 2.3 shows the python code that enables a better optimization of the read operation with BLADE.

This last piece of code is much more efficient than the previous options tested. It avoids to spend any clock cycle for the CPU computing the next address to access and the data to send, since all these computations are made during the python code generation. Thus, this code enables to test BLADE efficiency to perform reads. However, the main drawback of this code is that it forces to unroll all the program. It is not possible to use loops in this case or functions and the entire program is a list of successive reads and writes. The C code becomes very heavy and a lot of data and instructions have to be stored in the CPU memory. Despite these drawbacks, the same solution has then be applied to the code to do IMCs. Indeed since the protocol to request an IMC is to use 4 cycles of read or write in a row, the functions read and write optimized can be reused for the IMCs.

```
def Read (f, addr, sa, offset) :
    base_addr, offset_addr = OperationInit (0, addr, sa, offset)

    f.write ("\t__asm__ volatile (\"lw x0, " + str(offset_addr) + "(%[
    ptrblade])\" :  : [ptrblade] \"r\" (" + str("0x%08x" % base_addr) + "))
    ;\n\n")


def OperationInit (op, addr, sa, offset) :
    blade_intermediate_addr = op * cst.op_offset + addr * cst.addr_offset +
    sa * cst.sa_offset + offset

    offset_addr = blade_intermediate_addr % 2048
    parity_addr = blade_intermediate_addr // 2048

    if parity_addr % 2 == 1 :
        offset_addr -= 2048
        base_addr = cst.blade_base_addr + blade_intermediate_addr -
    offset_addr
    else :
        base_addr = cst.blade_base_addr + blade_intermediate_addr -
    offset_addr

    return base_addr, offset_addr ;
```

Figure 2.3: Python code to optimize a read with BLADE

2.2 Convolutions

One of the main application considered for BLADE at the moment is its use as an accelerator for Convolution Neural Networks. In the scope of this master thesis, a single layer of CNN has been simulated in order to demonstrate the power and calculation efficiency of BLADE in such an application.
In order to do this test, it has been decided to work with a convolution layer which input data was a 32 x 32 array. The filters were composed of a set of 6 kernels of 5 x 5 size (5 kernels for the 32-bits convolution). The output is hence a 28 x 28 x 6 array (28 x 28 x 5 array for the 32-bits convolution). In order to program a convolution on BLADE, 3 main problems have to be solved: first the distribution of data between the different SAs has to be defined, then the algorithm of the convolution must be implemented and finally the data placement within a SA has to be decided.

2.2.1 Data distribution between the SAs

Considering that all the 16 subarrays are performing the IMCs at the same time, the distribution of data in the memory has to be thought in order to optimize computations. Figure 2.4 shows the optimal data distribution between the subarrays in order to avoid as much as possible the data movement and redundancy. Each color corresponds to the data going in one SA. For instance, the data in green goes in the first SA, the red one in the second SA, the orange in the third one and this logic is repeated over all the SAs. The

total number of addresses available per SA is 512. It should then be possible in theory to store the 6 filters in each SA, a 6 x 18 portion of the convolutional layer input leading to 2 x 14 x 6 output. With this distribution, 150 addresses are used to store kernels, 108 are used for the input data and 168 are used to store the results. Thus, 426 addresses in total are filled and the rest can be used for the intermediate computations. Having the 6 filters in all the SAs enable to keep them during all the computation and limit the data movement. As we can see on Figure 2.4 in grey, the convolution layer input has to overlap between the different SA. This overlapping can not be avoided and is inherent to the way to compute the convolution.

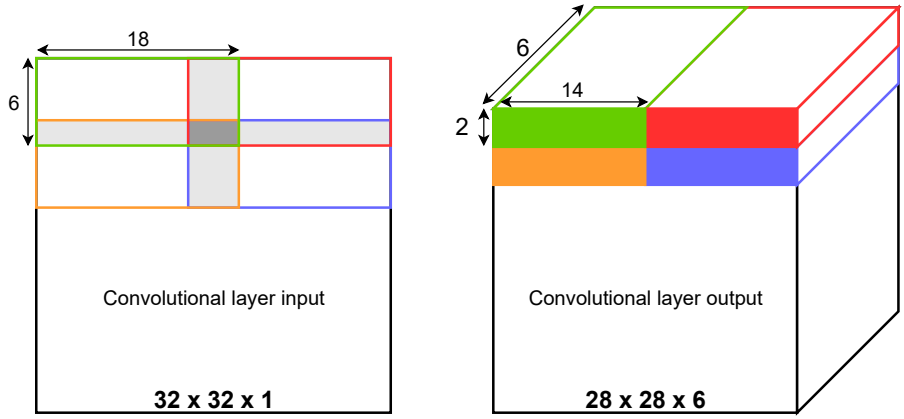

Figure 2.4: Data distribution in the subarrays

Indeed, to compute 2 lines of the output of the convolution, 6 lines of the input are necessary and out of these 6 lines 4 of them will be reused in an other SA for other computations. The same kind of overlap also applies to the columns. This figure give a rough idea of the way to place data in the memory, it determines where the data goes in the subarrays. However, the placement has to be defined more precisely within each SA to ease the computation. This placement is mainly dependent on 2 factors: the data width and the LG organisation that does not allow to make IMC inside the same LG. This placement within the SA will be defined in Subsection 2.2.3, first the convolution algorithm will be defined. For sake of easiness, the first test program have been written for 32-bits convolutions. Only this algorithm will be described since the data size has a low impact on the code.

2.2.2 Convolution algorithm

The convolution algorithm can be separated in 3 different parts: the initialization of the memory addresses and data placement, the computations and the reading of the memory. In this part, the indexing of the data and the principle of the convolution will be further explained.
The choice of the index of the data is important and can ease the code for the convolution since the address are usually the same than the index but with an offset. Figure 2.5 shows both the way the filter is moved across the data and the indexing of data. For instance, in order to compute the result of the first output, all the index in the red square have to

be multiplied by all the index of the first filter. Once the data has been multiplied all the partial result are summed, leading to the first result which will be know as output 1. In reality, the process of multiplying and summing is a bit different to avoid to loose space and time. In fact, each multiplication is directly summed to the result address where we accumulate the output. Once the first computation has been finished, the red square in the input data is shifted by one column for the new piece of computation. This square is displaced all over the input data to compute the 28 first results and once it is done, the same steps are repeated with the second filter and the next one until all the possibilities of calculation have been computed.

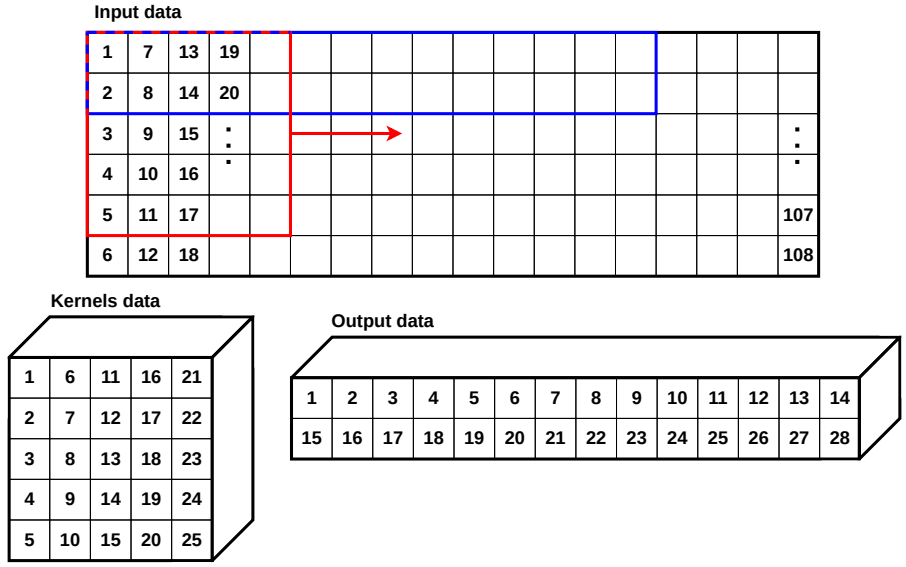

Figure 2.5: Indexing of the data for the convolution

In the end, the indices of the kernels goes from 1 to 25 for the first filter, 26 to 50 for the second one and until 125 for the last filter. The output indices go from 1 to 140. These indices are also very important, since the data must not be mixed during the reading of the results. Indeed, this convolution is only a small part of the CNN calculations and should be reused for further computations, the output will then be used as an input for the next convolution layer. For this reason, one should be able to perfectly know how the data has been ordered.

