POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Design of a Novel Precision Scalable Multiplier to Improve Quantized Neural Network Computation on a Low-Power RISC-V Processor

Advisors
prof. Mario Roberto Casu
dott. Luca Urbinati

Candidate
Edward Manca

Academic Year 2022-2023

Summary

In a world in which the amount of computation required is growing fast, and where one of the most computation-hungry task is the execution of Machine Learning (ML) algorithms, especially Neural Networks (NN) computation, the research started focusing on accelerating and optimizing for these tasks [1].

Among different possible approaches, leading to several trade-offs, varying from specialized ASIC accelerators to framework-level optimization, this thesis focuses on implementing a Precision Scalable (PS) multiplier to be placed inside the pipeline of the Ibex low-power RISC-V processing core [25]. PS multipliers are a research topic since some time now [15, 16, 17]; these are multipliers capable of executing more than one multiplication in parallel in the case these operations are executed at a lower precision than the maximum one the multiplier can support. Target for the PS multiplier architecture developed in this thesis is the acceleration of Quantized Neural Networks (QNN) computation. QNNs are an optimization of standard NNs to ease the deployment specifically, but not only, on low-power performance-constrained devices, which represent the typical application target of the Ibex core. QNNs make use only of integer operations with a variable precision selected on a case-by-case basis after a proper analysis that aims to strike the right balance in the trade-off between NN accuracy and performance [5, 6].

First, starting from the standard Baugh-Wooley (BW) multiplier architecture, in this thesis I derived a novel PS multiplier capable of changing the precision from 4 to 16 bit. Parallel execution at reduced precision involves two types of operations: Sum-Separate (SS), which consists in multiplying and returning the results of each multiplication independently, and Sum-Together (ST), where the multiplication results are also added before returning [18]. Moreover, I added the support for the Multiply and Accumulate (MAC) class of instructions to the Ibex core for all multiplication instructions, thus including SS and ST operations previously mentioned. All these instructions are suitable in different contexts inside the QNN acceleration domain, considering optimal execution differences in the most common ML algorithms for the edge scenario, which is the subject of the subsequent benchmark analysis.

On the basis of the two standard multiplier units originally present in the Ibex core, I defined two conceptually similar structures including in the design two gate-level multipliers based on the novel PS BW structure previously mentioned, which has been adapted to this context. For comparison, I also developed two purely behavioral descriptions, in System Verilog, for which synthesis tools are able to derive the most appropriate multiplier structure capable of dealing with the PS and parallel execution requirements. In total, configurations of all multiplier structures that I designed include the possibility of executing a single 32 bit multiplication, two 16 bit multiplications, four 8 bit multiplications, and eight 4 bit multiplications in parallel, all of these supporting both the SS and the ST modes, and also MAC operations.

These previously mentioned configurations define a set of 31 new custom instructions to be added to the RISC-V ISA. To support new instructions I also extended the decode unit of the Ibex core. Added instructions are thus in the category of MUL and MAC classes in the range from 32 bit (equal to the original design of the Ibex multiplier unit) involving one single operation, down to 4 bit for which, potentially, eight operations are replaced with a single one executing them in parallel.

In order to compile high level C/C++ code, support for these instructions must be added in a compiler. In this case, I modified the well-known GCC compiler to accept the assembly format of those instructions, allowing to include, and easily compile, inline custom assembly instructions inside C/C++ source code.

To evaluate the performance of the modified processor with respect to the original design, I developed a set of three benchmarks for the three most common QNN algorithms in the edge scenario. Each benchmark targets a single ML algorithm, namely, Fully Connected layer, Convolutional layer, and Depth-Wise Convolutional layer. For each of these, I made clear the performance advantages coming from the usage of reduced precision instructions exploiting, as a metric, the execution time, through a precise analysis of each modified Ibex core version running on an FPGA board.

Finally, I compared all previously mentioned Ibex core variations, targeting synthesis on silicon on a 28 nm production process, and analyzing power, performance, and area metrics to derive more comprehensive conclusions.

Acknowledgements

First, I want to make use of this section to express my gratitude toward my family for their support during all years of my study path. In the end, for me, they always desired the best.
Then, I want to acknowledge my friends, Mirko, Giovanni, Marco and Nicola, that along my university career made possible a special mix of common interests in studying, having fun, sharing opinions, thoughts, and experiences.
Moreover, I want to thank Luca Urbinati, not just for his support, but in particular for having shared his experience about his PhD path with an enthusiasm that ended up being contagious.
Last but certainly not least, I want to thank Prof. Mario R. Casu especially for the trust he gave me from this thesis proposal on, and for his effort at my personal growth path in embarking on the scientific research.

Contents

Chapter 1

Introduction

1.1 The importance of Machine Learning

From the time ML became a hot topic, widely diffused in several different fields, one thesis out of two includes a long in-depth introduction analysis, making ML algorithms and NN structures a well known argument over past years. Given that, I do not see the point of repeating it, thus I have chosen not to dedicate an independent chapter to an in-depth ML algorithm listing and description. I place instead here some references to books or scientific surveys where all concept description is well done. As for two books covering almost every aspect of theoretical ML algorithms refer to [2] and [3], for a quick survey see [4], for instead a focus on quantization techniques and relative issues [5, 6]. Of course, single ML algorithms and NN structures are tackled down individually when needed along chapters in this document. As an introduction, I want instead to focus on the importance of ML in today's world, with an emphasis on the importance of accelerating ML algorithms.
A way of seeing ML algorithms is the one of seeing them as a class of well-optimized algorithms able to approximate underlying functions in our world. Starting from a set of relevant data, a ML algorithm is able to detect, while approximating, relationships between data (if they exist). So, the final objective is to develop the mathematical definition of a function approximator, including computation done over the data, internal tunable parameters to approximate functions, and a learning procedure, usually referred as *training phase*, to update those internal parameters to fit that wanted function.

From this description we can derive 3 distinct processes:

- Data collection
- Algorithm structure definition
- Computation

These interact together within the definition procedure, but, once defined, they could be treated as conceptually separated.

Data collection is the process of crating collections of data in the most informative and suitable manner, also data format must be compliant with the algorithm and systems that carries out the computation. For example, considering humans, we acquire data from our own "sensors" (i.e. eyes, ears, nose, etc.) and we develop our own functions (at least the practical ones) starting from these data. But in a computer-centric world, it becomes evident this process is related to other types of sensors, and this influence not only data format but also data acquisition systems, that obviously relapse in the electronic and computer engineering fields.

Algorithm structure definition is the process of creating a mathematical definition of computation to be subsequently done over desired, relevant data. For this definition, two strategies have been derived over the years, a formal mathematical description able to consider theoretical aspect of function approximation (still it is possible to see, in a way, polynomial regression as a ML algorithm), or let the human intuition join mathematical aspects and do the job. Especially this last strategy have been, during last years, able to provide us most of the fancy architectures nowadays are considered the state-of-the-art for specific task accuracy, conceiving things from the residual connections in deep CNN structures [7] to that architectural monster that is the *Transformer block* [8] used for sequential data analysis.

Computation is intended as systems able to compute, possibly efficiently, that mathematical description with the input data collected, and allow all the above to exist in the real world. These systems are definitely the limit to the applicability of such algorithms. There are several reasons:

- **Technological:** we still rely on transistors for computation, and this will be also true for a long time in the future, thus the computation limit increases at the same time the transistor description and fabrication processes become more efficient (i.e. the integration density increases, single transistors require less power, parasitic effects are minimized, etc.).
- **Architectural:** from the most low-level machine description and organization (i.e. from classical von Neumann architectures, to memory-centric architectures [23]), to the most high-level system-design (i.e. exploit existing parallelism in algorithms to split and parallelize the computation, the local memory sizing procedures, etc.).
- **Algorithmic:** from the code organization and memory hierarchy management strategies (e.g. changing code structure to fit memory hierarchy of target devices) passing through optimization of code for specific machine architectures

(e.g. development platform as Nvidia CUDA [33], Intel OpenVino framework [34] or AMD ROCm [35]) coming to ML algorithm description and computation frameworks like TensorFlow [36] or Pytotch [37].

So, about the importance of ML in today's world, we can see every practical interaction between us and our world can be treated as a function. We know, from some decades now, we can not limit computation of such interactions to human entity, that is limited, even if very efficient. Nowadays, we discovered we can not limit also the development of these interactions to human entity. This is when ML techniques became essential, as it became essential the automatic treatment (i.e. learning) of information present in the data itself.

Indeed, the explosion of ML algorithms in different fields like the healthcare (specifically in biologic research), the industry (to optimally organize facilities), the automotive (leading to autonomous driving systems), and many more, all of which with outstanding results, is the confirmation that human-made algorithms are not a first choice for some specific tasks, or at least, have a clear slower development with respect to the ML approach.

The importance of machine learning now becomes clear in the development of our society, and, given the limit of that development, and deployment, is the computation capability, it becomes clear the importance of the machine learning acceleration.

Starting from the next section and for all the other chapters of this document, the focus is only on NNs and its derivations.

1.2 Neural Networks at the Edge: Optimization and Compromises

Continuing the discussion on giving some hints about the importance of ML algorithm acceleration, if we consider one hot topic in these days, one also well known to the average man, that is, ChatGPT, everyone can clearly see the possibilities offered by these systems. ChatGPT is a chat mechanism based on a Large Language Model (LLM) developed by OpenAI [38], when it has been launched it was based on the GPT-3 LLM, developed by the same group. By looking at a GPT-3 paper details [13], we can easily discover that these impressive results are reached with models that could need an amount of parameters in the order of hundreds of billions of parameters, so, considering 32 bit per parameter, a memory quantity in the order of hundreds of GB of RAM [13], even in the order of TB at most, and few systems in the world can manage these requirements. Now, the deployment and optimization concerns become clear. Of course ML applications could be everywhere, and not all NNs requires that amount of parameters, especially considering optimization for mobile devices. We can think dozens of application for which data

are generated at the edge, and could easily being processed in the same position without having necessity to send them to some server in the world able to deal with billions of parameters and even more floating point operations. Some examples are:

- Human activity recognition and tracking
- Anomaly detection for critical safety systems
- Predictive maintenance or quality control for industry
- Facial recognition in airports
- Traffic prediction in cities

But in order to spread more sophisticated NNs along class of devices accessible to everyone, especially those devices everyone is using even if they don't know they are, optimization techniques of NNs must be considered.

When talking about this approach we talk about "ML at the edge", where the edge could be imagined as the near jump between the computing system and the real world we also can interact with. These devices are usually low power, with little computation capabilities, little memory, and most important they must be cheap as they need to be widely diffused. The following are some of the most common ML optimizations techniques for this class of devices:

- **Reducing model parameters**
  This can be achieved by using fewer parameters in the model definition or by pruning [11], that is the operation of removing both some less meaningful input connections or entire neurons if that output result is not used. Both of these strategies could lead to a reduction in the NN accuracy, usually since pruning removes parameters after the training step done on a larger model, the reduction in the accuracy is lower than the same structure with fewer parameters at the beginning of the training.

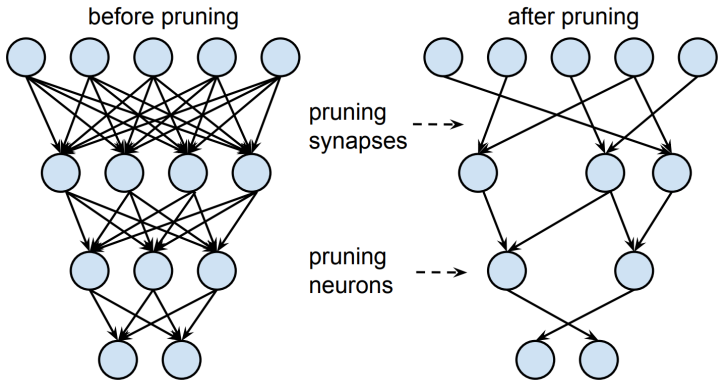


Figure 1.1: Network structure before and after pruning process [11].

- **Parameters quantization**
  NNs are one of the most over-parametrized application, the functional space of these systems is huge, but it is required to find a good parameter-set to approximate the underlying function of the desired application in a relatively small amount of time compared to the whole space exploration. Still this space is so large that, even if each parameter is quantized, it is likely it would approximate the underlying function in an acceptable way. Some techniques have been developed along the years to adjust parameters over a trained NN, or to train a NN considering at the same time the final quantization required. For the first we talk about *Post Training Quantization*, while for the latter we refer to as *Quantization Aware Training*. Amid quantization techniques, two more common techniques are *affine quantization* and *scale quantization* [5, 6]. Scale quantization, as the name says, scales floating point parameters equally distributing them to the range that could be represented by the selected integer precision. Accuracy of the integer quantization depends not only on integer precision, but also on how wide is the range of such parameters, and how near these are centered to the zero position. Affine quantization still performs the same scaling procedure as scale quantization, but also it makes the quantization affine to the center position where parameters are distributed subtracting the mean of these points before quantizing.

- **Winograd transformation**
  The Winograd transformation is a well known technique coming from signal processing domain. The same transformation is also used, and suitable, considering operations involved in convolutional layers typically present in NN (CNN), where a small size filter (smaller with respect to the input dimension) is convoluted along the input producing at each step a single output of the output matrix. Winograd transformation says, to be practical, that considering small size filters with an overlap at each stage, it exists a larger filter that, if applied in the same positions, in conjunction with a preprocessed version of the input, would produce the same output in less computation. This technique adds a trade-off, it allows reducing computation if we accept to pay the price of a higher number of parameters of the new bigger filter, instead of the smaller original one. Moreover, for small quantization bit-width it could be less beneficial if not considering paying also the price of increase the quantization bit-width for some layers. An analysis on pro-cons considering application of Winograd transformation on CNN can be found in the following paper [12].

- **HW support for reduced precision operation**
  Quantizing without having HW support to compute reduced precision operations is half an advantage, since, even if memory footprint is reduced, performance (translating in inference latency) and power consumption (translating in

target applications) remain the same. An analysis of computational requirements for the three most common layers, specifically considering low-power applications, follows:

- **Convolutional Neural Networks** (CNN) layers perform multiplications and accumulations of each input-parameter couple, for each output position, which determines also the filter position in the input feature map along the convolution process. For this layer the main, if not the only strictly, operation required is the MAC operation.
- **Depth-Wise Convolutional Neural Networks** (DW-CNN) are a variation of classical convolutional networks, which have been introduced in MobileNets [9] within the *Separable DW-CNN*, it aims in reducing model parameters and computational requirements, while trying preserving original expressiveness of convolutional layers. Still, being a convolutional-based layer, the main operation required is the MAC operation.
- **Fully Connected** (FC) layers (also known as Dense, or Single Layer Perceptron) were the basis of neural inspired networks at the down of the artificial intelligence research. In a fully connected layer all inputs are multiplied with corresponding parameters. The parameter vector has the same dimension of the input one. This means every output value depends on every input value, thus the number of operations is even higher. It is easy to make this layer over-parametrized in most applications, thus, most of the time, its main use is as the very last layer of NNs. Also in this case, the main operation is a MAC operation.

Given that the main operation is the **MAC** operation for almost every computational layer, and that other layers, either do not require any parameter, or can be folded inside the previous computational layers, as per the case of Batch Normalization layers [10], the focus of HW-acceleration is in optimizing the MAC operation.
Efficient HW support for reduced precision operations is an open challenge. An approach could be modifying multiplier structures that are prone to be parallelized, examples are [18] [21], but there are cases in which the amount of added logic is not negligible anymore, making a multiplier structure suitable for very limited applications. Another way is using several simpler and smaller low-precision multipliers to achieve the same computational capabilities in accelerators [22] or in dedicated units [19], but in this case the limit is at the interconnection and algorithm mapping level. The same approach could be used in FPGA, which started increasing the amount of internal DSP that exploit FPGA reconfigurable interconnections [24].
Technology leaders implement these solutions inside commercial product for different target markets. For example Google *TPU* [39] is an HW specifically

designed for ML acceleration, the main target is the cloud execution domain. The same TPU concept have been shrunk to target the edge devices, producing the *edge-TPU* [40], an architecture targeting the low-power domain. Nvidia developed the *Tensor Core* integrated in many solutions from datacenter GPU like *A100* [32], to consumer level GPU like the RTX 3xxx family, down to the embedded market with the *Jetson Orin* [41]. Nvidia developed also the *NVDLA* open-source accelerator [42]. AMD Xilinx is proposing an easier approach to accelerate NNs exploiting FPGA reconfigure capabilities with the *FINN* toolchain [43]. In the same direction also the open-source *HSL4ML* allow targeting FPGA to accelerate NNs [44].