Now, that the principle of the indexing has been defined, the algorithm for the convolution can be further explained. The basic block of the convolution is described by Algorithm 1.This algorithm mainly needs to know the input index it will start with, that is to say the upper left corner of the red square on Figure 2.5. It also requires to know the starting index for the filter. Then an accumulation address which is the same in all the subarrays is reset to zero performing an AND logic operation. This address must be reset to zero in order to be the result address for the multiplication. If this address is not set to zero at the beginning, then the result of the multiplication will not be correct. After each multiplication, the result is added to the final result address. This way, the result address is accumulating the partial result until the end of the computation.

Algorithm 1 Convolution basic block

```
1: procedure CONVOLUTION BLOCK(start_input, start_kernel)
2:     for index_i in index_input_list (start_input) and index_k in index_kernel_list
   (start_kernel) do
3:         Logic AND between the reset and accumulation addresses
4:         accumulation = index_i * index_k
5:         output = output + accumulation
```

The entire algorithm is made of a repetition of the previous block. Algorithm 2 explains all the steps from the initialization to the computation (the entire code is available in the Annexes). First all the subarrays are filled with the data and the reset address are set to zero.

Algorithm 2 Convolution

```
1: procedure CONVOLUTION
2:     // Initialization of the SAs
3:     for each subarray do
4:         Place the kernel data
5:         Place the input data
6:         Initialize the reset addresses to zero
7:     // Initialization of the result addresses
8:     for i in the result addresses within a subarray do
9:         Do a logic AND between the address i and the reset address
10:    // Convolution
11:    for filters going from 1 to 5 do
12:        for start_input in start_input_list do
13:            Convolution block (start_input, filters)
```

These reset addresses will be further used to set result addresses and accumulation address to zero by a simple logic AND. It enables to gain efficiency since the IMC happens in all the SAs at the same time, this way it is avoided to write 16 times zero at the same address in all the SAs. The convolution block is then called several times using as start the start_input_list which correspond to the indices in the blue rectangle of Figure 2.5. This block is repeated over all the filters to complete the entire convolution. The principle is the same whatever is the size of the data (32, 16 or 8 bits), the only thing that will change is the placement of the data within the subarray. The complete code for the convolution has been written in python to have a maximum efficiency in the convolution computations, as seen in Subsection 2.1. The python code compute all the addresses to be read or written at the compilation and it generates a C file with a list of read and write instruction in assembly to have a maximum control on the compilation of this second code. This way of doing enables to have a good mastery on the compiled code but it generates very large C files (until 600 000 lines for a 32-bits convolution). The limitation of this method will be further discussed in section 2.3.

2.2.3 Data placement within the subarrays

The criteria for the data placement are simple to understand, but difficult to fulfill. Ideally, it would be simpler to only use one LG for the kernels, one for the input and the 2 left for the result. This way there is no more constraint when performing IMCs, since the operands are in different LGs. It should be also noted that the more the SAs are filled with data the better the efficiency.

2.2.3.1 32-bits convolution

Figure 2.6 shows the first placement of the data within a SA. This figure represents an entire SA and for the sake of understanding, the ways are not interleaved on the schematic. That being said, each of the 16 squares represents a range of 32 addresses in the SA. The addressing goes from 0 up to 511 starting by way 0 increasing with the LGs and then increasing the way numbers.

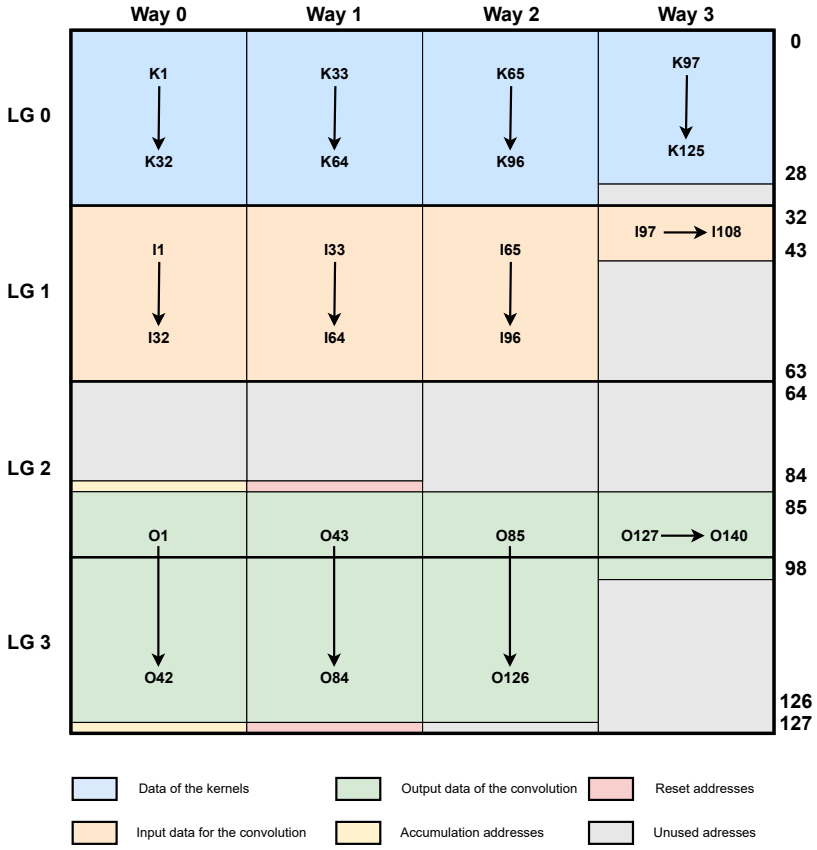

Figure 2.6: Data placement within the SAs for a 32-bits convolution

First thing to be noticed, with 32-bits data it is not possible to fit all the kernels of the convolution in only one local group. They would actually fit in the SA, but for sake of easiness it was better not to fill a second LG with these data. Thus, only five of the six filters were placed inside the memory. The input data suits totally in one LG and the results are placed in the 2 LGs left. Two addresses have been kept to accumulate the result of the multiplication and two other one have been used as reset addresses. One of

each kind has been placed in LG2 and LG3. Indeed, to be able to reset the accumulation address or to add the accumulation address to the result one, the 2 addresses must be in different local group. This placement results in the end with a important part of the SA that is unused neither for storing data, nor for the intermediate results (26.4 %). This is a problem for the convolution, because the size of the data input being important compared to the size of BLADE memory, the convolution is performed in two times. First, a portion of the input is written in the memory. Then, the result of this part of the convolution is computed and is read. These steps have to be repeated twice to complete the entire convolution. For this reason the unused part of the addresses is a loss of efficiency, but it can not be easily solved in this case. If an input line is added in the SAs, it takes 18 addresses and the additional results coming from this data are using 5 x 14 addresses. It results in the end in 88 additional addresses used and it can not fit in the SAs.

Number of clock cycles	BLADE
Instructions	154 016 (11.9%)
CPU computations	654 014 (50.3%)
BLADE Computations	491 416 (37.8%)
Total	1 299 446