The last optimization strategy, specifically the design of precision scalable HW for the edge scenario, is the focus of this thesis work.

1.3 Introduction to thesis work

As could be previously foreseen, in the research, there are several possible approaches to accelerate ML algorithms, given several optimization levels and performance target:

- **ASIC** is the most efficient solution since it targets just a restricted class of algorithms, but at the same time the flexibility is minimum as it is not able to carry out standard computation workload like the execution of general purpose programs. This is usually a good approach for statically designed accelerators.
- **FPGA** is another solution, in this case less efficient but more flexible, algorithms can be mapped through FPGA LUT while exploiting programmable interconnections to implement necessary binary functions. Nowadays, many FPGA designs dedicate a significant amount of area to implement more general DSP blocks able to compute operations that most of the time are useful also for ML algorithms, in this case the FPGA configurable structure is mostly dedicated to the programmable interconnections between those DPS blocks. A good example of tailoring an FPGA configuration to class of NNs or even specific NN structures is presented in [28].
- **GPGPU** is widely used to compute ML algorithms given the capability to perform easily parallel tasks coming from a structure of an order of some thousands of little specialized blocks organized in an hierarchical way, on which, e.g., computation involving a set of data in local SRAM can be easily mapped to the one required by ML algorithms, especially CNN computation.
- **CPU** is conceived primarily to perform sequential workloads and to guarantee the possibility to execute code compliant with a given ISA, usually these

architectures are necessary in almost every modern system to compute the control part of a given algorithm, for which the computation can be performed by dedicated accelerators like, for example, the ones previously listed. Often these machines could be as small as possible, an example can be seen in the M-series of ARM processors [46] used in small microcontrollers in which the area devoted to the computation must be extremely smaller to keep the cost as low as possible. However, most of the time this type of CPUs are the only computation capable devices present in a system, and if unit cost is one of the key element in the design, in order to accelerate other tasks, it is necessary to modify or integrate the minimum amount of computational capability, thus the minimum amount of silicon, to meet the requirements. Not only small devices includes CPU strategies to increase parallel computation capabilities, also server-class CPU started including set of instructions devoted to vector operations, e.g. AVX extensions proposed by Intel.

This last consideration leads to another way to accelerate ML computation, that is modifying the processor structure, adding new instructions (also, depending on the target application, removing unused instructions) and HW logic capable of computing those instructions. The result of this strategy is referred as **Application Specific Instruction-set Processors (ASIP)**. This thesis work focus on the applicability of a novel multiplier architecture in the context of modifications on a microprocessor core conceived for low-power scenarios. The microprocessor chosen is the fully available and open source [48] 32 bit RISC-V Ibex core, developed jointly by ETH Zurich and University of Bologna [49] in which 31 new instructions are added. These instructions are aimed in adding support to perform, in parallel, more than one multiplication (MUL operation) if these are at a reduced precision with respect to standard register bit-width of the processor, such as: two operations if operands and result are at 16 bit precision, four operations for the 8 bit case, and eight operations for the 4 bit case. Parallel execution involves two types of operations, Sum-Separate (SS), which returns independent results, and Sum-Together (ST), which sum the results before returning. Moreover, another group of instructions adds support for MAC operations for all previously mentioned multiplications. MAC operations accumulate the result of these parallel execution over time exploiting an internal register. In all cases, including standard MUL operations present in the unmodified RISC-V ISA, computation is carried out by a single in-pipeline multiplier unit compliant with the Ibex architecture for which several trade-offs are considered among possible implementations as treated in next chapters. An assembly support for all these instructions is then added to the GCC compiler with the objective to allow compilation of assembly instructions present in C/C++ code. Developed benchmarks, to retrieve performance improvements, make use of this modified GCC compiler targeting three, among the most common, layers in the QNN edge domain, that are, Fully Connected (FC) layer, Convolutional NN (CNN) layer, and Depth-Wise CNN (DW-CNN) layer.

1.4 Related work

As already mentioned, the research started focusing on the acceleration of NNs, specifically QNNs, from some time now. Related to the QNN acceleration domain, several PS multipliers have been recently proposed [17]. In [21] a radix-4 Booth ST multiplier have been realized to accelerate ML tasks. In [19] is proposed an approach based on several smaller architectures as *divide-and-conquer* strategy. Also, on the ST operations side, in [22] an accelerator for 2D-CNNs is proposed. Specifically on the Baugh-Wooley PS structures, [18] proposed an analysis on the usage of the Baugh-Wooley to target PS execution for SS or ST operations, realizing then two separated FC accelerators.

On the side of integrating dedicated HW units to accelerate specific tasks at microprocessor level, in [26] a Single-Instruction Multiple-Data (SIMD) dot-product unit have been developed, then integrated inside a RISC-V processor, targeting the execution of NNs on IoT-class SoCs. Moreover, the RISC-V ISA is extended to support these new instructions. Finally, they tested the resulting processor in an SoC containing a cluster of eight modified Parallel Ultra-Low-Power (PULP) cores.

1.5 Thesis outline

This thesis work focuses on modifications to the Baugh-Wooley multiplier architecture to define a gate-level precision scalable variant able to compute both SS and ST operations in the same structure.

Subsequently, this multiplier structure is adapted to implement two gate-level multiplier units integrating them in the pipeline of the RISC-V Ibex processor with the objective to carry out computation typically required by QNNs. Moreover, a behavioral description of these precision scalable multipliers is also defined, for comparison.

A class of 31 new instructions is defined and added to the RISC-V ISA. In order to compile code routines based on these instructions, the GCC compiler is extended.

Finally, to test performance execution, three different benchmarks based on the most common QNN layer algorithms in the edge computing domain have been developed.

Chapter organization is as follows:

Chapter 2 introduces the Baugh-Wooley structure and the two main techniques to implement parallelism for lower bit-width operations than the maximum one the multiplier structure can support, that are, *Sum-Separate* and *Sum-Together*, discussing also additions to support MAC operations. Lastly, all modifications to

the standard Baugh-Wooley architecture to implement each of the previously mentioned operations are step-by-step detailed.

Chapter 3 starts with a preamble about RISC-V ISA motivations, and describe the standard ISA. Then the Ibex processor core is introduced with an emphasis on the structure subject to the target modifications. Then the structure of both the behavioral and the gate-level PS multipliers is exposed. For the gate-level, the starting point is the set of architectural rules derived in Chapter 2. Finally, these structures, with both the two already present in the Ibex flow, have been synthesized on silicon, and a section is dedicated to expose PPA results.

Chapter 4 explains very briefly the GCC compiler and modifications done to add assembly level instructions to make them available during C/C++ compilation procedure. Moreover, a section is dedicated to an introduction to all instructions added to the RISC-V ISA, with regard to the Ibex core modifications previously proposed, concludes the chapter.

Chapter 5 presents the developed benchmarks to demonstrate performance gains over the baseline Ibex processors, and it presents the testing platform and methodology followed. The chapter concludes showing results of execution of these benchmarks deriving conclusions about performance improvements.

Finally, **Chapter 6** summarizes all the thesis work, exposing known issues, recalling results, and proposing some possible future work.

Chapter 2

Precision-scalable Baugh-Wooley multiplier

The first section of this chapter recalls the standard Baugh-Wooley multiplier structure deriving it from the classical multiplication operation.

The second section derives two class of operations for sub-word parallel multipliers starting from two possible usage of the Baugh-Wooley structure. These operations compute more than one multiplication in parallel, given that they are executed at a lower precision than the maximum one the multiplier can support.

Starting from these two class of operations, the last section presents a theoretical overview considering modifications needed to deal with all multiplications from 32 bit to 4 bit for both signed/unsigned cases, and considering the accumulation support as required by MAC operations.

2.1 Baugh-Wooley multiplier background

The Baugh-Wooley [14] is an HW structure conceived to perform multiplications between two numbers. The derivation of this kind of structure could be easily done starting from the step-by-step multiplication done by hand as follows.
For the case of two unsigned numbers, represented as:

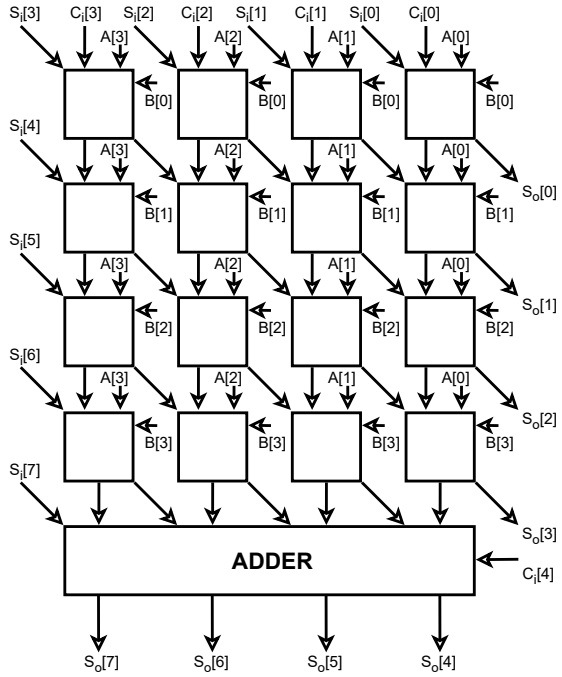
$$A = \sum_{i=0}^{n-1} a_i \cdot 2^i \tag{2.1}$$

$$B = \sum_{j=0}^{n-1} b_j \cdot 2^j \tag{2.2}$$

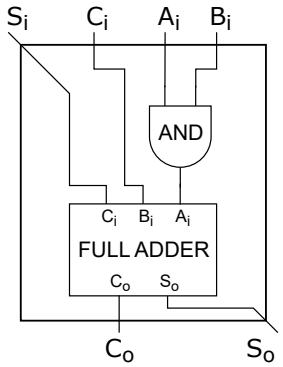
We can derive the multiplication of this two numbers as:

$$S_o = A \cdot B = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j \cdot 2^{i+j} \tag{2.3}$$

This structure can be seen as an array of progressively shifted summations of the input A multiplied, in each row, by a single bit of the input B, as presented in Figure 2.1a. Every block present in the structure requires an AND gate to compute the Partial Product (PP) of that $A_i \cdot B_i$ multiplication, and a Full Adder (FA) to add together the three bits coming in the block. An internal view of a single block is shown in Figure 2.1b. To perform just the multiplication operation all S_i and C_i bit present in the diagram 2.1a have to be set to 0.



(a) 4 bit Baugh-Wooley structure for the unsigned case.



(b) Single block.

Figure 2.1

For the case of two signed numbers, represented as:

$$A = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \tag{2.4}$$

$$B = -b_{n-1} \cdot 2^{n-1} + \sum_{j=0}^{n-2} b_j \cdot 2^j \tag{2.5}$$

The multiplication could be expanded as:

$$\begin{aligned} S_o = A \cdot B = a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} \\ - 2^{n-1} \cdot \sum_{i=0}^{n-2} b_{n-1} \cdot a_i \cdot 2^i - 2^{n-1} \cdot \sum_{j=0}^{n-2} a_{n-1} \cdot b_j \cdot 2^j \end{aligned} \tag{2.6}$$

In this case, we can see we need a sightly different structure. First, we need to correct all negative terms that cannot be directly summed along the array. To do this operation we can sum instead the two's complement (2C) of these negative terms. With the following names:

$$-x = -2^{n-1} \cdot \sum_{i=0}^{n-2} b_{n-1} \cdot a_i \cdot 2^i \tag{2.7}$$

$$x = 2C(x) = \overline{x} + 1 \tag{2.8}$$

$$-y = -2^{n-1} \cdot \sum_{j=0}^{n-2} a_{n-1} \cdot b_j \cdot 2^j \tag{2.9}$$

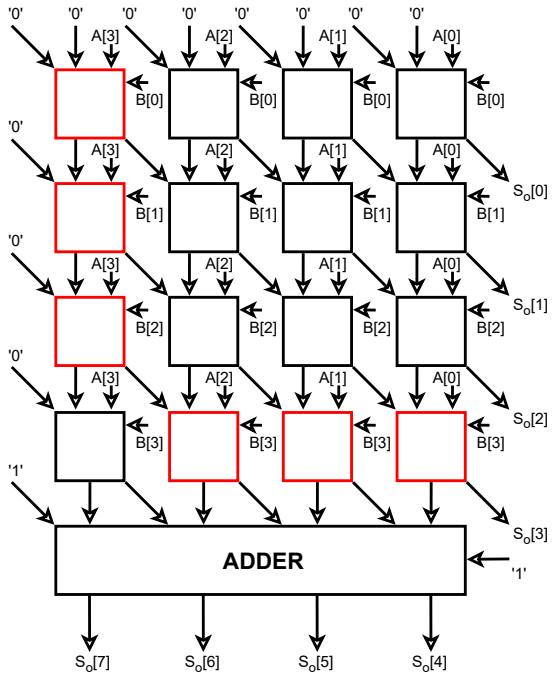
$$y = 2C(y) = \overline{y} + 1 \tag{2.10}$$

We can represent the sum of the two negative terms as in the following:

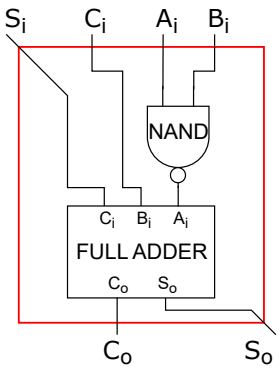
index:	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0	
$-x$:	0	0	x_{n-2}	...	x_1	x_0	0	...	0	0	
$2C(x)$:	1	1	$\overline{x_{n-2}}$	...	$\overline{x_1}$	$\overline{x_0}$	1	...	1	1	+1
x:	1	1	$\overline{x_{n-2}}$	...	$\overline{x_1}$	$\overline{x_0}+1$	0	...	0	0	
$-y$:	0	0	y_{n-2}	...	y_1	y_0	0	...	0	0	
$2C(y)$:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}$	1	...	1	1	+1
y:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0	
$x+y$:	1	0	$\overline{x_{n-2}}+\overline{y_{n-2}}$	...	$\overline{x_1}+\overline{y_1}+1$	$\overline{x_0}+\overline{y_0}$	0	...	0	0	

Starting from the previous result, we can easily see the addition of the two negative terms is equal to the addition of the two terms each with every bit complemented, and with a +1 in position n and in position $2n-1$.

The structure to have this result is the one shown in Figure 2.2a. We can appreciate the structure is almost the same as the one for the previous case, the only change resides in blocks in positions where is necessary to perform the complement of bits of the two negative terms. These blocks are marked in red, an internal view of the block is shown in Figure 2.2b where we can appreciate the inversion is performed through a NAND gate producing the PP $A_i \cdot B_i$. Moreover, here the addition of the two required +1 could be done exploiting all S_i and C_i unused inputs as shown. This structure solves every combination between positive/negative numbers multiplication since, if we consider $a_{n-1} = 0$ then we have $y = 0$, once inverted and a +1 is added, still $y = 0$. One last consideration, related to this sum step, is about the surely generated carry in position $2n$, in the case of a multiplication between two positive or two negative signed numbers. This is due to the inversion process. Usually, this is not a problem, but it becomes important in the following paragraphs analyzing MAC support for precision scalable multiplications.



(a) 4 bit Baugh-Wooley structure for the signed case.



(b) Inverting block.

Figure 2.2

The following case is valid for a multiplication between a signed and an unsigned number, represented as:

$$A = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \tag{2.11}$$

$$B = b_{n-1} \cdot 2^{n-1} + \sum_{j=0}^{n-2} b_j \cdot 2^j \tag{2.12}$$

The multiplication could be expanded as:

$$\begin{aligned} S_o = A \cdot B = -a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j} \\ + 2^{n-1} \cdot \sum_{i=0}^{n-2} b_{n-1} \cdot a_i \cdot 2^i - 2^{n-1} \cdot \sum_{j=0}^{n-2} a_{n-1} \cdot b_j \cdot 2^j \end{aligned} \tag{2.13}$$

As previously done, we correct the negative terms. Then we proceed to the summation.

$$-y = -2^{n-1} \cdot \sum_{j=0}^{n-2} a_{n-1} \cdot b_j \cdot 2^j \tag{2.14}$$

$$y = 2C(y) = \overline{y} + 1 \tag{2.15}$$

$$-z = -a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} \tag{2.16}$$

$$z = 2C(z) = \overline{z} + 1 \tag{2.17}$$

index:	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0	
$-y$:	0	0	y_{n-2}	...	y_1	y_0	0	...	0	0	
$2C(y)$:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}$	1	...	1	1	+1
y:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0	
$-z$:	0	z_{n-1}	0	...	0	0	0	...	0	0	
$2C(z)$:	1	$\overline{z_{n-1}}$	1	...	1	1	1	...	1	1	+1
z:	1	$\overline{z_{n-1}}+1$	0	...	0	0	0	...	0	0	
$y+z$:	1	$\overline{z_{n-1}}+1$	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0	

Here, few considerations are needed:

If we have one signed negative number, and one unsigned number having the MSB equal to 0, we can see the term z is equal to 0. So there is no point in performing inversion, since, even if the result is correct in all cases, in the next

passage we would end up producing a carry due to the summation of all +1 in position $2n-1$ and $2n-2$. Still the summation of a +1 must be performed in position $2n-2$ to accomplish the inversion of the term y, for the sake of using less external +1 as possible the best strategy could be inverting only the bit z_{n-1}, that is equal to zero anyway, without performing the full 2C operation. Thus, the final result is:

index:	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0
$y+z$:	1	$\overline{z_{n-1}}$	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0

Leading to the correct result of:

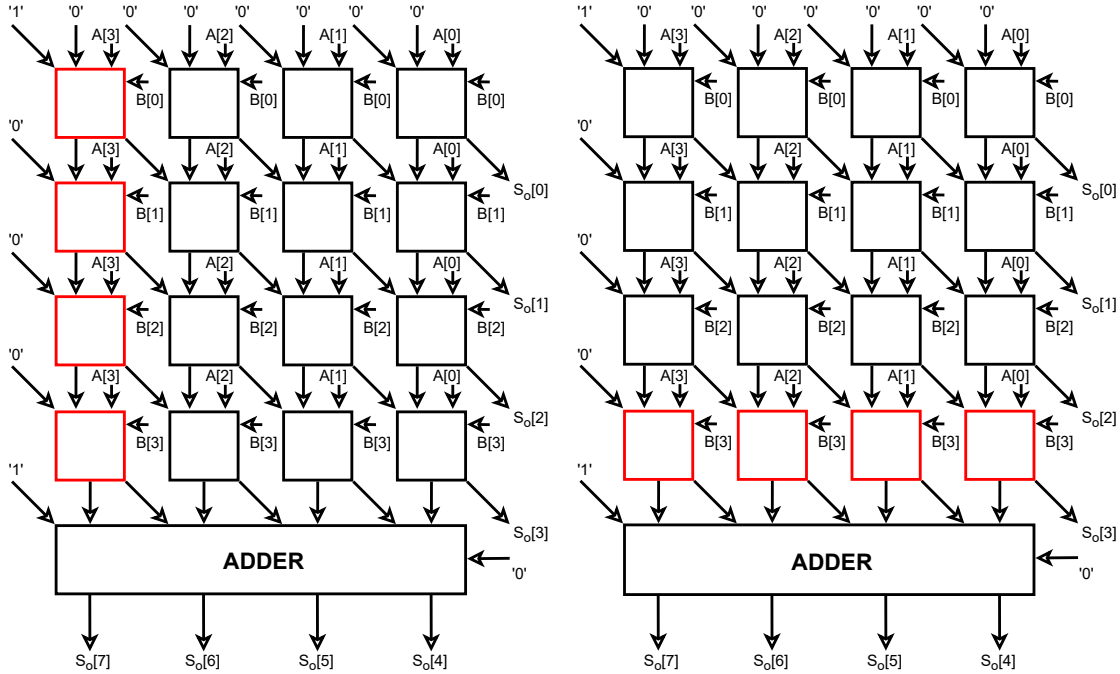
index:	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0
$y+z$:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0

When considering a multiplication between a signed negative, and an unsigned number having 1 as MSB, we can not rely anymore on the previous assumption saying the value of z_{n-1} is equal to 0. Contrary, in this case the value of z_{n-1} is equal to 1. Following the previous approach inverting both y and z:

index:	2n-1	2n-2	2n-3	...	n	n-1	n-2	...	1	0	
$-y$:	0	0	y_{n-2}	...	y_1	y_0	0	...	0	0	
$2C(y)$:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}$	1	...	1	1	+1
y:	1	1	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0	
$-z$:	0	z_{n-1}	0	...	y_1	y_0	0	...	0	0	
$2C(z)$:	1	$\overline{z_{n-1}}$	1	...	1	1	1	...	1	1	+1
z:	1	$\overline{z_{n-1}}+1$	0	...	0	0	0	...	0	0	
$y+z$:	1	$\overline{z_{n-1}}+1$	$\overline{y_{n-2}}$	...	$\overline{y_1}$	$\overline{y_0}+1$	0	...	0	0	

We can see there would be a sum of two +1 in position $2n-2$, leading to a final 0 in that position and a carry to position $2n-1$. In position $2n-1$ there would be a summation of three 1 values, producing again a 1 as a result in position $2n-1$ and a carry to position $2n$. We can notice the result would have been the same if we had just the inversion of the value z_{n-1}, saving computing the 2C operation on the whole z. Fortunately, this condition is the same as the one used for the previous cases of mixed signed-unsigned multiplications.

The structure shown in Figure 2.3a is derived from definitions given in Equation 2.13, along with previously done considerations on where to place the +1 additions, and where to use inverting blocks (the ones in red). Structure of inverting blocks is the same as before. Figure 2.3b follows the same reasoning considering the opposite case, that is, the one of unsigned number multiplied by a signed number.



(a) 4 bit Baugh-Wooley structure for the signed · unsigned case.

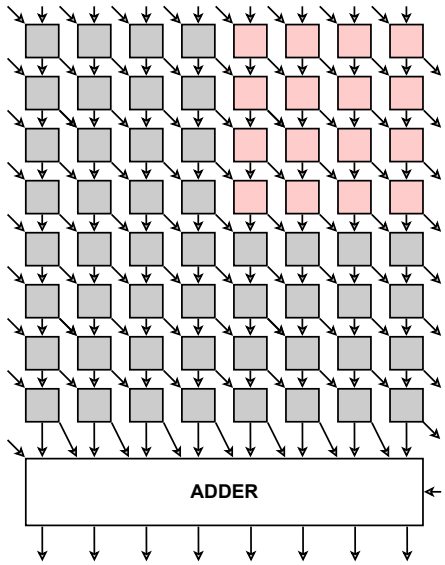
(b) 4 bit Baugh-Wooley structure for the unsigned · signed case.

Figure 2.3

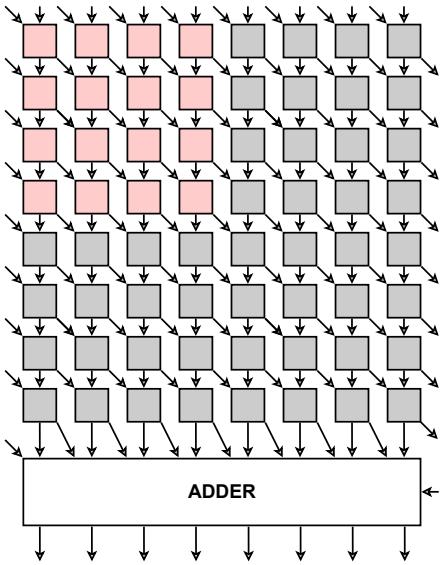
2.2 Sub-word parallel usage of the Baugh-Wooley architecture

Starting from the Baugh-Wooley structure as shown in Section 2.1, it is easy to think the structure is internally progressively performing multiplications and accumulations from given bits with increasing weights along the chain. Considering different input configurations we can derive the following:

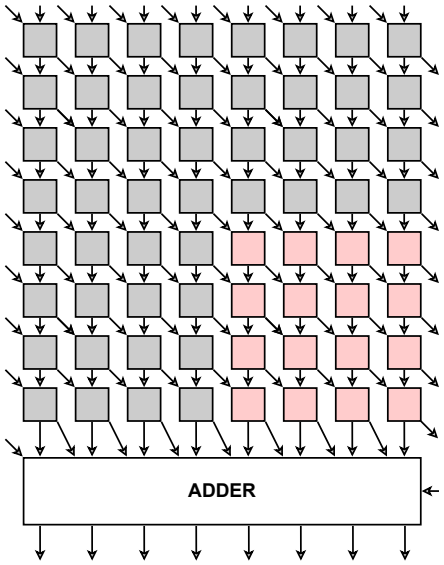
Given a Baugh-Wooley structure capable of performing multiplications on 8 bit inputs, if we consider having inputs on 4 bit, the structure behaves as in Figure 2.4a. Shifting one of the two inputs by 4 position to the left, thus multiplying by 16, leads to the same result shifted by the same amount, and the structure behaves like in Figure 2.4b. There would be the very same result if we invert the shifted operands, thus falling in the condition of Figure 2.4c. Performing a 4 position left shift on both operands leads us to the same result shifted left by 8 positions, and the structure assumes the configuration presented in Figure 2.4d.



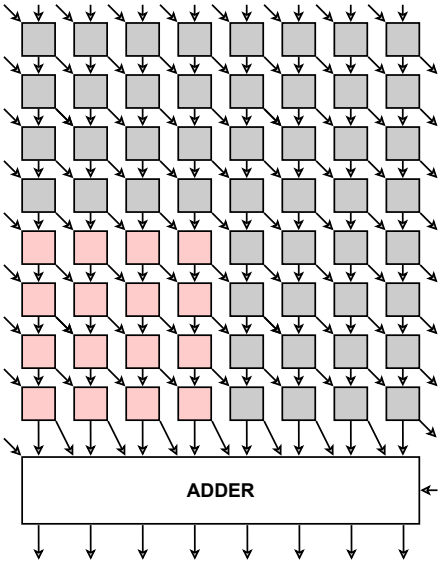
(a) 8 bit Baugh-Wooley used with 4 bit inputs.



(b) 8 bit Baugh-Wooley used with one 4 bit input shifted by 4 position left.



(c) 8 bit Baugh-Wooley used with one 4 bit input shifted by 4 position left.



(d) 8 bit Baugh-Wooley used with both 4 bit inputs shifted by 4 position left.

Figure 2.4

2.2.1 SS operations

Considering the case of Figure 2.4a and the one of Figure 2.4d, these two can exist together independently forming a configuration like the one in Figure 2.5. Here each one returns the corresponding result if we pay attention to disable all other

blocks than the one marked in red. Disabling a block means its internal PP $A_i \cdot B_i$ must be gated to 0. In this way we can perform two independent multiplications in parallel having operands on 4 bit precision, using a structure conceived to perform 8 bit multiplications. This way of computing independent multiplications in parallel is called **Sum Separate (SS)**. As previously mentioned, there are cases in which a multiplication between two signed numbers leads to a carry in the final result, and considering the case of Figure 2.4a this carry could end up being summed to the other multiplication performed in parallel, to counteract this phenomenon a masking procedure must be considered. In Figure 2.5 these positions are marked with an X inside blocks, meaning that the carry propagation have to be selectively masked and not propagated to the subsequent blocks. Given input format on 8 bit final results reflect the following relation:

$$Res[7:0] = A[3:0] \times B[3:0] \tag{2.18}$$

$$Res[15:8] = A[7:4] \times B[7:4] \tag{2.19}$$

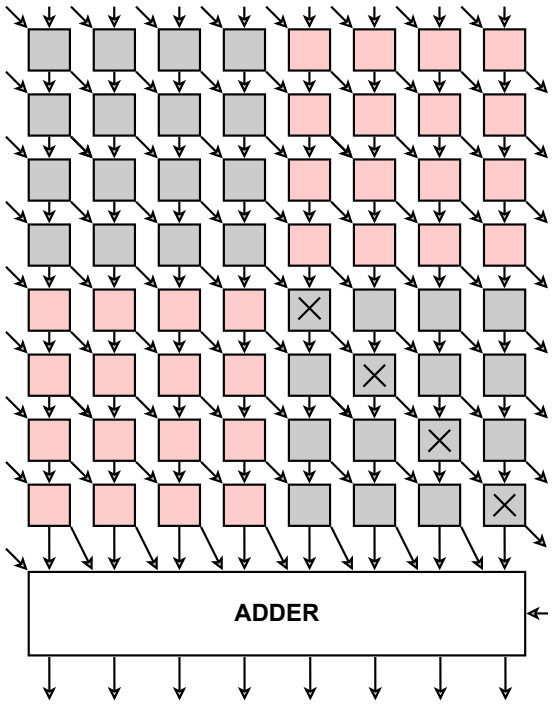


Figure 2.5: 8 bit Baugh-Wooley in SS configuration with 4 bit inputs.

2.2.2 ST operations

Considering the case in Figure 2.4b and the case in Figure 2.4c also these two can exist together forming the configuration present in Figure 2.6, but in this case

given the Baugh-Wooley structure, results of the two independent multiplications are sum together like it happens in a MAC operation. The operation performed is similar to what required by a dot-product. Like in the previous case, all blocks not marked in red have to be disabled. This final result must then be shifted back by 4 position right to counteract the shifting effect resulting from this usage of the Baugh-Wooley architecture. The maximum amount of bits of the final result is 9 bit, given a sum of two independent multiplications with 4 bit operands. This way of computing two multiplications in parallel, adding the results, is called **Sum Together (ST)**. The relation between input format and output result follows:

$$Res[12:4] = A[3:0] \times B[7:4] + A[7:4] \times B[3:0] \tag{2.20}$$

Moreover, from the previous relation we see partial products refers to different input portions. The first comes from the multiplication of the least significant part of input operand A with the most significant part of input operand B, while for the second it happens the opposite. In following sections of this document, this phenomenon is referred as ***cross-product***.

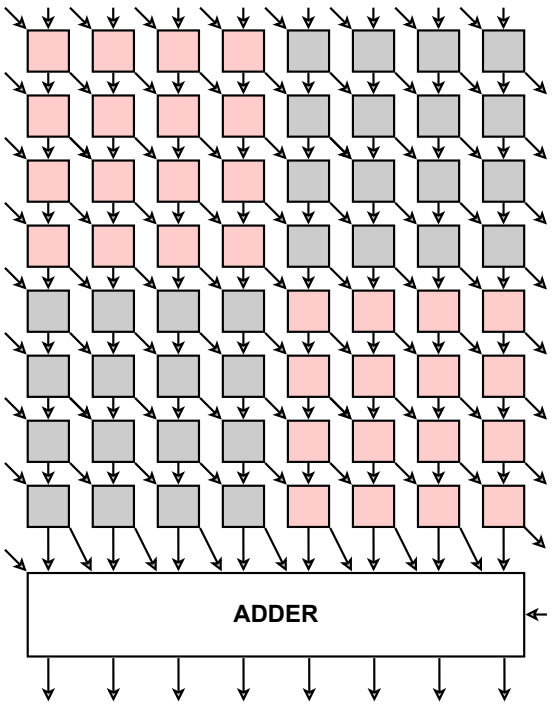


Figure 2.6: 8 bit Baugh-Wooley in ST configuration with 4 bit inputs.

2.3 Theoretical per-operation analysis

In this part some multiplier structures are defined with the objective to satisfy the computation required by different sets of operations. All structures are based on the 16 bit Baugh-Wooley multiplier scheme.

Operations considered are the following:

- One 16 bit operation involving two signed numbers, two unsigned numbers or a mixed combination of signed-unsigned numbers.
- Two 8 bit operations in SS configuration involving two signed numbers or two unsigned numbers.
- Two 8 bit operations in ST configuration involving two signed numbers or two unsigned numbers.
- Four 4 bit operations in SS configuration involving two signed numbers or two unsigned numbers.
- Four 4 bit operations in ST configuration involving two signed numbers or two unsigned numbers.
- Support for MAC operations, involving an external register, for all combinations above.