Table 2.1: Use of the clock cycles during the 32-bits convolution

As BLADE is being optimized for running CNN computations, it is primordial to understand what type of operations take more time to perform and to find ways to improve their efficiency. For the sake of designing a new controller for BLADE, it was very interesting to have an idea on the proportion of the clock cycles that were used to send instructions to the controller and the actual time spent for the computations. The Table 2.1 shows the results of the convolution simulation on BLADE with a RISC-V core CPU. This result give an idea of the limitations of BLADE with the actual version of the controller. First, thing that should be noticed, the time devoted to the actual computation inside BLADE is very short, only 37.8% of the clock cycles are computing cycles. All the rest of the time is used either to do internal computation in the CPU, to transfer information through the memory bus, or to receive instructions for BLADE. This is an obvious weak point of BLADE that would need to be overcome for the next version of the controller. The problem is mainly that it takes seven clock cycles to send an instruction for any IMC. It is not a major issue if hardware loop are used and that several IMCs are about to be executed, but in the case of a single bitwise operation, it takes seven clock cycles to send the instructions and only two to do the computation (one cycle for the operation and one to write back the result in the right address of the memory). To these instruction cycles adds an overhead due to preparation of the address and data to be sent by the CPU. This ratio between instruction time and computation time is not acceptable for running complex applications. This loss of cycles must be solved absolutely to give more flexibility to the future controller. An other big issue that have been faced with the convolution is the time dedicated for the multiplication, that is much larger than any other operation. Figure 2.7 shows the distribution of the clock cycles allocated for each operation considering both the instruction time and the computation time. Looking at the pie chart, it is striking that the multiplication is the longest operation. Indeed, more than three-quarter of the operation cycles are dedicated to multiplication. It is mainly due to the fact that multiplication is a sequential operation in BLADE, it is made of a succession

of addition and shift. Moreover, the way multiplications have been implemented takes a fix number of clock cycles to be performed. For 32-bits multiplications, it takes 7 clock cycles to send the instructions and 65 to complete the multiplication. It is by far the longest operation performed on BLADE. In order to improve the multiplication time, one proposed option is to add the embedded shift to the design of the subarrays. The embedded shift are consisting in way to reduce the number of cycles to complete the multiplication by checking several bits of the multiplicand at the same time to make several shift in a row when there are adjacent "0" in the multiplicand. Even though, the multiplication is the slowest operation on BLADE, it must be noticed that its efficiency to do multiplication keeps being competitive compared to a CPU, since in reality 16 multiplications are performed in parallel in all the SAs. This parallelization of operation largely increases the throughput. In the case of this convolution, 98 000 multiplications can be performed in 1 013 568 clock cycles which corresponds to a throughput of one multiplication every 10.3 cycles. In order to be able to accelerate the multiplications, a possibility is to work with 16-bits or 8-bits data, which reduces largely the number of clock cycles required to finish the operation and increase the number of operations done in parallel.

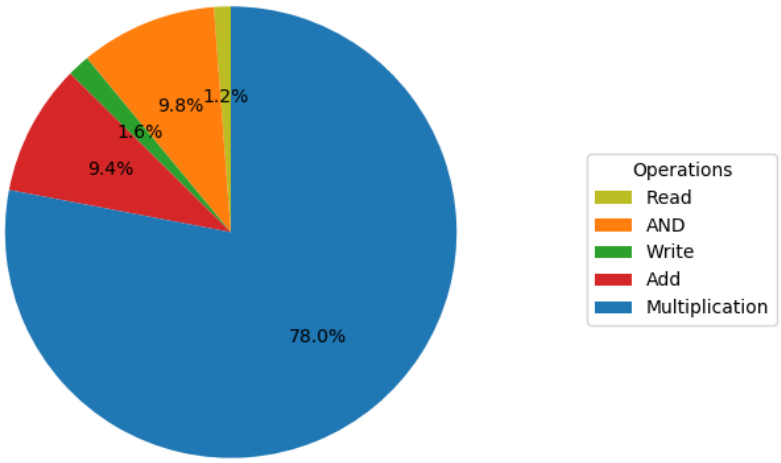

Figure 2.7: Clock cycles proportion per operation for the 32-bits convolution

2.2.3.2 16-bits convolution

In order to accelerate the convolution, one option that have been explored is to reduce the data size. Indeed, the shorter the data, the less cycles it will take to complete the multiplications on BLADE. This data width reduction is legitimate since most of the data within a CNN are usually small and close to 0 [10]. The reduction of the data size have an important impact on the data placement and if the data is wisely placed in the memory, it is possible to significantly increase the efficiency to perform a convolution. The guideline for the placement of data within the subarrays was a bit different from

the 32-bits convolution. Unlike, in the previous algorithm, the kernels were loaded as the multiplicand and the input data was considered as the multiplier. This choice might seem not relevant since the multiplication is usually an associative operation, but for BLADE it is not, as explained in Section 1.1.2.3. The decision of the multiplicand and multiplier is crucial in this case since it enables to compute at the same time the result of the convolution for 2 different filters. An other import decision concerning the data placement is to put the accumulation address in the same LG than the inputs. During the multiplication only the kernels are added to the accumulation address which avoid any trouble because of operation between data in the same LG.

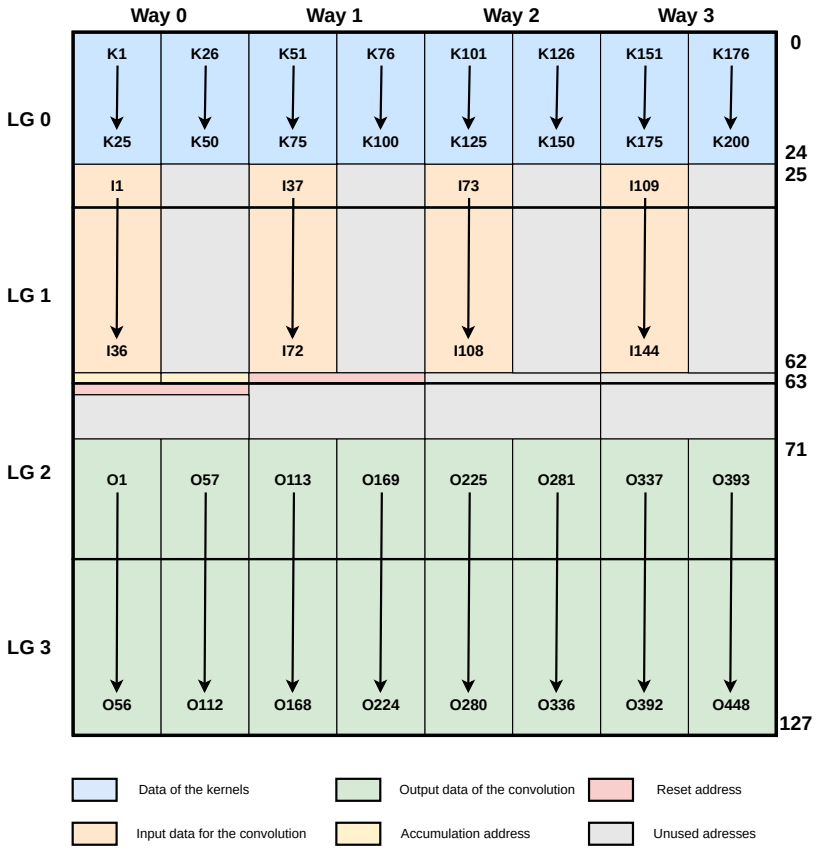

Figure 2.8: Data placement within the SAs for a 16-bits convolution

Finally, the result addresses and the accumulation address must absolutely be placed in different LG, since addition are performed between these addresses. Figure 2.8 shows the placement decided for the 16-bits convolution. This figure is still representing the entire subarray, but unlike the Figure 2.6 the squares that represent a range of 32 consecutive addresses have been separated into two rectangles depicting the 16 Most Significant Bits (MSB) and 16 Less Significant Bits (LSB) of a word. The left rectangles represent the MSB and the right ones the LSB. The addressing still increase with the ways.

Concerning the data placement, it should be noticed that for the kernels, the first filter have been placed in the MSBs and the second filter has been placed in the LSBs of the same word. This placement have been repeated for the 8 filters placed in the SAs. By the way, it should be also noted that there is place for 8 filters. That is more than the

number initially required, but more space is available when reducing the size of data. For the input data of the convolution, since it is used as the multiplier, it has to be placed in the MSBs of the word. The LSBs have been left unused for easiness of computations and to avoid data redundancy. Also, it should be seen that more data have been placed in the SA and it enabled to store all the data to do the convolution in one time. There is no more need to load the data in 2 times. This improvement is only possible due to the extra space available thanks to the data size reduction. Reducing the size of the data, the place available in the SA doubles. Finally, the result of a way correspond to the computation made with the filters of the same way for simplicity. The MSBs of the result are related to the filter saved in the MSBs of the same way. The same logic has been followed for the LSBs of the result.

The reduction of the data size, beside changing the data placement, has a strong impact on the speed of the algorithm. Table 2.2 shows the simulation results of a 16-bits convolution on BLADE with the same CPU than Table 2.1.