Structures presented to accomplish subsets of operations in the previous list are the basis for the definition of the final multiplier units presented in Chapter 3.

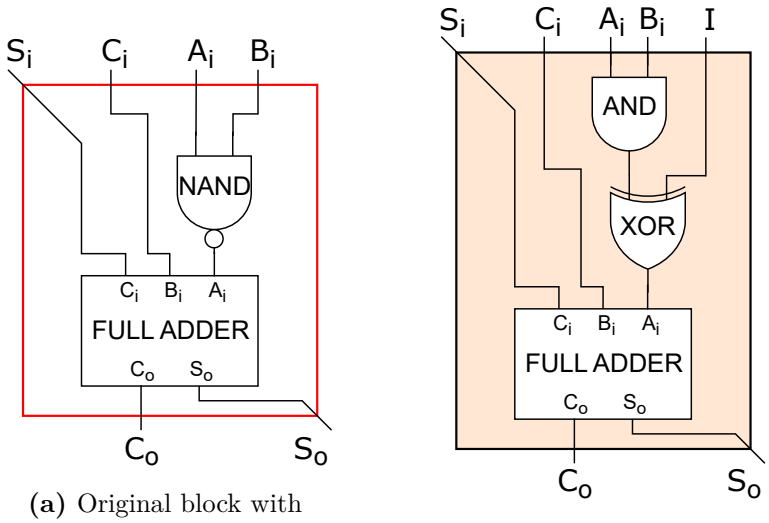
The following subsections derive structures in an incremental way, meaning that, for each subsection, at beginning a baseline structure to deal with a selected group of instruction is derived, but in the end of each subsection is also presented a structure that includes operations of all previous subsections. This way of deriving the structure has the advantage to arrive in the last subsection, coming from a step-by-step derivation, and having the final version including all instructions above at once.

2.3.1 16 bit operations structure

To have a 16 bit structure capable of handling multiplications amid all signed-unsigned operand combinations, it must satisfy all conditions presented in the first paragraph of this chapter.
Specifically, by looking to Figures 2.1, 2.2 and 2.3a we can easily derive the structure represented in Figure 2.8. To summarize, the structure considers the following cases:

- In the case of two unsigned numbers it ends up in a similar condition as the one in Figure 2.1. External signals are all set to logical 0 value.
- In the case of two signed numbers it ends up in the condition of Figure 2.2. External signals assume the following configuration: $S_0 = 0$, $S_1 = 1$ and $N = 1$. Moreover, the output carry of the final adder assumes the logical 1 value both in cases of multiplications between two positive or two negative numbers.
- In the case of one signed number and one unsigned number it relapses in the condition of Figure 2.3a. External signals assume the following configuration: $S_0 = 1$, $S_1 = 0$ and $N = 1$.

Fist of all, inverting blocks must have the possibility to select if inverting the internal partial product $A_i \cdot B_i$ or not. To perform this, the inverting block structure presented in Figure 2.2b, and recalled in Figure 2.7a, is changed adding a XOR gate acting as controllable inverter through the signal I, standing for "Invert". Final block structure is visible in Figure 2.7b.



(a) Original block with inversion.

(b) Block with controlled inversion.

Figure 2.7

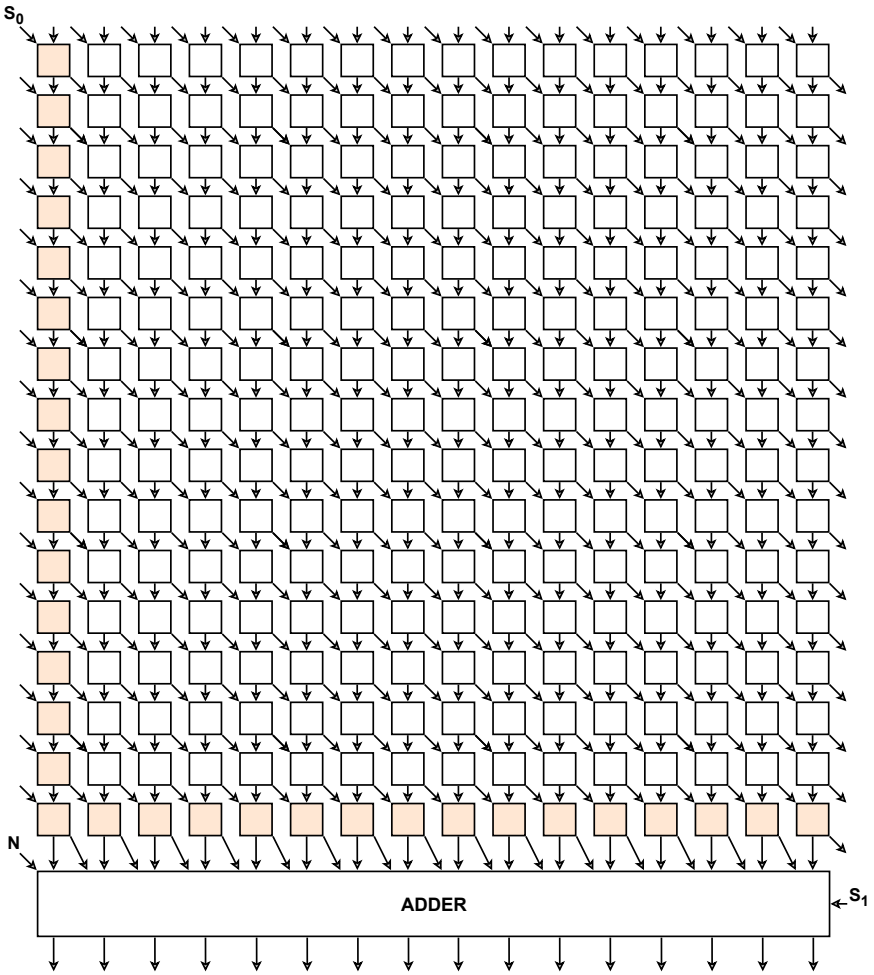
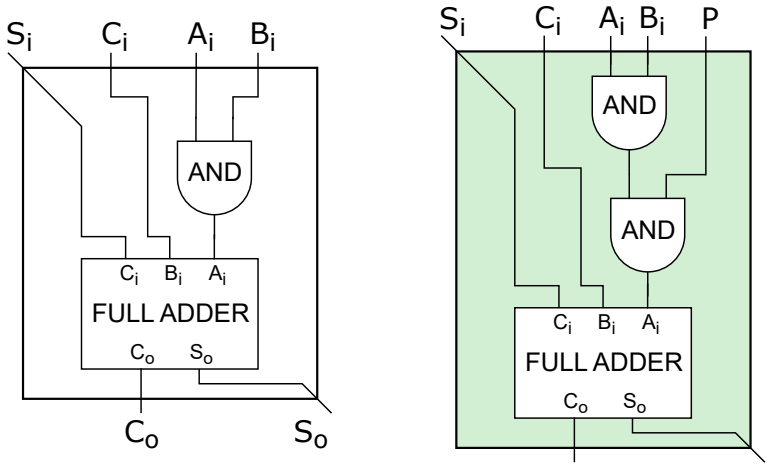


Figure 2.8: Baugh-Wooley structure to accomplish operations on 16 bit inputs.

2.3.2 SS operations structure

To accomplish two 8 bit operations in SS configuration, considering Figure 2.5, it is clear, all unused blocks must be in a condition in which the partial product term $A_i \cdot B_i$ does not contribute to the overall result. To accomplish this state, each multiplicative block must be modified adding a masking interface between the AND gate, performing the internal multiplication, and the corresponding full adder input. By recalling the block in Figure 2.1b, the same visible in Figure 2.9a, we can appreciate this modification is obtained using a second AND gate, which input here called P, standing for "Propagate", is visible in Figure 2.9b. In this case propagation and internal sum of signals S_i and C_i keeps being computed even if the masking signal is asserted.

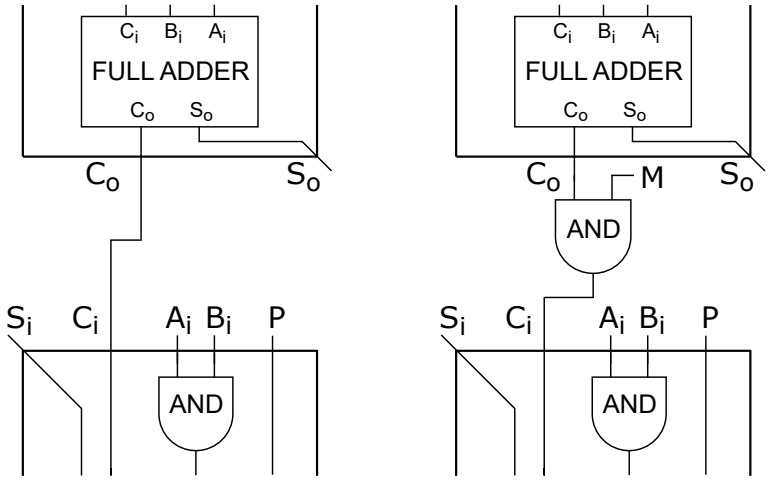


(a) Original multiplicative block.

(b) Block with masking.

Figure 2.9

At this point, by looking at Figure 2.5, we can notice also propagation of carries between independent computations have to be blocked in order to have the two operations fully independent (i.e. where an X is placed inside a block). To accomplish this, the propagation of signals C_i is subject to another gating procedure as shown in Figure 2.10b. Again, in this condition an AND gate is exploited to mask the propagation.

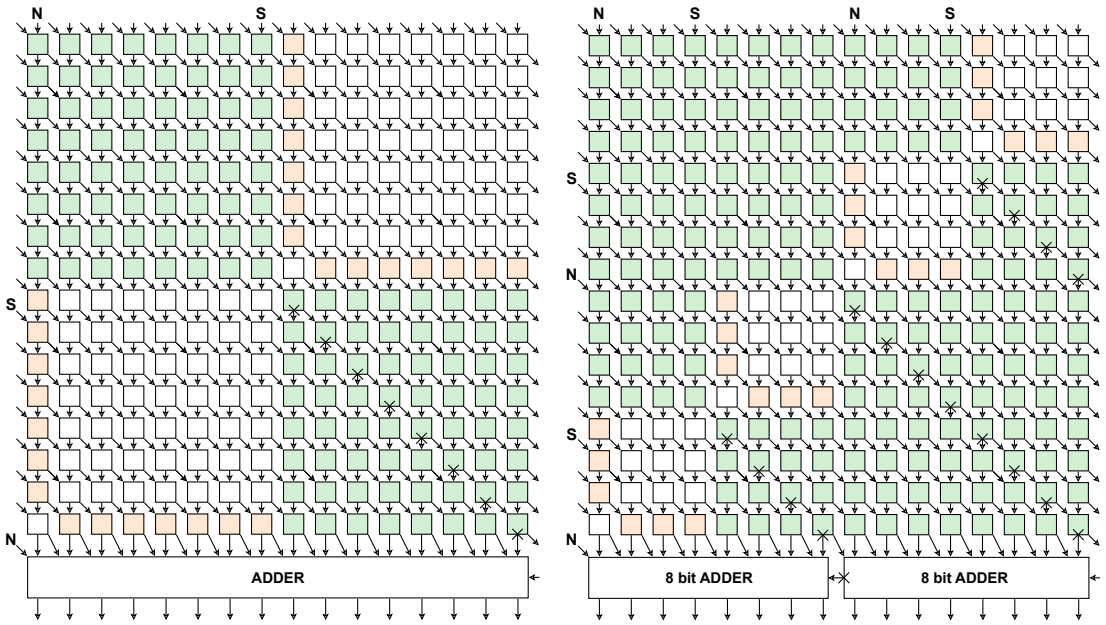


(a) Propagation of C_i signal between blocks.

(b) Masking of C_i signals between blocks.

Figure 2.10

The overall structure is presented in Figure 2.11a. Where an X is placed at the bottom of a block, it refers to the technique in Figure 2.10b. Moreover, for signed operations also the addition of +1 terms is needed, this is represented in the same figure by signals S and N, which are both at logical value 1 only for signed operations. Where not an S nor an N is placed, it means a logic 0 is placed.



(a) Baugh-Wooley structure capable of computing two 8 bit operations in SS mode.

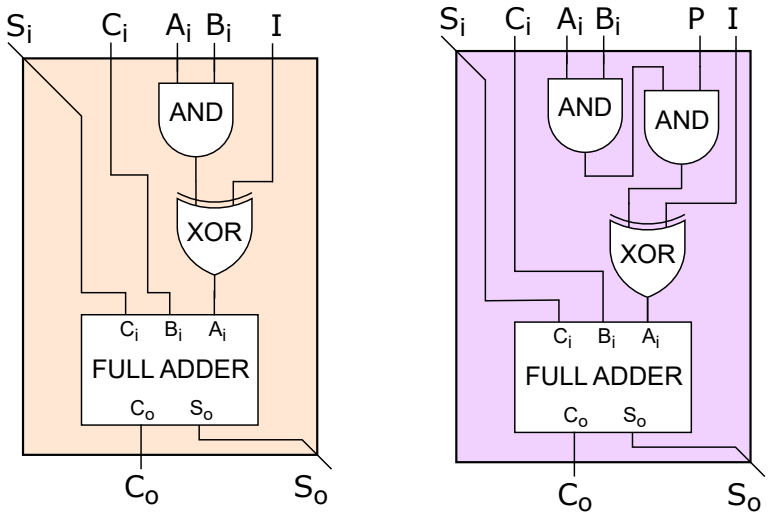
(b) Baugh-Wooley configuration capable of computing four 4 bit operations in SS mode.

Figure 2.11

The case of four 4 bit operations in SS configuration follows the very same considerations as this previous one. The final structure is the one in Figure 2.11b. Again, where an X is placed inside a block it means the subsequent connection for C_i signal is as the one presented in Figure 2.10b. As before, the addition of +1 terms is performed through S and N signals in the figure. These are present for each scaled Baugh-Wooley which computes an independent operation. Finally, in this context, carry propagation must be blocked also inside the adder unit. To do this, the 16 bit adder unit is split into 8 bit adders, and the carry from one adder to the following is masked with the same strategy as in Figure 2.10b.

Now, we can consider the case of a structure able to compute the subset of operations including 16 bit multiplication, two 8 bit operations in SS mode, and four 4 bit operations in SS mode. To accomplish all operations in this subset we need a union of all previous structures all in one. The first thing to note is there are blocks performing the inversion operation, recalled in Figure 2.12a, which in the

SS multiplication cases must be capable of being switched off. The modification exploited is the same as for the previous case, thus adding an AND gate in between the AND performing the multiplication and the XOR to selectively perform the inversion, leading to the block depicted in Figure 2.12b. One characteristic of this block is also the possibility of generating a +1 itself, only in the case the internal multiplication is gated, in fact if external signal condition is $P = 0$ and $I = 1$ then the input to the full adder is at logical value 1.



(a) Block with controlled inversion.

(b) Block with controlled inversion and masking.

Figure 2.12

The final structure of this multiplier is presented in Figure 2.13. External input signals marked with names $SS4$, $SS8$ and $M16$, are considered to be at logical value 1 only when computing the corresponding operation in signed mode. Single blocks control signals, P and I, are set accordingly to all previous cases.

2.3.3 ST operations structure

Moving to the ST operation group, here it starts with the case of two 8 bit operations in ST configuration. For this operation a structure similar to the one present in Figure 2.6 is required. Also in this case unused blocks have to be switched off using the modification presented in Figure 2.9b. Final structure can be seen in Figure 2.14a. To accomplish the +1 addition in specific position as requested by the Baugh-Wooley algorithm, we can pre-compute the result to be summed and sum it only once. In this view, summations of +1 terms should be done in positions

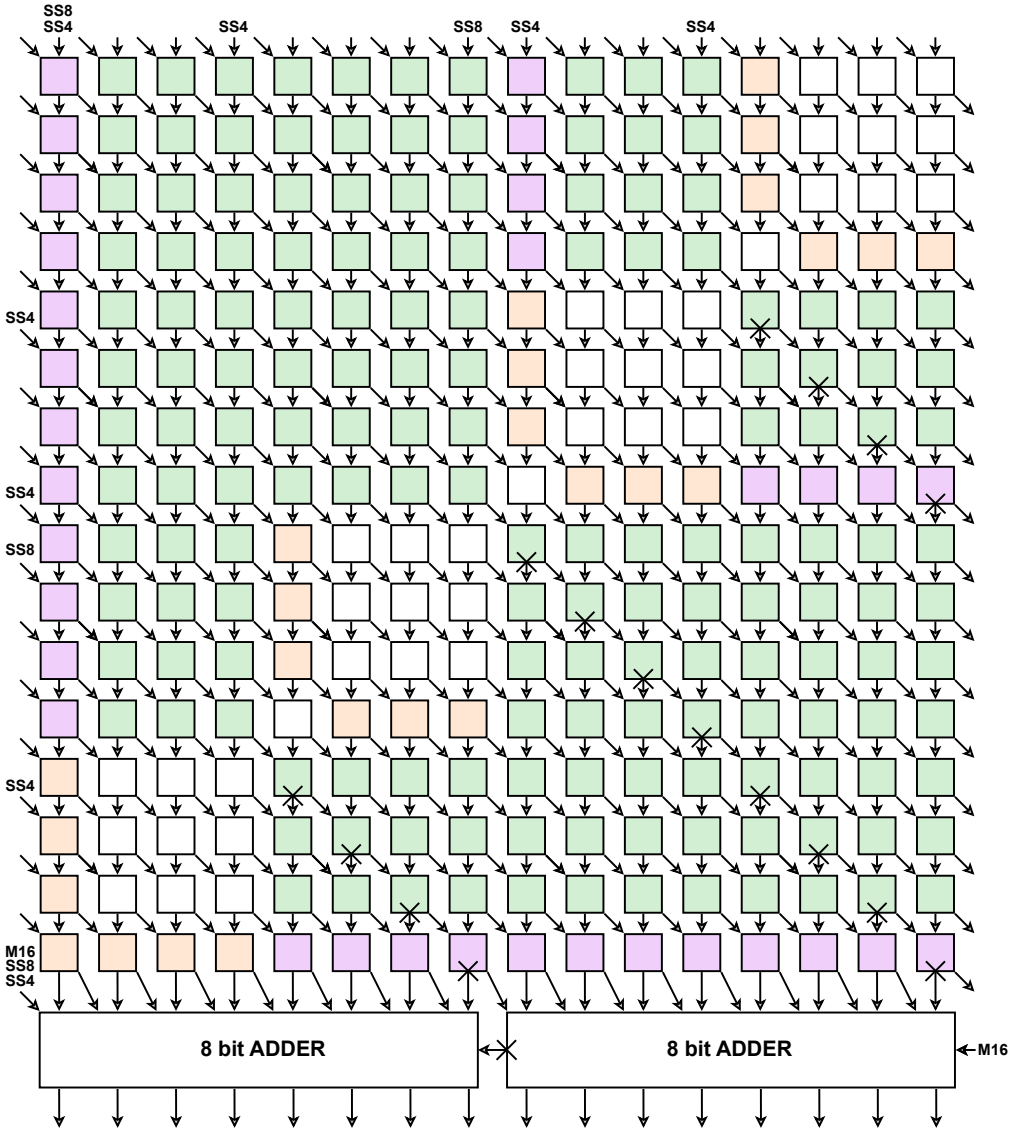
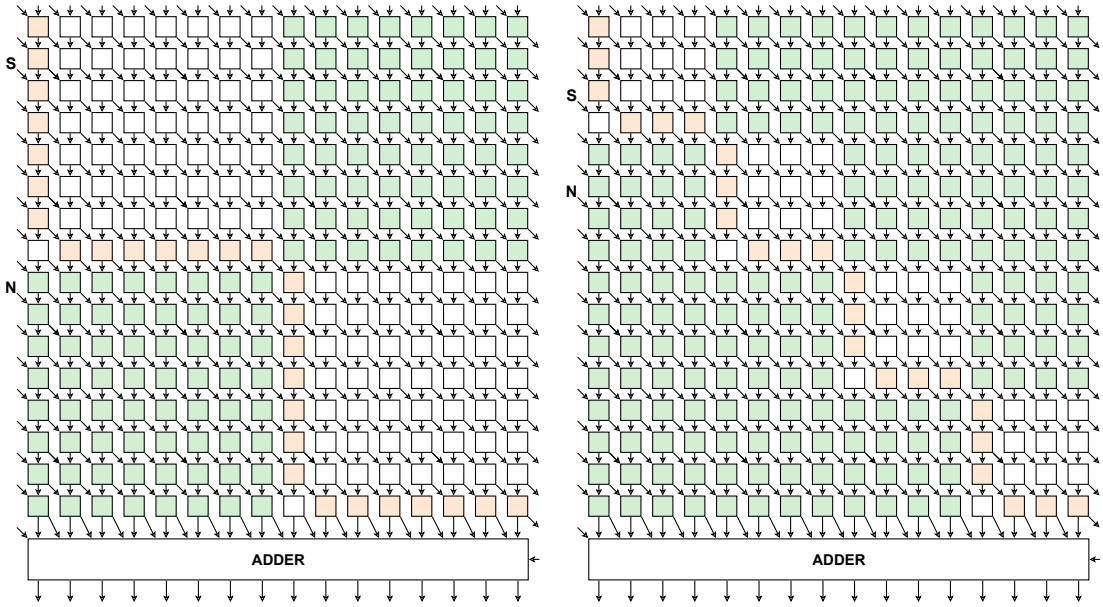


Figure 2.13: Reconfigurable Baugh-Wooley structure for 16 bit standard multiplication, 8 bit and 4 bit operations in SS configuration.

16 and 23 (counting indexes of S_i signal) for both the 2 internal multiplications. The pre-computing phase is essentially considering that two summations of +1 in position 16 could be expressed as a single summation in position 17. In the same way, two summations in position 23 would end up as a single summation in position 24. In Figure 2.14a, the two signals S and N already takes this strategy into account.



(a) Baugh-Wooley configuration capable of computing two 8 bit operations in ST mode.

(b) Baugh-Wooley configuration capable of computing four 4 bit operations in ST mode.

Figure 2.14

A similar structure could be derived to compute four 4 bit operations in ST configuration and is presented in Figure 2.14b. Here, four multiplications are carried out in parallel, meaning that the pre-computing phase moves positions where to add the +1 terms incrementing them by 2. Also in this case, positions of signals S and N take already into account the pre-computing phase.

As previously done we can consider the case of computing the subset of operations that is a 16 bit multiplication, two 8 bit multiplications in ST mode, and four 4 bit multiplications in ST mode. Also in this case we need the block modification presented in Figure 2.12b to be able to switch off blocks performing the inversion of the internal multiplication. Final structure is presented in Figure 2.15. External signals with names $ST4$, $ST8$ and $M16$ are considered to be at logical value 1 only when computing the corresponding operation in signed mode.

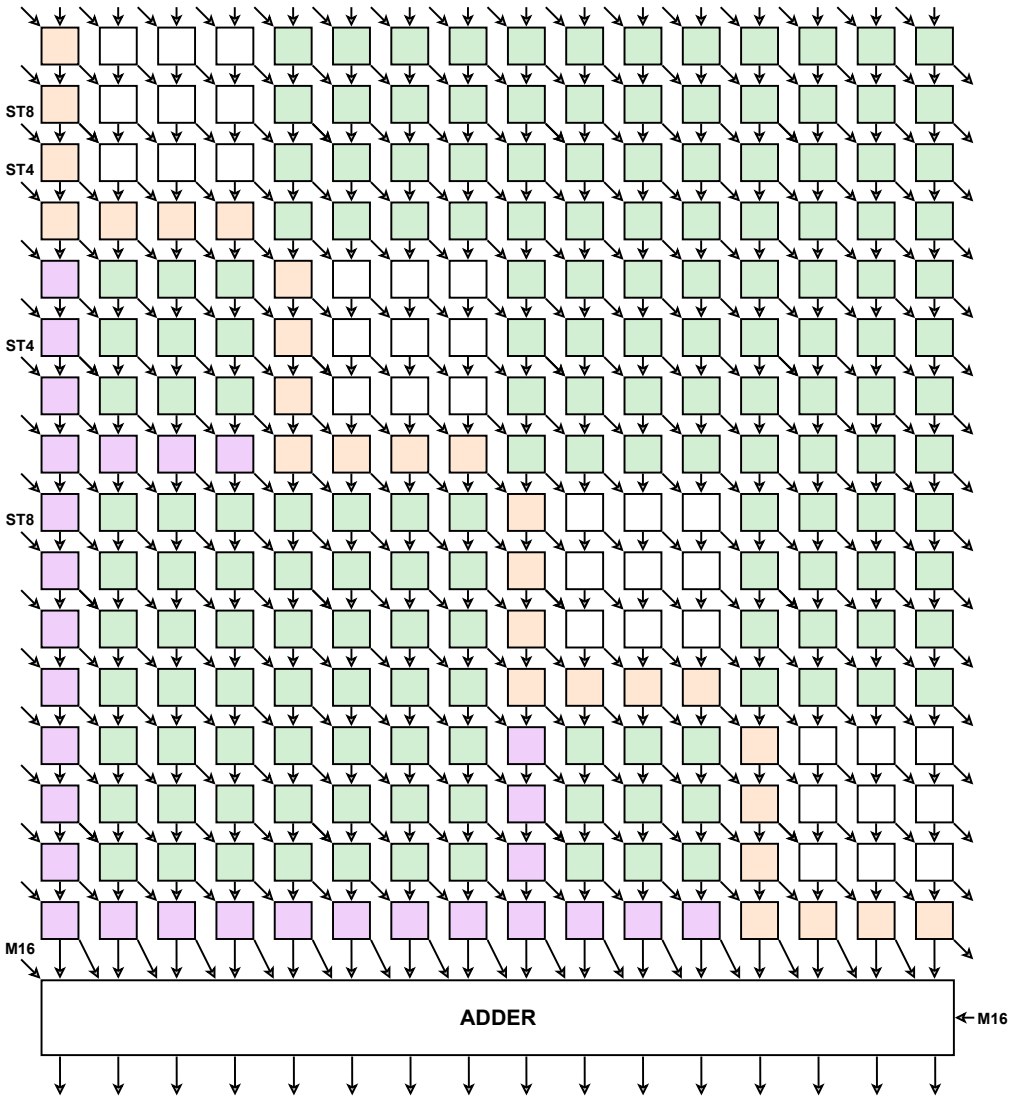


Figure 2.15: Reconfigurable Baugh-Wooley structure for 16 bit standard multiplication, 8 bit and 4 bit operations in ST configuration.

For the sake of developing requirements for subsets of operations, we can now consider a structure able to deal with all subsets of operations presented so far, that is, one 16 bit multiplication, two 8 bit operations both for SS and ST configurations, and four 4 bit operations both for the SS and ST modes. In this case, the structure is a combination of ones presented in Figure 2.13 and Figure 2.15, all blocks have to be able of switching off leading to the structure visible in Figure 2.16. It still holds the same convention for external per-operation inputs and for control bit of single blocks (i.e. $SS4$ $SS8$ $ST4$ $ST8$ $M16$).

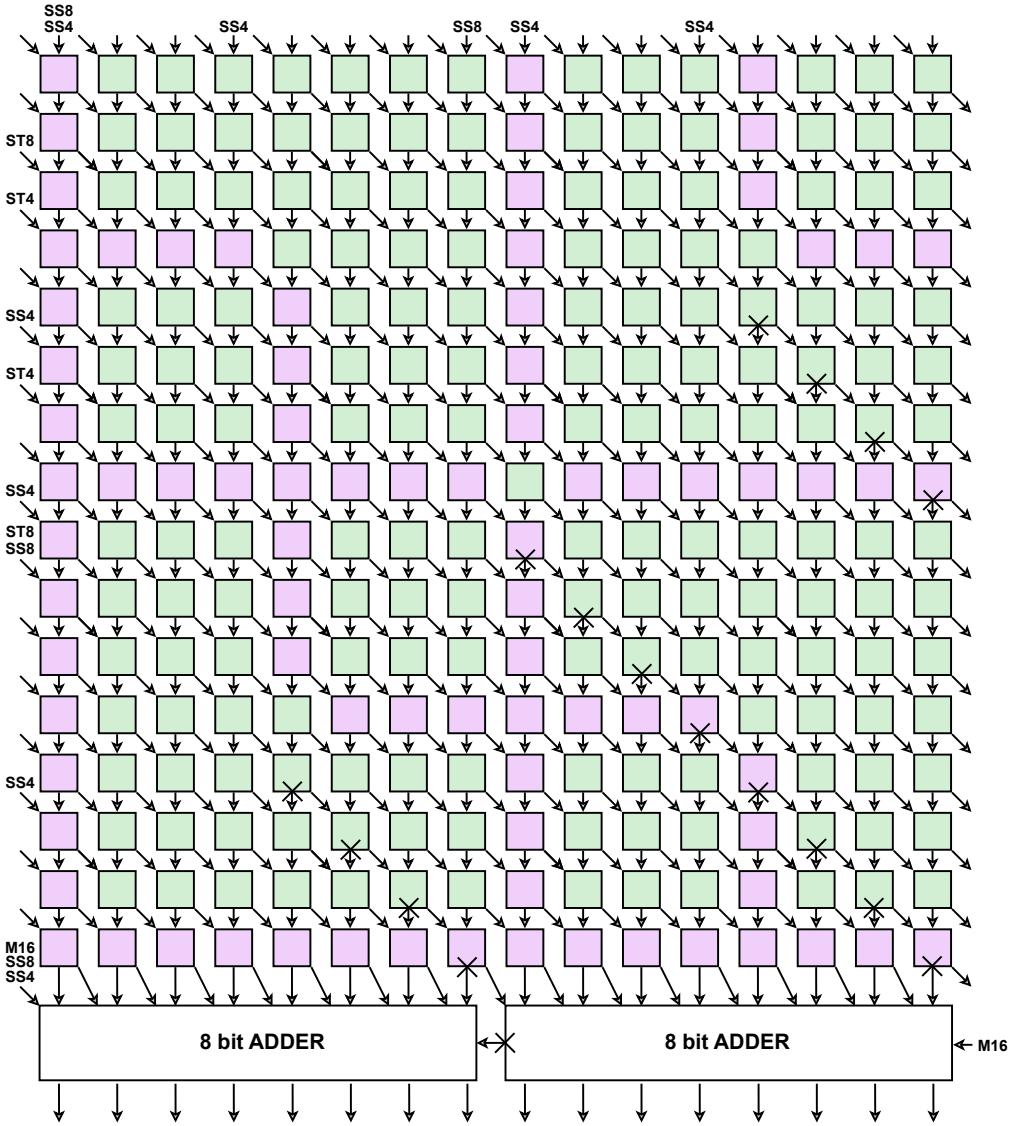
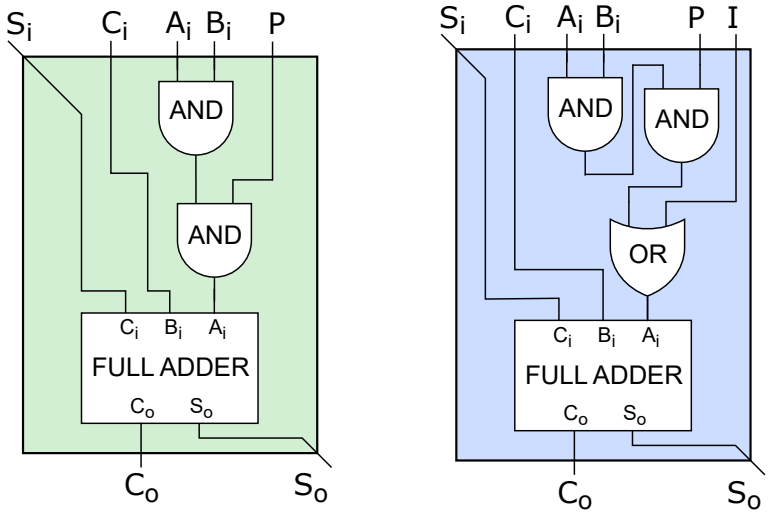


Figure 2.16: Reconfigurable Baugh-Wooley structure for 16 bit standard multiplication, 8 bit and 4 bit operations in both SS and ST configuration.