Number of clock cycles	BLADE
Instructions	93 248 (14.3%)
CPU computations	397 258 (61%)
BLADE Computations	161 288 (24.7%)
Total	651 794

Table 2.2: Use of the clock cycles during the 16-bits convolution

It should first be noted that the total number of instructions have lowered since 2 results are computed at the same time in the same SA. The reduction of the number of instructions sent to BLADE have mainly 2 impacts. It first obviously reduces the number of clock cycles dedicated to the receipt of instructions by BLADE, but it also largely decreases the number of clock cycles used by the CPU to send addresses and data to BLADE. Reducing the size of the data is also strongly reducing the number of cycles to perform the multiplication, which significantly impacts the number of computation cycles. The data size reduction improves the time required for the computation but have no direct impact on the instruction time. It highlights even more the fact that computation time in BLADE is loosing in proportion (24.7%) when reducing the data bit-length. As ideally, the proportion of BLADE computations should be maximized to speed up the convolution, it shows that the way to send instructions to the controller impacts more the efficiency of BLADE with small bit width data. Not considering the instruction time proportion, the impact of bit width reduction on the operation cycle is striking since the multiplication time is divided by 4 (the number of multiplication done with one instruction is doubled and the time of execution is divided by 2).

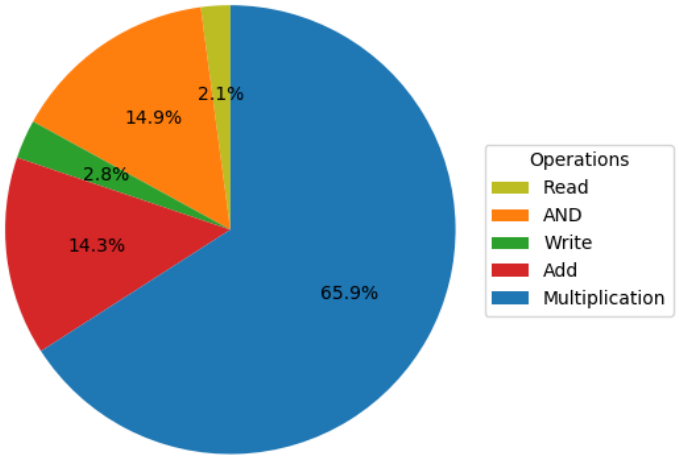

Figure 2.9: Clock cycles proportion per operation for the 16-bits convolution

Figure 2.9 shows the proportion of clock cycle taken by each operation. The multiplication keeps being the operation taking more time but its proportion have reduced significantly compared to Figure 2.7. This can be explained by the increase of the throughput. In this convolution, 117 600 multiplications are completed in 429 532 clock cycles, this correspond to a multiplication result every 3.65 cycles. Reducing again the data bit-length to 8 bits would again improve the efficiency by 4. These results clearly show that one important asset of BLADE is its capacity to make in parallel a lot of operations. This asset is however less significant when fewer operations need to be completed and that all SAs are not filled.

2.3 Application constraints

This part of the study have enabled to find out first a way to optimize the code for running applications on BLADE. These optimizations however show some limitations in their efficiency. On the one hand, assembly is not the most user-friendly language to program applications and it is very challenging to optimize application at the instruction level. Thus, it would be a good option if it was possible to run a complete application writing only python or C. On the other hand, the idea of writing python code to generate a main file where all the encoding for the data and addresses are computed, efficiently speeds up the process. However, the code length also increases drastically. The entire code for the convolution becomes so long that it was not possible to store the complete program in the memory of the chip that was used. The whole size of memory available for the code was 512kB and the application size was around 1MB. This problem comes mainly from the fact that the python code generating the *main.c* unrolled completely the program and no function were written in C. For this reason, the compilation time of the C was extremely long.

The convolution applications also were enlightening on the hardware design limitation of the controller. As it has been showed on Section 2.2.3, the main default of the controller is the way it receives instructions for the IMCs. Indeed, the instruction reception that is performed on 4 cycles is really not optimal and for the 16-bits convolution, since the operation cycles are lower, the instruction time is proportionally increasing. This problem is even more striking for the IMCs that are performed in 2 clock cycles since the instruction time is longer than the computation time. This limitation of the hardware must be circumvent by finding a way to pass the instruction to the controller in only one clock cycle.
In addition, to the inefficiency of the communication with the controller, an other issue encountered is the impossibility to use hardware loops for the convolution applications. The problem comes in fact from the memory mapping of BLADE that have not been thought to enable this feature. The addressing of the memory increases within the ways and not within the LGs. When running applications on BLADE, the operand of a same type are often placed in the same LG to avoid problems of operations that can not be done. Having a mapping of the memory that is ordered correctly would enable to largely use hardware loop at least for the smallest repetitions of the convolution.
The last important point to be mentioned concerning the convolution, is the way BLADE performs multiplications. The multiplication is done on a fix number of clock cycles that is determined by the controller. This is not a very efficient way to make a multiplication compared to ALUs that are optimizing the number of clock cycle to perform multiplications. As it has been showed, the time for a multiplication is linearly proportional to its multiplier width. However, this time to perform multiplication that is irreducible, is balanced by its capability to perform several operation in parallel. Indeed, a multiplication is done at the same time in all the 16 SAs which enable to reduce the number of clock cycles per operation by 16 if the memory is correctly filled. This parallelization of operations can even increase more when the size of the data is reduced to 16 or 8 bits. In the best case, until 64 multiplications can be achieved in 17 clock cycles, which leads to a throughput of 3.76 multiplication per clock cycle. What can be noted about this way to perform multiplications is that it really is efficient only when the subarrays filling is optimized. This is a good solution with the view of running CNNs applications on BLADE, but application must be smartly written to consider these constraints of BLADE.

Chapter 3

A new controller design

As it has been described previously, the current version of the controller is somehow limiting for running complex operations. Some choices that have been made were not considering the fact that users may at some point write applications with BLADE. This first part of the study motivated the need for a new controller that would be more efficient for receiving instruction, more flexible and also more user-friendly. With this new motivation, a way to ease instructions for both fine or coarse grain operation have to be found. The idea is to find a way to communicate that would be efficient both for a single operation or hundreds of operations in a row. This Chapter will concentrate on the hardware design considerations.

3.1 Design specificity

3.1.1 State registers

The previous version of the controller was limited by the way to receive instructions because they were encoded in the address and data sent by the CPU. The address was decoded instantaneously by the controller at the output of the bus. This way of proceeding was triggering a direct decoding and the information once decoded were stored in registers waiting for the next instructions. Figure 3.1 shows the way the previous controller was interfacing the bus. It had mainly 2 disadvantages. First, the range of addresses necessary for BLADE was much larger than the real memory size due to the 3 operation bits (see Figure 1.4). Then, the number of instructions that could possibly be passed to the controller was limited to the data length. The solution that have been adopted for the new controller was to integrate few state registers to the controller. These registers are part of the range of memory and are used to be written or read by the CPU. The controller can access at any time the value of these registers and is able to configure the operation to perform from there. Figure 3.2 resumes the new organization of the controller. In the end, when we compare the 2 approaches, the new one enables first to use a smaller range of addresses for running the same operations on BLADE, it is in fact almost divided by 8 since only 15 bits are required for accessing the entire memory, we need to add only 5 extra addresses for the state registers that will enable to do the same operation than the old controller. Now, all the addresses of the memory can be written or read as any other memory by the CPU without encoding the addresses and data. Using these registers is

more suited for the applications that are running on BLADE since it only requires to write the first state register that contains the addresses of the 3 operands (operand A, operand B and result) and the operation code to launch the operation for the controller. The other registers (op A, op B and res) are used to store information for the hardware loop, such as the *length* and the *stride* mentioned in Section 1.1.2.2 and the width of the operand. The width of the operand makes sense only for the terms of the addition and the factors of the multiplication. The result size is always defined by the size of the first operand and the bitwise operation do not care about the size of operand. Finally, the last register is used for a new type of operation added on the new controller that performs a basic block of the convolution composed of a logic AND, a multiplication and an addition. For this new operation, the information about the 3 operands used for the other IMCs are not sufficient and 2 extra addresses must be given. The first one is the address of accumulation previously presented in the last Chapter that is used to store the partial result of the multiplication. The second one is the reset address where the data is kept to "0x00000000" during the entire convolution and that is used to reset the accumulation address before starting a new multiplication. All these registers are not necessarily written for each new operation and only the first one has to be written to start an IMC. This way of sending instruction find a justification by the fact that for a convolution the operations are very repetitive and these registers are written only once at the beginning of the application and then they keep being the same for the entire test. This section explained the novelties found to enable a more efficient communication with the new controller, but the controller is not made of only one file and the architecture of BLADE peripheral will be discussed in the next section.