2.3.4 MAC operations structure

Now we can consider adding the possibility of computing MAC operations, thus accumulating the result over time. To satisfy this requirement, a clever move could be exploiting inputs S_i present along the top side and left side of the standard Baugh-Wooley structure as marked in Figure 2.1a. When considering using the S_i inputs, it means these can not be used to add the +1 in specific positions as done before. When deriving the block in Figure 2.12b, that is, the block having both masking and controllable inversion, we saw this block is able to generate a +1 if it

is inactive (i.e. $P = 0$ and $I = 1$); moreover inputs marked as C_i are still available. Exploiting this property makes the already derived structure almost ready-to-use. Still, some considerations are needed. For the case of a single 16 bit multiplication, the only addition required is an extra block containing the logic of a half adder placed in the left-down corner. Motivations reside in the fact this case can be seen as a sum of two 32 bit values, one produced by the multiplier, while the second provided through S_i inputs. All SS operations are ready-to-use to accumulate the result, moreover there is no risk of a carry propagation in the area involved in computing another multiplication considering countermeasures already present in blocks having an X in Figure 2.16, that are, the ones visible in Figure 2.10b. For unsigned ST case, no addition is required; for the signed ST case, unfortunately, external +1 required by the algorithm are in a position in which no blocks capable of generating a +1 are present, this means one of blocks in that diagonal, indifferently, have to be changed accordingly. In Figure 2.9b, recalled here in Figure 2.17a, there is the standard multiplicative block with the masking AND gate, the minimum logic to make it able to produce a +1 starting from an external signal, is to add an OR gate whose output is at logic 1 whenever the input I is asserted. Changing a green block to a blue block gives at least a valid position in the diagonal to generate the +1 required.



(a) Multiplicative block with masking.

(b) Block with masking able to generate a logic 1.

Figure 2.17

This is not the only modification required by ST operations. The output of an operation at a reduced precision in ST mode is, as already analyzed in Paragraph 2.2, is shifted left by an amount of bits equals to bit-width of the multiplier for standard operations minus the bit-width of the reduced operation. Thus, in this

case, considering a 16 bit multiplier the shift amount for an 8 bit operation in ST mode is equal to $16-8=8$ bit, while for a 4 bit operation is $16-4=12$ bit. This also means the maximum precision is reduced with respect to the precision of the overall structure, and limited to $32-8=24$ bit for the 8 bit case, and $32-12=20$ bit for the 4 bit case.

Intuitively, each multiplication for which the result is negative does not guarantee a sign extension of the result, and this fact limits the precision of the two 8 bit operations in ST configuration to 17 bit, while for four 4 bit operations in ST configuration the precision is limited to 10 bit.
The reason of this phenomenon follows.

Analyzing the result of each combination for the case of four **4 bit** operations in ST configuration, given the order of internal operations does not matter, we can reduce the number of combination to the following five conditions:

1. positive + positive + positive + positive.
2. positive + positive + positive + negative.
3. positive + positive + negative + negative.
4. positive + negative + negative + negative.
5. negative + negative + negative + negative.

In condition 1, we have each partial result having a carry bit set in position n, where n is the number of bit of the result, that is, 8 bit for a 4 bit multiplication. The sum of those carries would be a carry in position 10, so outside the 10 bit in which the result is guaranteed to be correct.

position:	10	9	8	7	6	5	4	3	2	1	0
x:	0	0	1	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
y:	0	0	1	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
z:	0	0	1	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
k:	0	0	1	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
$R=x+y+z+k$:	1	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0

We can see we just have to correct the last carry in position 10. This can be done summing a -1 term in that position, thus in two's complement notation, adding a series of $1s$ of the required precision, in this case, at most $20-10=10$ bit, to fully achieve the maximum precision of 20 bit given by this configuration.

In condition 2, we have one negative result out of four, in this case we should have an extension of this result through positions 8 and 9, but exploiting the carry of each positive operations the extension is given for free, leading to:

position:	10	9	8	7	6	5	4	3	2	1	0
x:	0	0	1	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
y:	0	0	1	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
z:	0	0	1	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
k:	0	0	0	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
x:	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
y:	0	0	0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
z:	0	0	0	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
k:	0	1	1	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
$R = x + y + z + k$:	0	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0

Still the extension of the sign from position 10 is required, that is, adding a series of 1 as previously done.

The same stands for conditions 3 and 4 with two/three negative and two/one positive results to sum. Here also the sign extension is the same, thus, adding a series of 1s from position 10 to 19.

For condition 4, there are four negative results to sum, but four negative results would have meant an addition of +1 in positions leading to the same result on 10 bit as if these are not present, as shown in the following:

position:	10	9	8	7	6	5	4	3	2	1	0
x:	0	0	0	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
y:	0	0	0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
z:	0	0	0	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
k:	0	0	0	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
x:	0	1	1	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
y:	0	1	1	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
z:	0	1	1	z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
k:	0	1	1	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0
$R = x + y + z + k$:	0	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0

So, we can just extend the result, as previously done, from position 10 to position 19.

For the case of two 8 bit operations in ST mode the same assumptions hold. In this case we have just two intermediate values to sum, and the extension is from position 17 to 23.

In all previous cases the sign extension could be done exploiting the capability of the last row of blocks to generate a series of +1 with the proper control signal combination. Moreover, positions in the Baugh-Wooley structure are shifted of the

same amount the result is shifted, thus it is from position $23 + 8 = 31$ to position $17 + 8 = 25$ for the 8 bit case, and from position $19 + 12 = 31$ to position $10 + 12 = 22$ for the 4 bit case.

Final structure is shown in Figure 2.18. Every block that need to generate a $+1$ is marked inserting internally the corresponding operation name with the same meaning as previous cases, whenever that operation is performed in signed mode those blocks generates a $+1$. Control signals of each block, P and I, are set according to all previous cases of SS/ST operations.

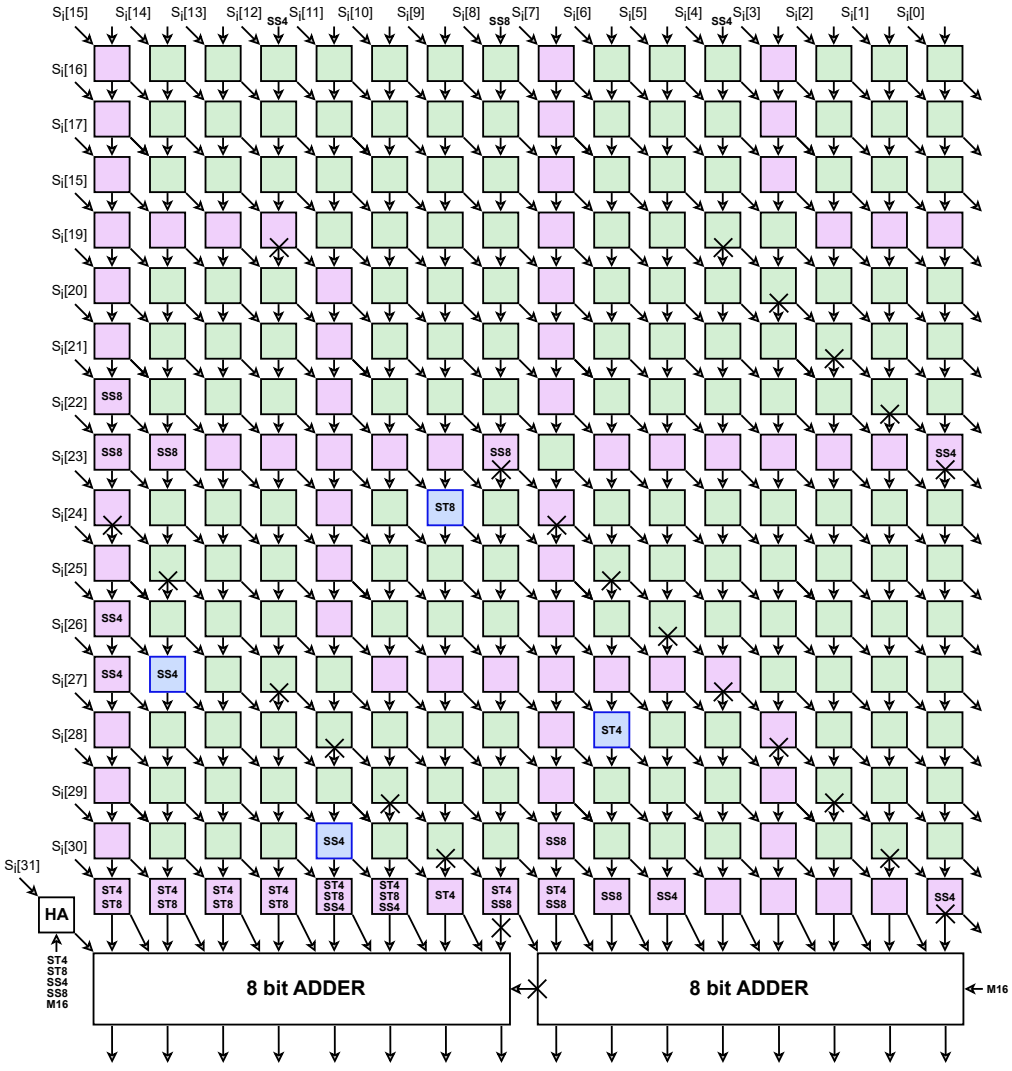


Figure 2.18: Reconfigurable Baugh-Wooley structure for 16 bit multiplication, 8 bit and 4 bit operations in both SS and ST configuration, and MAC support.

Chapter 3

Ibex core modifications and synthesis results

This chapter starts with the description of the Ibex core architecture, including a preamble concerning RISC-V ISA [51] motivations, concluding the first paragraph analyzing in a more detailed way the two optional behavioral multiplier units, named ***Fast*** and ***SingleCycle*** respectively, already included in the Ibex flow.

The second part details variations to describe, in a behaviorally equivalent way, both the *Fast* and the *SingleCycle* structures such that these are able to accomplish all required set of operations (see Paragraph 1.3). Then, starting from considerations done in Paragraph 2.3, another section details the two final gate-level structures, again targeting both the *Fast* and the *SingleCycle* variants.

The chapter concludes with the last section devoted to the synthesis reports of these four derived Ibex core structures, analyzing power, performance and area of each one compared to the corresponding two baseline versions originally present in the Ibex core.

3.1 Ibex core introduction

The Ibex core processor, developed initially under the name *zero-riscy* [25], is part of the PULP platform [52], which stands for "Parallel Ultra Low Power". The objective of the PULP platform is to demonstrate good performance through parallelization of small processors, organized in clusters, and conceived for near/sub-threshold conditions. Contribution to the Ibex core is actually done by lowRISC [47], who maintain actively the development.
The Ibex core is compliant with the open-source RISC-V ISA. RISC-V is, as the name says, the fifth definition of a "Reduced Instruction Set Computer" ISA, proposed and maintained by the University of California, Berkeley. Reasons behind the definition of a new RISC ISA is mainly to cleverly define a good starting point

each one can agree with, to simplify development of processors without having to provide a software stack, and to guarantee compatibility to existent software stacks that only a standardized basis could guarantee; and in the opposite view, to simplify the development of a software stack on a reference of low-level instructions, that guarantees the existence and feasibility of the HW counterpart. All of this is provided in a royalty-free condition. This is not only about producing reference cores to be licensed to partner to simplify dedicated SoC development, but it is about giving everyone the possibility to start its own processor core development without worrying of something that is not electronic related.

The RISC-V ISA [54] [1] is organized as a base set of a given instruction length, and several extensions compatible with each specific base set.

There are six different base sets:

- **RV32I**: this is the base set for a 32 bit processor, including only simple operations, such as jump/branch management, addition, interrupt callback, load/store, shift, logical, memory ordering (FENCE). Not focusing on the instruction format which can be retrieved from the reference [54], we note the number of register for this base set is fixed to 31 general purpose registers of length equal to 32 bit, with 1 register hardwired to logical value 0.
- **RV64I**: the 64 bit base set redefine the meaning of instructions present in the RV32I. The reason is every instruction by default operates on the length of registers, that, in this case, is extended to 64 bit. Still, new instructions are added to keep having the 32 bit variant of each operation, e.g., ADDW is performing a 32 bit addition extending then the result to 64 bit. All these instructions are named in the format $*W$.
- **RV128I**: this base set is still in definition, but the derivation is similar to the RV64I one. All instructions now operate on an increased bit-width of 128 bit while some new instructions are added to keep guaranteeing the existence of operations on 64 bit and 32 bit.
- **RV32E**: this base set comes as a specific variation for embedded systems. In the embedded world optimization is relevant. For these reason, in this base set the number of registers is reduced from the standard 31 registers to 15 register, plus 1 register hardwired to value 0, gaining, as stated [54], 25% less area coming from the register file halving in most of the synthesis scenarios.
- **RV64E**: also the 64 bit embedded variance has a number of register in the register file that is halved.

[1] Version used in this chapter is the 2023-04-27, more recent version should be compliant.

- **RVWMOI**: adds support for multithreading memory coherence other than FENCE instructions.

There are also many extensions to a base set, some of them not ratified yet. Here it follows the description only of extensions supported by the Ibex core.

- **M-extension**: this extension adds instructions related to multiplication and division operations. The total number of instructions depends on the base set considered, in the case of RV32I 8 instructions in total are added, while for the RV64I case it increases to 13 instructions to guarantee compatibility for the 32 bit operations. Multiplication related instructions are:
  - **MUL** returns the lower-part of the result of a 32 bit multiplication between values in two registers (R-type).
  - **MULH** returns the upper-part in the case of two signed values.
  - **MULHU** returns the upper-part in the case of two unsigned values.
  - **MULHSU** returns the upper-part in case the first value is signed while the second is unsigned.

  Division related instructions are:
  - **DIV** returns the result of a division between values in two registers considering them as signed.
  - **DIVU** returns the result of a division between values in two registers considering them as unsigned.
  - **REM** returns the remainder of a division operation considering signed values.
  - **REMU** returns the remainder of a division operation considering unsigned values.
- **C-extension**: in some applications, specifically embedded systems related ones, the code size is still important, so, there may be scenarios in which a lower code size is preferable than reducing the final core area occupation. For this reason the C-extension adds support for compressed instructions on 16 bit. Instructions offered in compressed mode are a limited number, and are the most commonly used ones present in any base set, thus, this extension could be added not only to the RV32I and RV32E but also to RV64I, to RV64E and even to RV128I. Compressed instructions have some limitations in expressiveness, but they can replace typically 50%-60% of instructions in a given program [54]. Still architectures supporting those instructions have to include special HW for the decoding phase meaning, as previously mentioned, the reduction in code size come at the cost of a higher area. To have a view of the instruction format see [54].

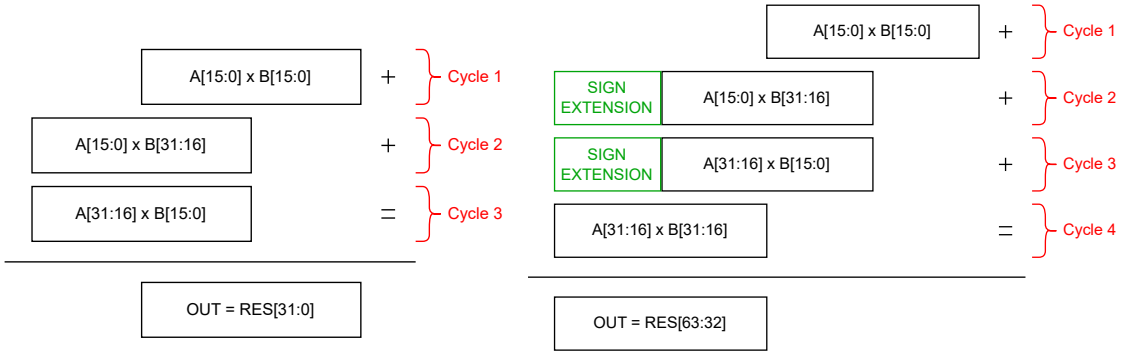
- **B-extension**, stands for "Bit manipulation extension" and it's aimed in providing support for all operations involving bit-level manipulation, that are, bit insertion, masking, bit test, rotation, funnel shift, bit permutation and others. In the official reference [54] there is no draft of these instructions yet. Information about proposed work so far could be retrieved from [55] where the version $v.1.0.0$ of this extension is made available.

The description of the Ibex core that follows is a high-level view of the architecture. The only in-depth analysis is done for the multiplier units and what is related to that. To have an in-depth analysis for all other parts of the architecture see the Ibex official reference [49].

The Ibex structure is synthesized starting from some design parameters to better fit the final application it should be compliant with. Ibex comes in four different versions, all of them describe an in-order single-issue processor compliant with the RV32I RISC-V base set. For the ***micro*** and the ***small*** versions, core structure is organized as a two-stage pipeline. The first stage just includes the fetch operation from memory, while the second stage includes all other operations involving also the writeback phase, and data reading/writing from/to memory phase. For the ***maxperf***, and the ***maxperf-pmp-bmfull*** versions the core structure changes to a three-stage pipeline moving the writeback phase to the third stage. Other main difference between Ibex versions is the different support of the RISC-V ISA extensions. The *micro* variant supports the RV32E base set, and the C-extension. The *small* variant, like *maxperf* variant, support the RV32I with M-extension and C-extension, while the *maxperf-pmp-bmfull* supports also the B-extension, specifically the version $v.1.0.0$ previously mentioned. Other differences are the *small* version has not a dedicated branch target ALU, and has a small multiplier unit, that is, the *Fast* version, while the *maxperf* and the *maxperf-pmp-bmfull* includes both a dedicated branch target ALU and a larger and faster multiplier unit, that is, the *SingleCycle* version. Both these two versions of multipliers are described in the following subsections.

This previous description is a preamble to the internal Ibex parameters activating and deactivating single features. Even if the Ibex core comes with those four versions, others may be conceived mixing, carefully, single internal parameters. For this thesis, the considered structure is more similar to the *small* version of the Ibex, where it makes use of a multiplier unit, not supporting the RISC-V B-extension and having a two-stage pipeline. The only modification is in the $RV32M$ internal Ibex parameter, which can be set to three values: *Slow*, this is not instantiating a real multiplier, the multiplication procedure uses the main ALU adder, looping until the result is fully computed. This computation takes up to 32 clock cycles for a single multiplication, but of course it is better than nothing. Then we have the *Fast* and the *SingleCycle*. When considering selecting the *Fast* multiplier unit,

we have exactly the *small* version of the Ibex processor. When instead considering selecting the *SingleCycle* multiplier unit, we are creating a version that is faster in performing multiplications but occupies more area. These two multiplier units are the basis for subsequent analysis of PPA metrics considering the four designed multiplier structures substituting them.



(a) MUL instruction internal cycles.

(b) MULH, MULHU and MULHSU instructions internal cycles.

Figure 3.1: *Fast* multiplier, 32 bit MUL operations splitting procedure.

3.1.1 The 32 bit *Fast* multiplier

The Ibex *Fast* version of the multiplier includes a single 17 bit, behaviorally described, multiplier. This multiplier is used for all multiplication related instructions present in the RISC-V RV32I M-extension. Along with the multiplier, also an adder is described taking as input the value from multiplier output, and, optionally, the value from the ALU unit output register which is used to keep partial results of the actual operation being computed. In this way, the same register used to keep the result in other ALU operations, is exploited here to accumulate the result over time. With this structure single 32 bit operations can be split into smaller 16 bit operations and can be computed in more than one clock cycle. For this reason, the result of a MUL instruction, the one returning the lower 32 bits of the 64 bit result of a 32 bit multiplication, is computed in three clock cycles; while for MULH, MULHU and MULHSU instructions, the ones returning the upper 32 bits of the 64 bit result, are computed in four clock cycles. Cycles performed by the multiplier are visually represented in Figure 3.1. In Figure 3.1b the sign-extension part is made necessary only for signed operations. A structure of this multiplier is depicted in Figure 3.2 where the flow starting from input signals A and B going to the output OUT is made clear. At each step the right input, already sign extended or zero extended, is selected by the control signals of the multiplexers providing this selection to the MUL unit. MUL unit result is then connected to the 33 bit

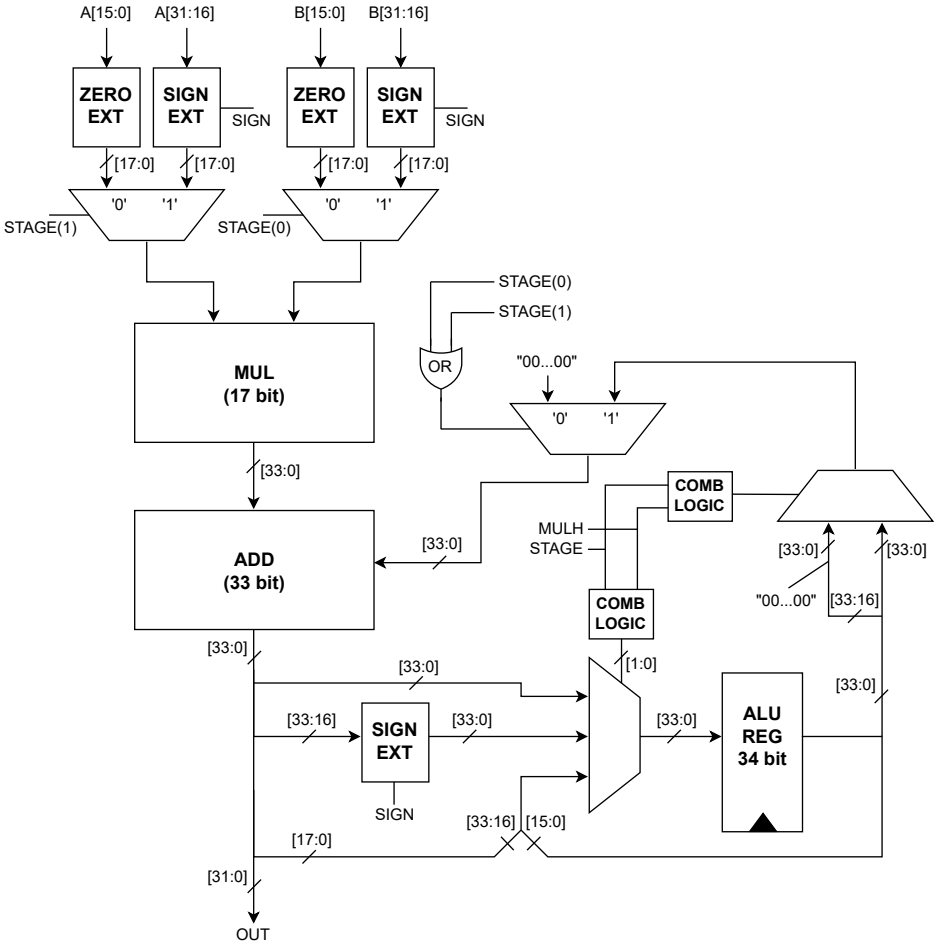
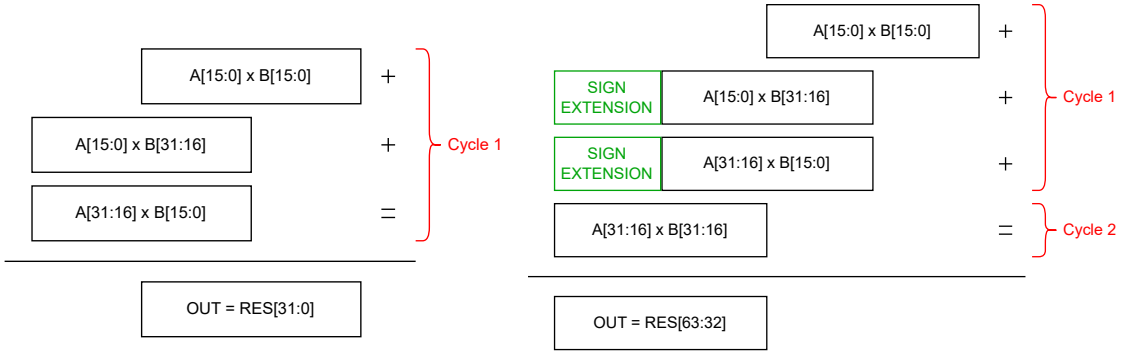


Figure 3.2: High-level circuit diagram of the *Fast* Ibex multiplier.

ADD unit which accumulate it exploiting the ALU register storing partial result of the previous operation. This process is repeated until the actual computation is completed according to steps in Figure 3.1a and Figure 3.1b. Selection of the value to be stored in the ALU register, and the value to be provided to the adder, is done exploiting two multiplexers driven by a minimal combinational logic depending on the actual stage and the type of operation performed. Reasons behind the choice of a single multiplier structure on 17 bits, instead of a more common one on 16 bit, come simply from the fact an unsigned operand on 16 bit can be treated as a signed operand on 17 bit. This allows having a simpler multiplier for all operation cases, that are, signed×signed, unsigned×unsigned and signed×unsigned, which are all treated as signed×signed, extending the sign such that the extension of an unsigned number is done adding a 0 as MSB value.

3.1.2 The 32 bit *SingleCycle* multiplier

A similar approach has been followed to describe the *SingleCycle* version. Internally it includes three 17 bit multipliers (for the same reason previously exposed), which compute the three partial products necessary in order to complete in a single cycle the MUL operation, as shown in Figure 3.3a. For MULH, MULHU and MULHSU instructions, as visible in Figure 3.3b, two clock cycles are needed to get the final result. The second change is in the adder structure, that is now requested to add three terms instead of two. Previously done description is still valid, indeed, the accumulation mechanism to split computation in different cycles exploiting the ALU register is the same, but this time there are fewer multiple connections to be taken into account due to the lower number of cycles performed.



(a) MUL instruction internal cycles.

(b) MULH, MULHU and MULHSU instructions internal cycles.

Figure 3.3: *SingleCycle* multiplier, 32 bit MUL operations splitting procedure.

Moreover, this architecture, in common with the *Fast* variation, is not designed to keep partial result of the actually computed MUL instruction and check if first cycles can be omitted in the case the near following instruction is MULH, MULHU or MULSHU operating on the same set of registers, as suggested in the RISC-V documentation [54]. For this reason the sequence to compute both the lower part and the upper part of a multiplication takes $1 + 2 = 3$ clock cycles. In the *Fast* version this is even more evident since the number of clock cycles become $3 + 4 = 7$ clock cycles instead of just 4. Even if this possibility could be an advantage in some algorithm, in defining the following multiplier variations this phenomenon is not counteracted, first to not add extra logic with respect to the strictly necessary modifications, secondly because in the QNN domain very rarely all 64 bit result of a 32 bit multiplication is needed.

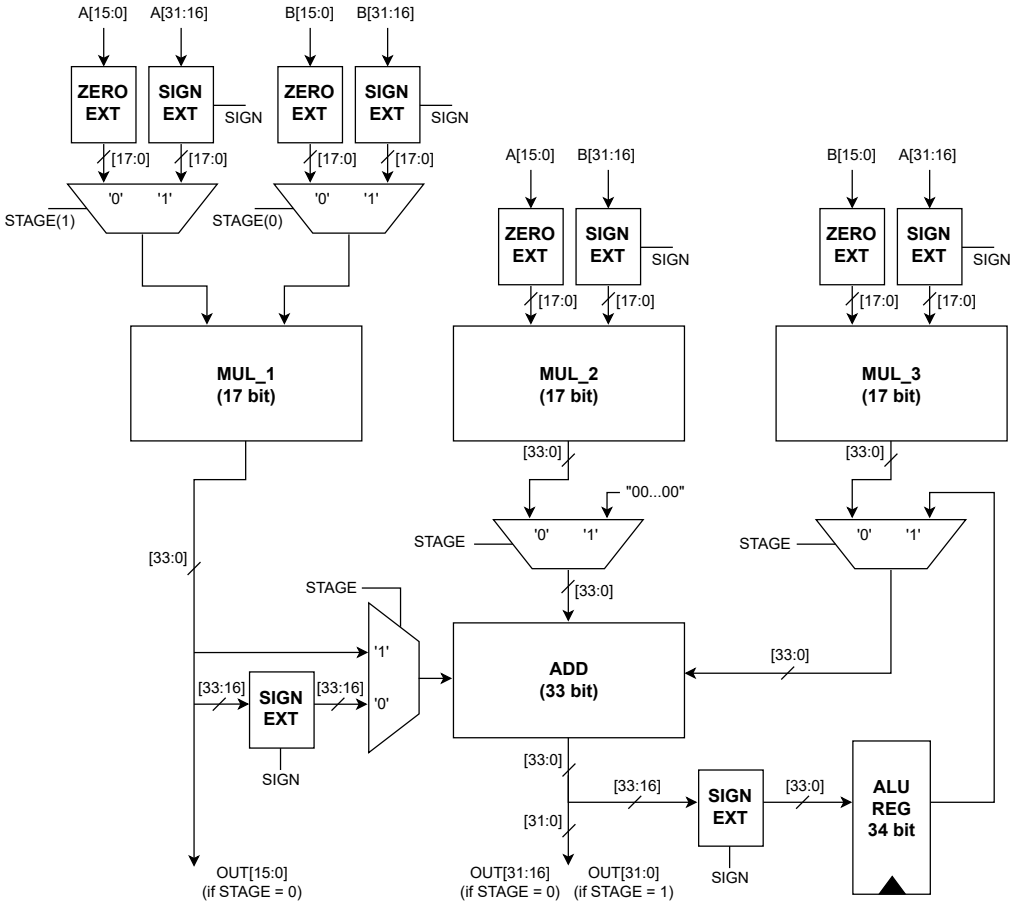


Figure 3.4: High-level circuit diagram of the *SingleCycle* Ibex multiplier.

3.2 Target modifications

First possible way to define a HW support for SS and ST operations is to define a behavioral description following the same approach used in the definition of two multiplier units originally present in the Ibex core, which have been presented in the previous paragraph. Here, in the first two subsections, both the *Fast* and the *SingleCycle* behavioral multiplier variants, adapted to parallel execution, are described. Last two subsections present the gate-level description, of both *Fast* and *SingleCycle* multiplier architectures, using the PS Baugh-Wooley structure derived in Section 2.3. All multiplier versions should agree on input format to be able to freely change the multiplier type without making changes to assembly-level RISC-V instructions, nor to software routines, both introduced in next chapters. Following the already known behavior of a Baugh-Wooley multiplier concerning input/output relation in SS and ST modes we can fix the input format to be the same also for the two behaviorally described counterparts and just mimic the gate-level description.

In this view, we know from Section 2.2 that the Baugh-Wooley results depend on the plain product of inputs for SS operations and on the *cross-product* of inputs for ST operations. For a reason explained only in Section 3.2.4, also the SS operation is implemented in a way similar to a *cross-product*. As a clarifying example, relations between result and operands for the case of an 8 bit instructions in both SS and ST follows:

- For the case of an SS operation on 8 bit operands:

$$\begin{aligned} Res[15:0] &= A[7:0] \times B[23:16], \\ Res[31:16] &= A[15:8] \times B[31:24], \\ Res[47:32] &= A[23:16] \times B[7:0], \\ Res[63:48] &= A[31:24] \times B[15:8] \end{aligned} \tag{3.1}$$

- For the case of an ST operation on 8 bit operands:

$$\begin{aligned} Res[42:24] = A[7:0] \times B[31:24] + A[15:8] \times B[23:16] \\ + A[23:16] \times B[15:8] + A[31:24] \times B[7:0] \end{aligned} \tag{3.2}$$

Moreover, an internal 64 bit register is added to keep time-by-time results of MAC operations. As discussed in the introductory part, MAC operations are such that the internal register value can be set by a specific RISC-V assembly instruction, and it accumulates over time until manually set again. Stored value is dependent on the type and precision of instruction that is looping on, thus there is no guarantee in the behavior of instruction mixing, nor with non-compliant initial set values. A single 64 bit register can keep results up to the following precision for each operation:

- Standard 32 bit operations: up to 64 bit.
- SS 16 bit operations: up to 32 bit per parallel result. Each value is maintained and added to the same position at each iteration cycle.
- SS 8 bit operations: up to 16 bit per parallel result. Each value is maintained and added to the same position at each iteration cycle.
- SS 4 bit operations: up to 8 bit per parallel result. Each value is maintained and added to the same position at each iteration cycle.
- ST 16 bit operations: up to 32 bit.
- ST 8 bit operations: up to 24 bit.
- ST 4 bit operations: up to 20 bit.