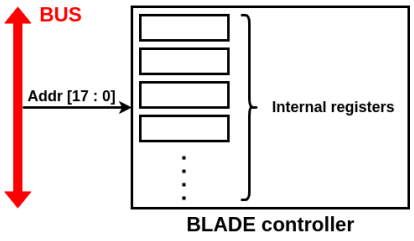

Figure 3.1: Previous controller interfacing with the bus

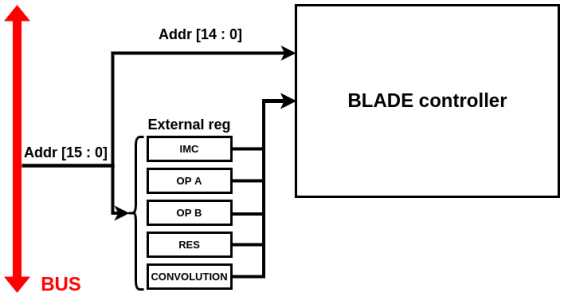

Figure 3.2: New controller interfacing with the bus

3.1.2 Architecture of BLADE peripheral

This new controller is supposed to be integrated on the next tape out involving BLADE and must integrate perfectly with the bus signals of HEEP. A challenge of this work was to organise the file hierarchy in order to be able keep the structure as simple as possible. Figure 3.3 shows the BLADE peripheral architecture. The toplevel is the file that interface the bus of HEEPpocrates with the peripheral, all the input and output signals are the one defined in the bus. Inside this file are instantiated all the blocks of BLADE. The controller file manages the state registers since it is generating the *gnt* and *r_valid*. It is more convenient to deal with the output signal in only one file to avoid having to much logic in the toplevel file. The H tree file is responsible for the interface

between the controller and the subarrays. It brings the signals to all the SAs and manage the cases of read and write that apply only to one subarray. Finally, a wrapper for each SA have been added in order the reorganise the addressing of BLADE. This new file enables to reshape the memory mapping without having to restart all the full custom design of the memory. This wrapper aims to have continuous addresses within the LGs of the SAs. This reorganisation of the addresses eases the use of the hardware loops and is supposed to ease the convolution application on BLADE. It is also possible to imagine in the future that several way of addressing could be used depending on the application that would be running and that they would be controlled in an other state register.

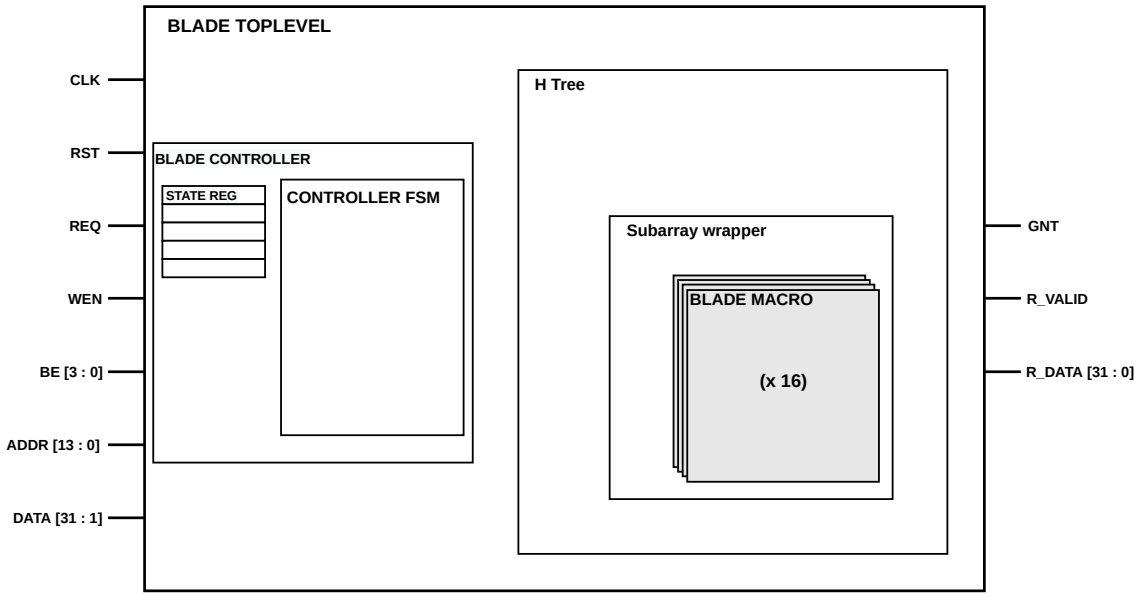

Figure 3.3: New controller general architecture

3.1.3 FSM of a new controller

The global architecture being explained, a presentation of the controller file has to be done. This file as all the others is written in System Verilog for sake of homogeneity of the code. This code being more complex than the other file, it was relevant to organise it differently. The idea of writing the controller HDL (Hardware Description Language) as an FSM was an evidence. The state were simple to imagine and a first FSM has quickly emerge. To ease the code at the beginning the choice have been made not to directly write a controller with hardware loops since they can add on top of the basic features of the controller. The first version of the FSM is depicted on Figure 3.4. Of course, the working of the controller being very complex, only the condition to change states have been sketched to make it understandable. All the output signals have been omitted. On this first FSM, 5 states have been defined. *Wait* is the starting state in which the controller returns each time it completes a computation. The four other states are operations states that take a different number of clock cycles and they need to be considered independently. The *Read/Write* state lasts only one clock cycle, so the controller come back in the initial state right after the operation and without any condition. The *2-cycles IMC* state corresponds to all the operations that take 2 clock

cycles (AND, NOR, XOR, Add, Shift): first computing, then writing back the result in the memory. These operations are started when the operation code is smaller than 5. The counter for the operation time is set to 1 in order to quit this state after 2 cycles. The state *Multiplication* correspond to the multiplication as the name suggest. It is treated separately since the multiplication takes more cycles as we explained previously in the first Chapter. The operation code is 6 and the initialization value of the counter is dependant on the muliplier width. The multiplication is done by a succession of "*addshift*" and "*writeBack*". The last cycle is just an addition without shift. These elements have to be all managed by the controller which request a good knowledge of BLADE to understand its way to perform operations. The last state of this FSM is the *Convolution block*, it is a state that performs 3 IMCs in a row. First it does a logic AND between the reset address and the accumulation one to prepare it for the second step which is the multiplication between the operand A and the operand B. The result is stored in the accumulation address and finally an addition is performed between the result address and the accumulation one. The final result is stored in the result address. This operation state need one extra counter to count the operation step. Once both the counter of the operation and the step counter are null, it means that the result is computed. Each time one operation step is finished, the step counter is decreased and the running counter is reinitialized for the next operation to perform.

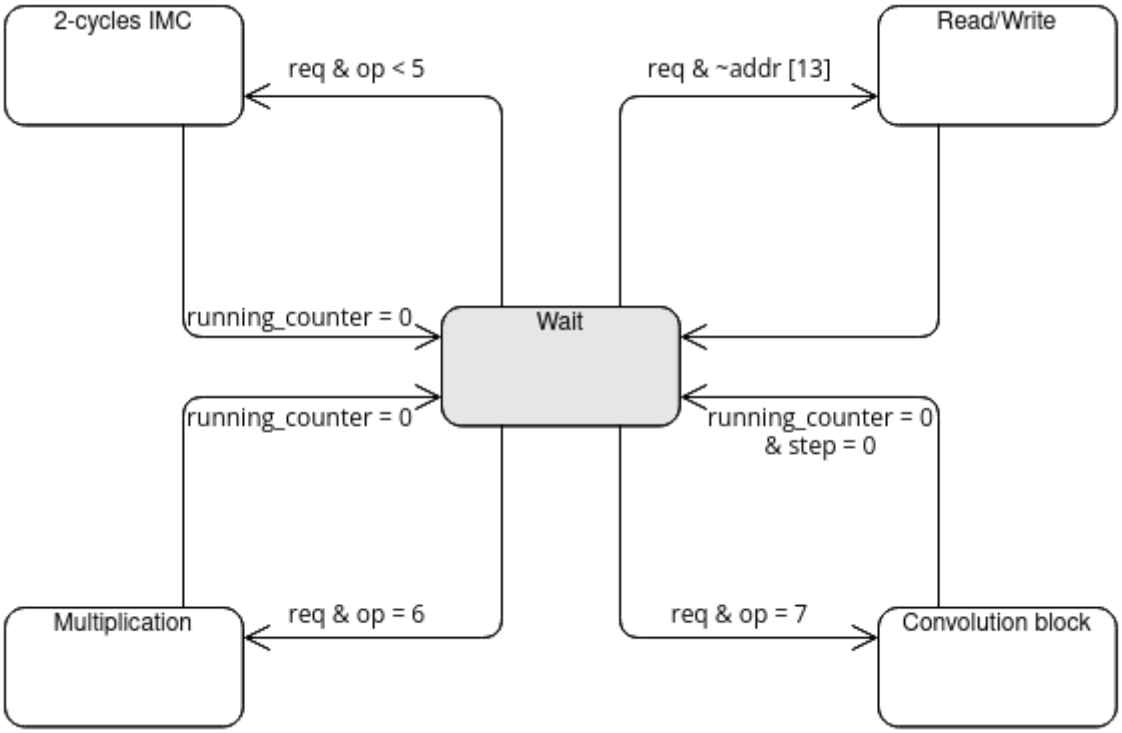

Figure 3.4: FSM of the new controller without hardware loops

This solution was a first version that have been coded for the controller and one feature: the *Convolution block* have been added. However, the addressing of the memory have been changed in order to find a use for the hardware loops. This FSM have thus been changed in order to integrate this feature. The only thing that have been removed is

the *Repeat* parameter since it was less interesting for the applications. Figure 3.5 shows the simplified diagram of this FSM. On this diagram, the same states have been used and only one extra state have been added: *IMC*. This state has been added in order to ease the implementation of hardware loops. Indeed, since this feature requires to load specific counters, this step is done in this state and when an operation is finished if all the *length_ counter* for the 3 operands are not equal to 0 the *finished* signal stay to '0' and the hardware loop counter are updated in *IMC* state.

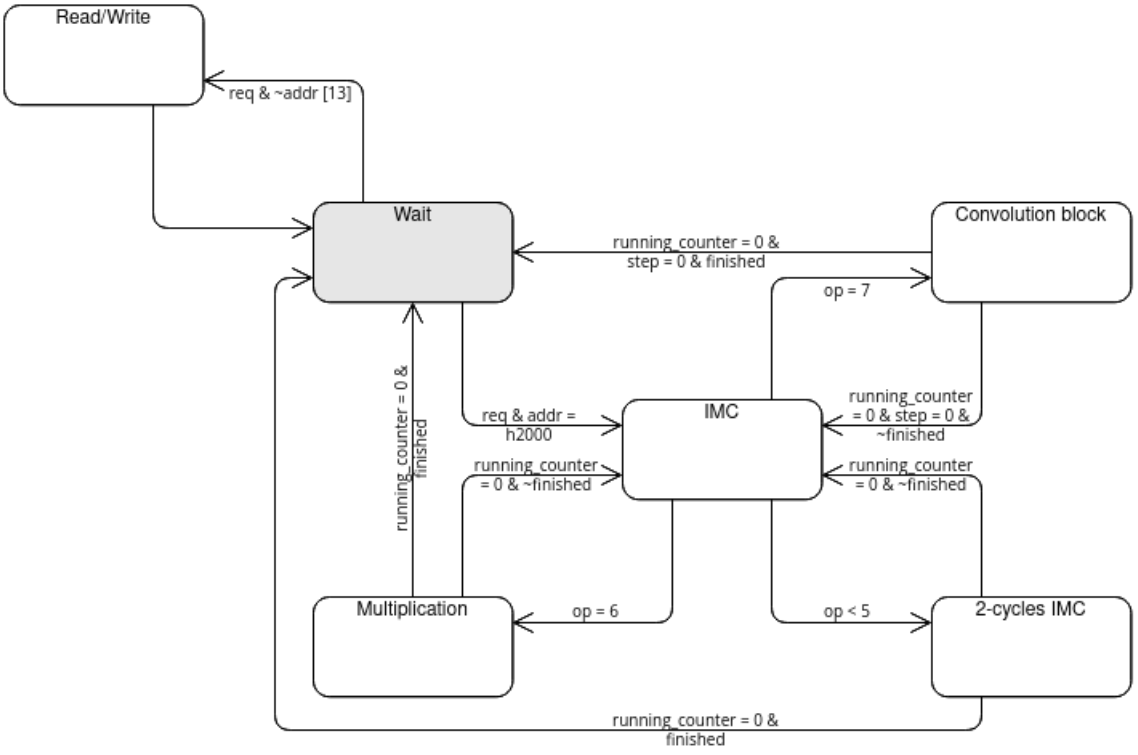

Figure 3.5: FSM of the new controller with hardware loops

The objective being to loose less clock cycle as possible, this state can be skipped when the hardware loop registers are kept to '0'. The FSM looks then more like the first one. Concerning the *Read/Write* state, it is the only one that will never be impacted by the hardware loop since it is not expected to do some loops for these operations.

3.2 RTL simulations

In the process of integrating BLADE on a new chip, the large majority of the work is to simulate the design and verify that all the functionality expected are well implemented and functional. In order to verify that everything was correctly working with the new controller 2 different kind of simulations were run in which all the blocks were tested. During the design of the controller, most of the simulations were done with System Verilog testbenches. Indeed, these testbenches have the big advantage to enable to control all the input signal of the block tested. A very demanding work have been then to create testbenches for the controller file that were setting the input signal. In order to debug the

RTL code, a lot of time have been spent to check all the internal signals of the controller and to see if all the waves were well coordinated. This part being simulated without the memory SAs, it is not possible to know if the instruction required were leading to the good result, but at least it allowed to verify that the global features were running correctly. In these simulations, for instance, the number of cycles of the multiplication was checked along with the hardware loop. It was useful to debug roughly the most important problems in order to be able to run more complex simulations later.
Once the first simulations with only the controller file were done, a testbench for the toplevel have been written. This System Verilog testbench was the first one to enable testing the entire BLADE peripheral, it was testing at the same time all the blocks of Figure 3.3. Only the physical macro was not tested, but a Sytem Verilog file simulating the behaviour of the memory was used in order to simplify simulation. In this part of the simulations, it was interesting to test a maximum of IMCs in order to be sure that all the timing between the waves were correct. For example, the result of a read must be available when the *r_ valid* signal is high. These are the kind of checking that were done. A list of operation to be checked is drawn up:

- Write the entire memory.
- Read the entire memory.
- Perform a single 2-cycles IMC.
- Perform 2-cycles IMCs using *length* and *stride* to loop operations.
- Perform a single multiplication.
- Perform multiplications using *length* and *stride* to loop operations.
- Perform a single convolution block.
- Perform convolution blocks using *length* and *stride* to loop operations.

Once most of the functionalities were verified with the testbenches, the new architecture of BLADE have been integrated inside HEEPpocrates to be able to run more complex tests such as convolutions. These applications were written in C for easiness. It was a much more efficient way to test the features of the new controller and to be sure that all the results were correct. The possibility to use *printf* on HEEPpocrates have eased the verification avoiding to check systematically the waves. Figure 3.6 and 3.7 show respectively the wave form for a single multiplication and for an hardware loop for an AND operation.

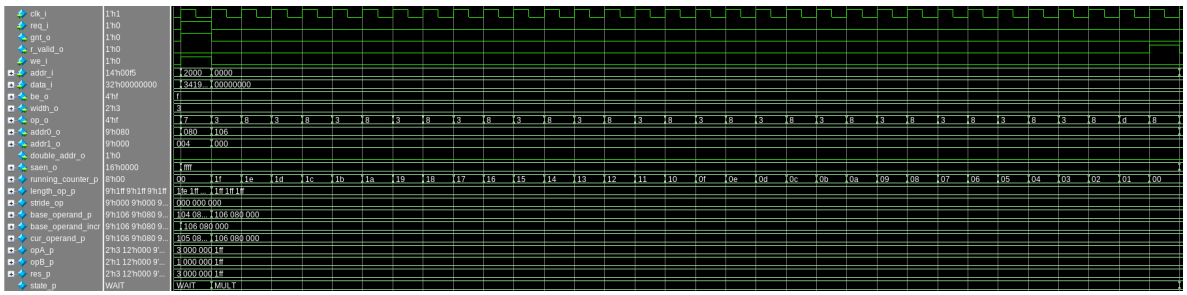

Figure 3.6: Single multiplication with the new controller on HEEPpocrates

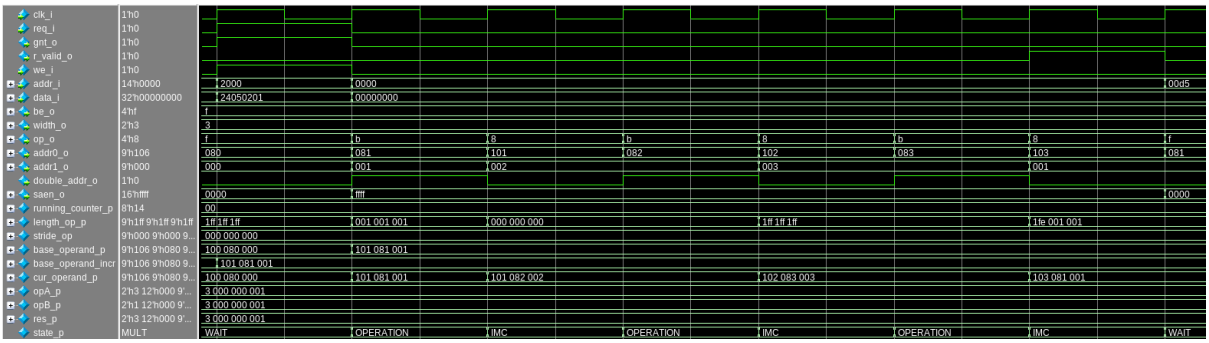

Figure 3.7: Hardware loop on an AND logic operation with the new controller on HEEPpocrates

A 16-bits convolution test have been run to verify the new features added on the new controller. This simulation have highlighted the benefits of the new design.

3.3 Benefits of the new design

This new controller aimed to reduce significantly the instruction time. This objective have been reached by finding a new way to interface the bus to the controller of BLADE, adding state registers that can be written from the CPU and read at any time by the controller. Tables 3.1 and 3.2 show respectively the number of clock cycles per operation result with the previous an the new controller. These tests have been made with testbenches and are just simulating BLADE peripheral before its integration on HEEPpocrates. The 2-cycle IMCs represent the operations that are performed in 2 cycles (all the operations except reads, writes and multiplications). First thing to be noticed is that the throughput of BLADE always increases with the bit-length of the operands thanks to the parallelization of operations. However, the improvement is better with the multiplication than the other operations due to the reduction of the operation time. Then , it is clear that the new controller improve the operation throughput in any case, even though the progress is more important for the 2-cycle IMCs (-79% of clock cycles per operation). Indeed, the proportion of the instruction time is more important for operations that have short computation time. Thus, the reduction of instruction cycles greatly improves the overall operation efficiency.

Cycles per operation	2-cycle IMCs	Multiplication
32 bits	0.56	4.50
16 bits	0.28	1.25
8 bits	0.14	0.38

Table 3.1: Cycles per operation with the old controller

Cycles per operation	2-cycle IMCs	Multiplication
32 bits	0.13	4.06
16 bits	0.06	1.03
8 bits	0.03	0.27

Table 3.2: Cycles per operation with the new controller

As most of the effort in this master thesis have been done to improve the controller in the view of running convolution on BLADE, a 16-bits convolution simulation have been run on HEEPpocrates. This application aimed to put in context all the modifications that had been implemented to speed up computations. Figure 3.8 presents the new

data placement inside the subarrays. This placement differs from the one on Figure 2.8 in order to be able to use the hardware loops. Indeed, previously they were not useful since the data addresses of the input were not continuous. By adding a wrapper around the subarrays, the addresses are now increasing in the same way than input indices. For the same reason result addressing are following to be able to access them continuously. Concerning the placement of the kernels they did not change compared to the last application, the only constraint about the filters is to have the 25 indices of each filter continuous and it was already possible with the previous addressing of memory.

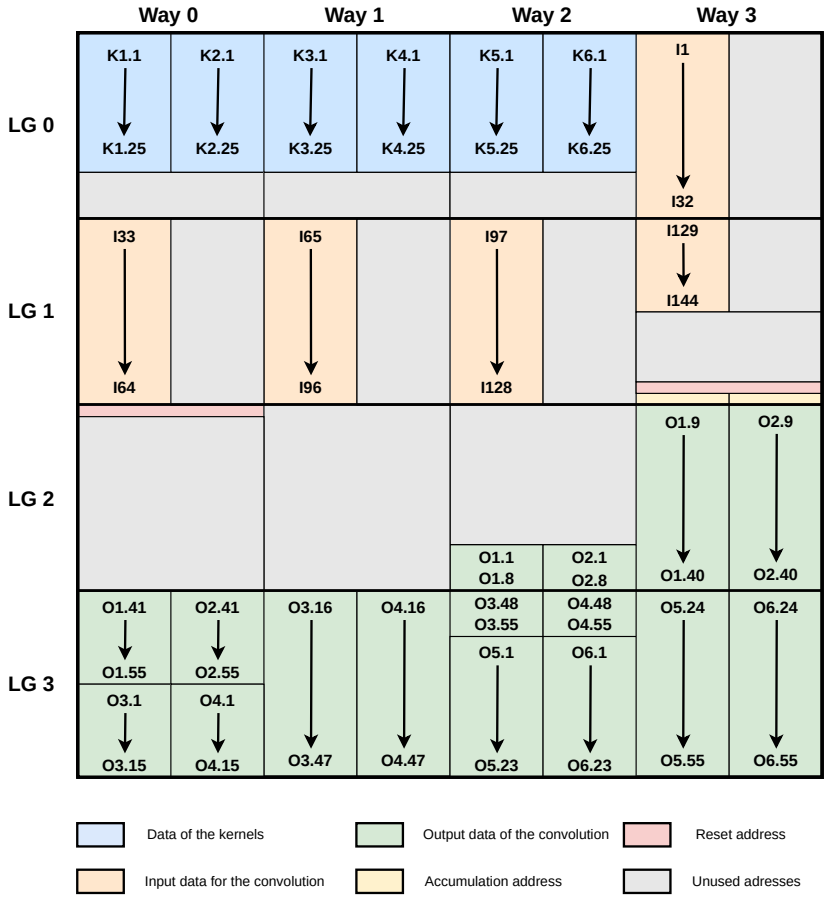

Figure 3.8: Data placement within the SAs for a 16-bits convolution

The result of the 16-bits convolution with the new controller design have been presented on the tables below. Table 3.3 shows the distribution of clock cycles during the data placement. These results consider only the placement of the input and the filters are already considered placed in the memory since they should not be changed between 2 convolutions. They would then be placed only once. Here it is clear that the data placement is limited by the CPU speed to send the instructions. This is a factor that could be probably eliminated in the future by implementing a DMA (Direct Memory Access) that would be able to place the input data in real time while the CPU would be in sleep mode. As the sensor would be measuring and sending data the DMA would be placing them inside the memory. The data would be already placed inside the memory at the end of the sensing phase. Thus, all the clock cycles that are used to place data

inside BLADE could be neglected. Table 3.4 presents the distribution of clock cycles to perform the convolution computation. This part is the one that have been optimized the most during this master thesis. As it can be seen on the table, the number of instruction is reduced to 208 which is only a tiny part of the time used for the computations. With the addition of the *convolution block* operation and the use of the hardware loops it is now possible to send one instruction that take one clock cycle to do 75 operations in a row. This is a huge improvement compared to the previous controller design that used 7 clock cycles to send 4 instructions to do only one operation in the end. The time of instruction have been in the end divided by 420. On the top this enhancement, the CPU working time also reduces very significantly. Indeed, since the instructions are very few, the communication between BLADE and the CPU are drastically decreasing. BLADE memory is now able to compute 97.3% of the time which is an important contribution of this work. The distribution of clock cycles during the result reading is depicted on Table 3.5. As for the writing of the data, it is obvious that this phase is limited by the CPU working time. Once again, implementing a DMA would enable to automate this work and would greatly reduce the time of this part. The CPU could stay in sleep mode and the work would be almost reduced to the instruction time and BLADE computations time. This a thing that should be thought in order to reduce at its minimum the CPU overhead.

Number of clock cycles	BLADE
Instructions	2 016 (3.4%)
CPU computations	55 255 (93.2%)
BLADE Computations	2 016 (3.4%)
Total	59 287

Table 3.3: Number of clock cycles to place data inside BLADE

Number of clock cycles	BLADE
Instructions	208 (0.1%)
CPU computations	4 140 (2.6%)
BLADE Computations	155 775 (97.3%)
Total	160 123

Table 3.4: Number of clock cycles to perform convolution inside BLADE

Number of clock cycles	BLADE
Instructions	4 704 (4.3%)
CPU computations	99 893 (91.4%)
BLADE Computations	4 704 (4.3%)
Total	109 301

Table 3.5: Number of clock cycles to read the results inside BLADE

Figure 3.9 shows the comparison of the time taken to do the 16-bits convolution with the old and the new controller design using a RISC-V core CPU. The time to do the convolution have been reduced by 49.6% which is a great improvement knowing that the computation time inside BLADE is not changed. This enhancement could even reach 74.0% of time reduction if the DMA was implemented. This implementation will be done in the continuity of this work.

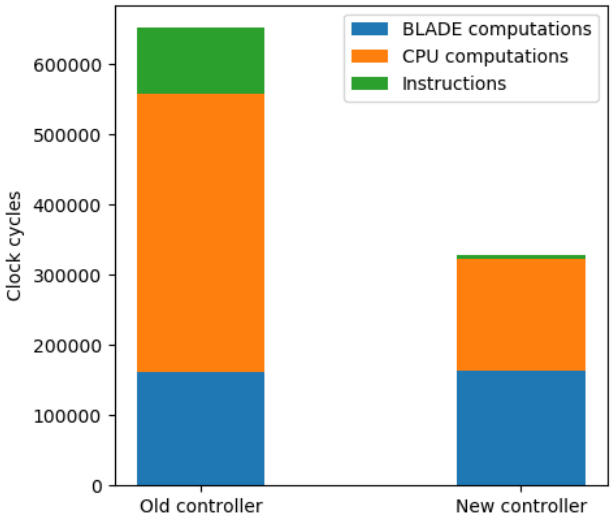

Figure 3.9: Comparison of the 16-bits convolution with the old and new controller design

A last comparison have been made to demonstrate the efficiency of BLADE to perform convolution for CNNs by comparing the time to do the 16-bits convolution on the CPU of HEEPpocrates with the time necessary to do the exact same operation on the new design of BLADE. The results of this experiment have been plotted on Figure 3.10. This figure clearly shows that BLADE can act as an accelerator for convolution on HEEPpocrates, leading to a 75.0% of the time reduction for a convolution. This time reduction could also be improved until 87.1% by implementing the DMA to transfer data in the future the future iteration of BLADE.

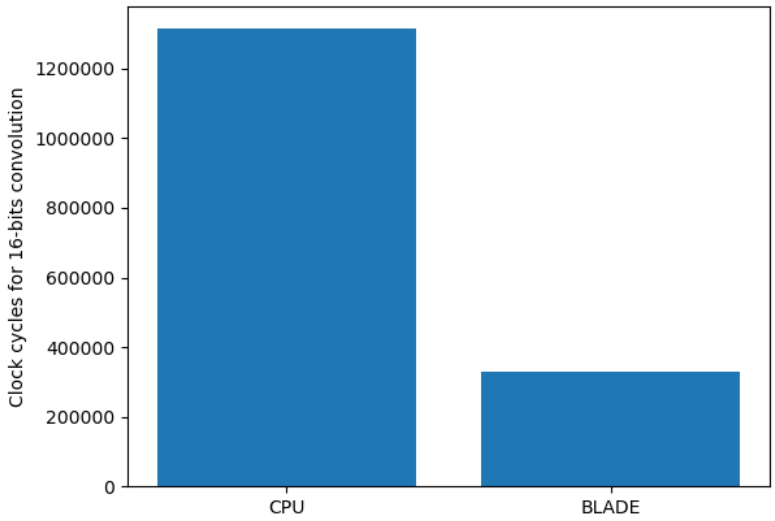

Figure 3.10: Comparison of the 16-bits convolution between the CPU of HEEPpocrates and BLADE integrated inside HEEPpocrates

Conclusion

In the context of this master thesis, that was targeting a new tape-out in TSMC 65nm technology on which BLADE would be integrated as an accelerator for CNNs, a lot of work have been effected.

As a starting point, the old design of BLADE have been studied and and simulated in order to understand the way it was operating. These simulations have led to many optimizations in the code writing, trying to write efficient C code or to write assembly in order to keep control on the compilation of the code. Python functions have been written to auto-generate the C code to be compiled. These software optimizations have enabled to determine the hardware limitation to be improved in the new controller. A way to efficiently make convolution on BLADE have been designed for 32-bits and 16-bits data bit-widths. These convolutions have been compared and have shown that the reduction of the data bit-width have a significant impact on the convolution time. The bottlenecks of the old controller have been identified in order to solve them in the new design.

In a second time, a new design architecture have been thought for the new controller of BLADE in order to improve the time dedicated to instructions. The insertion of state registers have enabled to write and read BLADE as any other memory, leading to a better integration in the standard architecture of processor platform. These states registers have also greatly decreased the time spent to send instructions between the CPU and the controller. These optimization of hardware have enabled significant improvement of operations inside BLADE.

Finally, additional implementations have enabled to accelerate two times the convolutions on BLADE. This enhancement is possible thanks to the new addressing of the macro that have been done by adding a wrapper around the SAs and also thanks to the *convolution block* that have been integrated in the controller FSM. These improvements have enabled to reduce drastically the time used for instructions, increasing consequently the computation of BLADE time proportion. These changes have led to important result for BLADE showing that it could be very efficient for convolutions compared to CPUs.

Glossary

ALU Arithmetic Logic Unit. 3, 21

BLADE BitLine Accelerator for Devices on the Edge. 2–7, 9–11, 16–28, 30–32

CNN Convolution Neural Network. 3, 11, 13, 16, 17, 21, 31, 32

CPU Central Process Unit. 2, 3, 5, 6, 9, 10, 16, 17, 19, 22, 28–32

DMA Direct Memory Access. 29–31

FSM Finite State Machine. 24–26, 32

GBL Global BitLine. 4

GPU Graphic Process Unit. 2

HDL Hardware Description Language. 24

HEEP Healthcare Energy Efficient Platform. 3, 23

IMC In-Memory Computing. 2, 4–7, 9–12, 14–16, 21, 23, 25, 27, 28

LG Local Group. 4, 12, 15, 16, 18, 21, 24

LGP Local Group Periphery. 4

LSB Less Significant Bits. 18, 19

MSB Most Significant Bits. 7, 18, 19

RTL Register Transfer Level. 26, 27

SA SubArray. 4, 7, 11, 12, 14–21, 24, 27, 29, 32

SRAM Static Random Access Memory. 2

References

[1] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. "Edge machine learning for ai-enabled iot devices: A review". In: *Sensors* 20.9 (2020), p. 2533.

[2] Maha Kooli et al. "Smart instruction codes for in-memory computing architectures compatible with standard SRAM interfaces". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1634–1639. DOI: 10.23919/DATE.2018.8342276.

[3] R. Gauchi et al. "Memory Sizing of a Scalable SRAM In-Memory Computing Tile Based Architecture". In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. 2019, pp. 166–171. DOI: 10.1109/VLSI-SoC.2019.8920373.

[4] William Andrew Simon et al. "BLADE: An in-Cache Computing Architecture for Edge Devices". In: *IEEE Transactions on Computers* 69.9 (2020), pp. 1349–1363. DOI: 10.1109/TC.2020.2972528.

[5] *Rosetta (2019)*. URL: http://asic.ethz.ch/2019/Rosetta.html. (accessed: 27.07.2022).

[6] *Darkside (2021)*. URL: http://asic.ethz.ch/2021/Darkside.html. (accessed: 27.07.2022).

[7] Marco Antonio Rios et al. "Associativity-agnostic in-cache computing memory architecture optimized for multiplication". In: (2021). URL: http://infoscience.epfl.ch/record/290423.

[8] William Andrew SIMON. *Blade controller documentation*. ESL, 2022.

[9] *Ibex documentation*. URL: https://ibex-core.readthedocs.io/en/latest/03_reference/load_store_unit.html. (accessed: 27.07.2022).

[10] Marco Rios et al. "Error Resilient In-Memory Computing Architecture for CNN Inference on the Edge". In: *Proceedings of the Great Lakes Symposium on VLSI 2022*. GLSVLSI '22. Irvine, CA, USA: Association for Computing Machinery, 2022, pp. 249–254. ISBN: 9781450393225. DOI: 10.1145/3526241.3530351. URL: https://doi.org/10.1145/3526241.3530351.