For all ST operations, we need to make sure the result is shifted by the right amount of bits before returning. Shift amount for the *Fast* version is equal to 8 bit for an 8 bit ST operation, and 12 bit for a 4 bit ST operation, no shift is necessary for 16 bit ST operations, while for the *SingleCycle* version it is equal to 16 bit for a 16 bit ST operation, 24 bit for an 8 bit ST operation, and 28 bit for a 4 bit ST operation. Result is shifted back only when exposed in output, internally the MAC register store non-shifted results to ease subsequent computation.
In all structures, the assembly operation to set internally a value in the MAC register is resolved in a single clock cycle since signal path have just to pass in a single multiplexer.

3.2.1 Precision scalable *Fast* behavioral multiplier

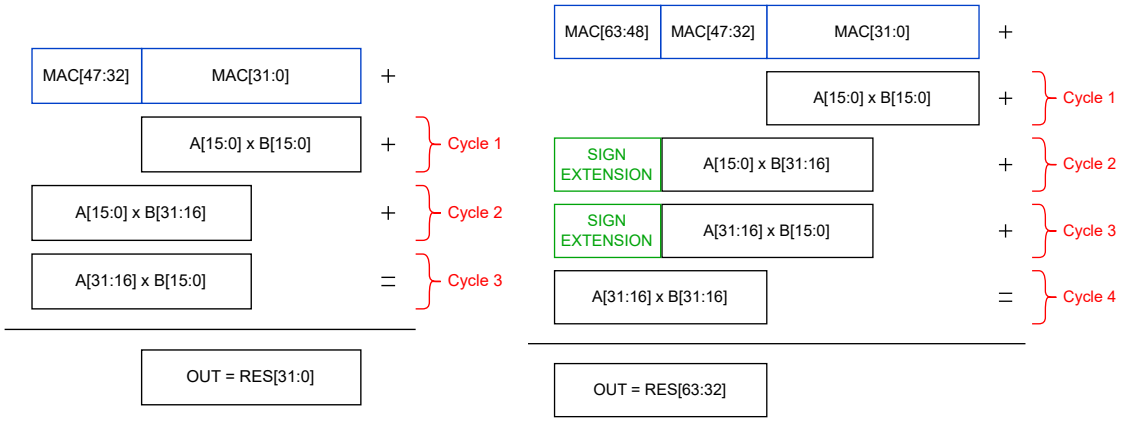
Modifications to the original *Fast* structure involves the definition of an additional Verilog *always comb* block managing the multiplication behavioral operator $*$ in a bit-oriented selection done with a *unique case* switch over the control signal. This control signal also selects the right output/operands relation depending on the instructions in execution (i.e. the one decoded in the same clock cycle). This kind of approach is presented in Listing 3.1, the 4-bit case have been omitted, given that it follows the same definition process of the 8-bit case, thus the extension is trivial.

By looking at Figure 3.6, to have minimal variations with respect to the non-PS version in Figure 3.2, the behaviorally described MUL structure is changed in a behaviorally described MAC structure. For the same reason, bit-width of this MAC structure is kept on 17 bit exploiting, as before, the extension of all unsigned numbers to signed numbers on 17 bit by appending a 0 as MSB. This approach has not been adopted in the reduced precision scenario due to the simple fact, for parallel execution involving the least precision supported operation, equal to 8 operation on 4 bit precision, it would end adding a total of 8 bit along all sub-multipliers (one for every operation) instead of just one bit for a single 16 bit multiplier, thus, adding too much complexity in the final structure. For this reason, in all SS and ST operations a distinction is also made between signed and unsigned numbers, casting properly the result, and allowing the synthesis tool to select the best multiplier structure to deal with this requirement in the internal CFG.

As previously mentioned this structure make use, conceptually of a single multiplier, meaning that, standard 32 bit multiplications, including MAC instructions, are computed in three clock cycles for the least 32 bit significant part, and in four clock cycles for the most 32 bit significant part respectively; while all SS/ST instructions, including the ones with MAC capabilities, are computed in two clock cycles. The MAC support is visible in the left feedback-branch in the architecture of Figure 3.6, where the internal MAC register, **MAC REG**, is visible. The

Listing 3.1: System Verilog behavioral description of the *Fast* multiplier.

```
always_comb begin
      // Default operation - MUL
      mult_res = $signed(mult_acc) + ($signed(mult_op_a) * $signed(mult_op_b));
      mult_res_uns = $unsigned(mult_res);

      unique case (operator_ext_i)

        MD_OP_MUL16SS: begin  // 16-bit SS mode
          if (signed_mode_i[0]) begin
            mult_res_uns[31:0] = $signed(mult_acc[31:0])
                               + ($signed(mult_op_a[15:0]) * $signed(mult_op_b[15:0]));
          end else begin
            mult_res_uns[31:0] = $unsigned(mult_acc[31:0])
                               + ($unsigned(mult_op_a[15:0]) * $unsigned(mult_op_b[15:0]));
          end
        end

        MD_OP_MUL8SS: begin  // 8-bit SS mode
          if (signed_mode_i[0]) begin
            mult_res_uns[15:0]  = $signed(mult_acc[15:0])
                                + ($signed(mult_op_a[7:0])  * $signed(mult_op_b[7:0]));
            mult_res_uns[31:16] = $signed(mult_acc[31:16])
                                + ($signed(mult_op_a[15:8]) * $signed(mult_op_b[15:8]));
          end else begin
            mult_res_uns[15:0]  = $unsigned(mult_acc[15:0])
                                + ($unsigned(mult_op_a[7:0])  * $unsigned(mult_op_b[7:0]));
            mult_res_uns[31:16] = $unsigned(mult_acc[31:16])
                                + ($unsigned(mult_op_a[15:8]) * $unsigned(mult_op_b[15:8]));
          end
        end

        MD_OP_MUL4SS: begin  // 4-bit SS mode
          if (signed_mode_i[0]) begin
            ...
          end else begin
            ...
          end
        end

        MD_OP_MUL16ST: begin  // 16-bit ST mode
          if (signed_mode_i[0]) begin
            mult_res_uns[31:0]  = $signed(mult_acc[31:0])
                                + ($signed(mult_op_a[15:0]) * $signed(mult_op_b[15:0]));
          end else begin
            mult_res_uns[31:0]  = $unsigned(mult_acc[31:0])
                                + ($unsigned(mult_op_a[15:0]) * $unsigned(mult_op_b[15:0]));
          end
        end

        MD_OP_MUL8ST: begin // 8-bit ST mode
          if (signed_mode_i[0]) begin
            mult_res_uns[23:0]  = $signed(mult_acc[23:0])
                                + (($signed(mult_op_a[7:0]) * $signed(mult_op_b[15:8]))
                                +  ($signed(mult_op_a[15:8]) * $signed(mult_op_b[7:0])));
            mult_res_uns[31:24] = {8{mult_res_uns[23]}};
          end else begin
            mult_res_uns[23:0]  = $unsigned(mult_acc[23:0])
                                + (($unsigned(mult_op_a[7:0]) * $unsigned(mult_op_b[15:8]))
                                +  ($unsigned(mult_op_a[15:8]) * $unsigned(mult_op_b[7:0])));
            mult_res_uns[31:24] = 8'b0;
          end
        end

        MD_OP_MUL4ST: begin  // 4-bit ST mode
          if (signed_mode_i[0]) begin
            ...
          end else begin
           ...
          end
        end

        default: ;
      endcase
    end
```



(a) MUL instruction internal cycles.

(b) MULH, MULHU and MULHSU instructions internal cycles.

Figure 3.5: *Fast* multiplier, 32 bit MAC operations splitting procedure.

first input of the MAC register is the actually computed value presented in output, passing along a conceptual unit called *Shift and Mask* that simply shifts the result placing it in the correct position of the 64 bit MAC register, depending on the actual stage, masking it with the value present in the MAC register itself for sub-words that must be kept frozen. This structure is behaviorally described in Verilog, trivially, as some signal assignments as follows:

- Cycle 1:
$$\begin{aligned} MAC\_REG_{IN}[15:0] &= ADD_{OUT}[15:0], \\ MAC\_REG_{IN}[63:16] &= MAC\_REG_{OUT}[63:16] \end{aligned} \tag{3.3}$$
- Cycle 2:
$$MAC\_REG_{IN}[63:0] = MAC\_REG_{OUT}[63:0] \tag{3.4}$$
- Cycle 3:
$$\begin{aligned} MAC\_REG_{IN}[15:0] &= MAC\_REG_{OUT}[15:0], \\ MAC\_REG_{IN}[31:16] &= ADD_{OUT}[15:0], \\ MAC\_REG_{IN}[63:32] &= MAC\_REG_{OUT}[63:32] \end{aligned} \tag{3.5}$$
- Cycle 4:
$$\begin{aligned} MAC\_REG_{IN}[31:0] &= MAC\_REG_{OUT}[31:0], \\ MAC\_REG_{IN}[63:32] &= ADD_{OUT}[31:0] \end{aligned} \tag{3.6}$$

Where MAC_REG_{IN} is the input of the MAC register, MAC_REG_{OUT} is the output of the MAC register, while ADD_{OUT} is the current output exposed by the internal adder. The other input signals to the MAC register is the input operand

concatenation, i.e. the *OP Concat* unit in the figure, used by the MACSET assembly instruction to set the initial MAC register value. The last input of the MAC register is directly the current output exposed by that register, this is done to preserve the internal value in the case of other instructions, rather than MAC instructions, being executed in a computational loop, allowing a little more flexibility in the code routines definition.

Finally, the selection of the input to the MAC unit is done with the same strategy of the minimum logic inferred in the synthesis tool which here I represent as a combinational logic block controlled by some control signals.

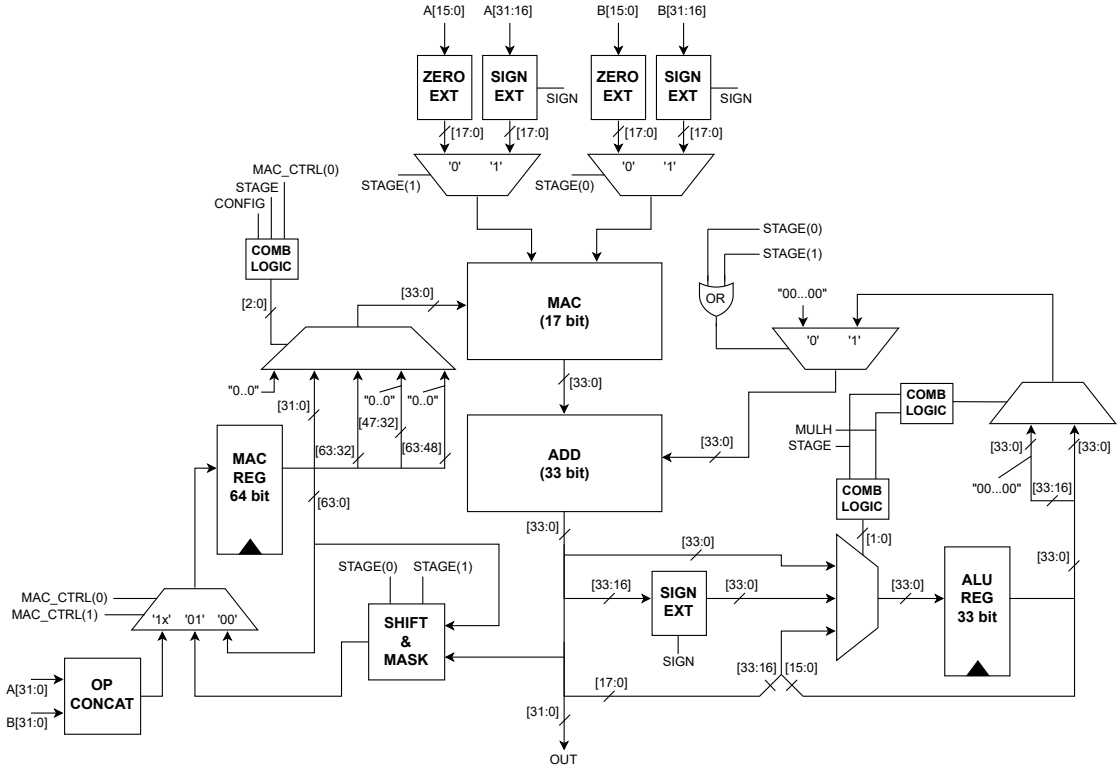
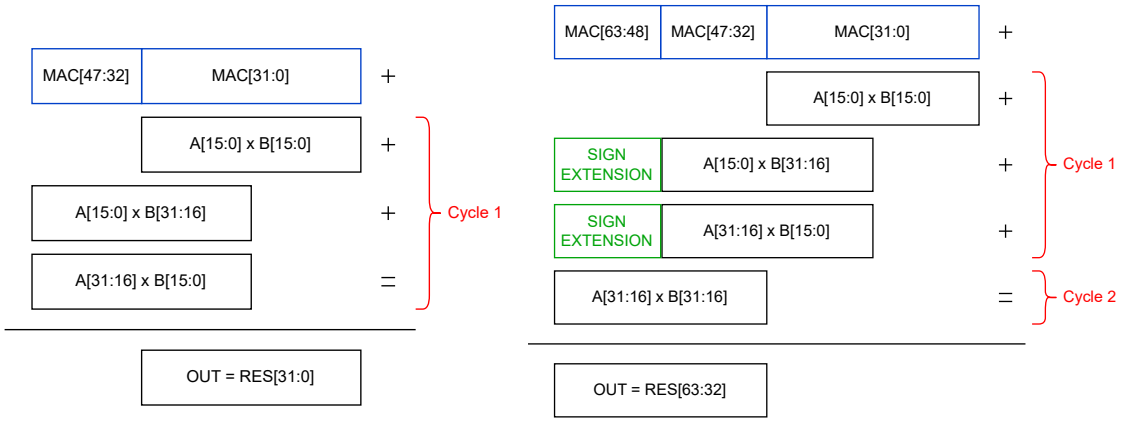


Figure 3.6: High-level circuit diagram of the behavioral PS *Fast* multiplier unit.

3.2.2 Precision scalable *SingleCycle* behavioral multiplier

As per the *SingleCycle* variant, this follows the same changes implemented for the *Fast* one. The Verilog definitions are presented in Listing 3.2, in this case I decided to omit also the 16 bit ST case, because it is equal to the 16 bit SS case, while as for the 4 bit SS/ST cases these can be trivially extended by the reader from the 8 bit ones. The only relevant difference in this structure is in the usage of two MAC units and a MUL unit, instead of a single MAC unit, allowing completing



(a) MUL instruction internal cycles.

(b) MULH, MULHU and MULHSU instructions internal cycles.

Figure 3.7: *SingleCycle* multiplier, 32 bit MAC operations splitting procedure.

all instructions in one clock cycle except the ones having to return the upper 32 bit of a MUL/MAC 32 bit operation, which are computed in two clock cycles. Since, to complete all SS/ST instructions in a single clock cycle, just two multipliers are strictly required, the usage of the two multipliers is done trying minimizing the signal re-assignment to keep the number of multiplexers as low as possible. Given that, the left-branch, regarding the MAC accumulation procedure over time, is split in two paths going respectively in MAC_1 and MAC_2, where the feedback to sum the value in the ALU output register is instead placed in the MUL_3 branch. The *OP Concat* block is the same as for the *Fast* variation, while the *Shift and Mask* now deals with a reduced number of stages and assign the signal as follows:

- Cycle 1:

$$\begin{aligned} MAC\_REG_{IN}[15:0] &= MAC_{1\_OUT}[15:0], \\ MAC\_REG_{IN}[31:16] &= ADD_{OUT}[15:0], \\ MAC\_REG_{IN}[63:32] &= MAC\_REG_{OUT}[63:32] \end{aligned} \tag{3.7}$$

- Cycle 2:

$$\begin{aligned} MAC\_REG_{IN}[32:0] &= MAC\_REG_{OUT}[32:0], \\ MAC\_REG_{IN}[63:32] &= ADD_{OUT}[31:0] \end{aligned} \tag{3.8}$$

The final structure is shown in Figure 3.8.

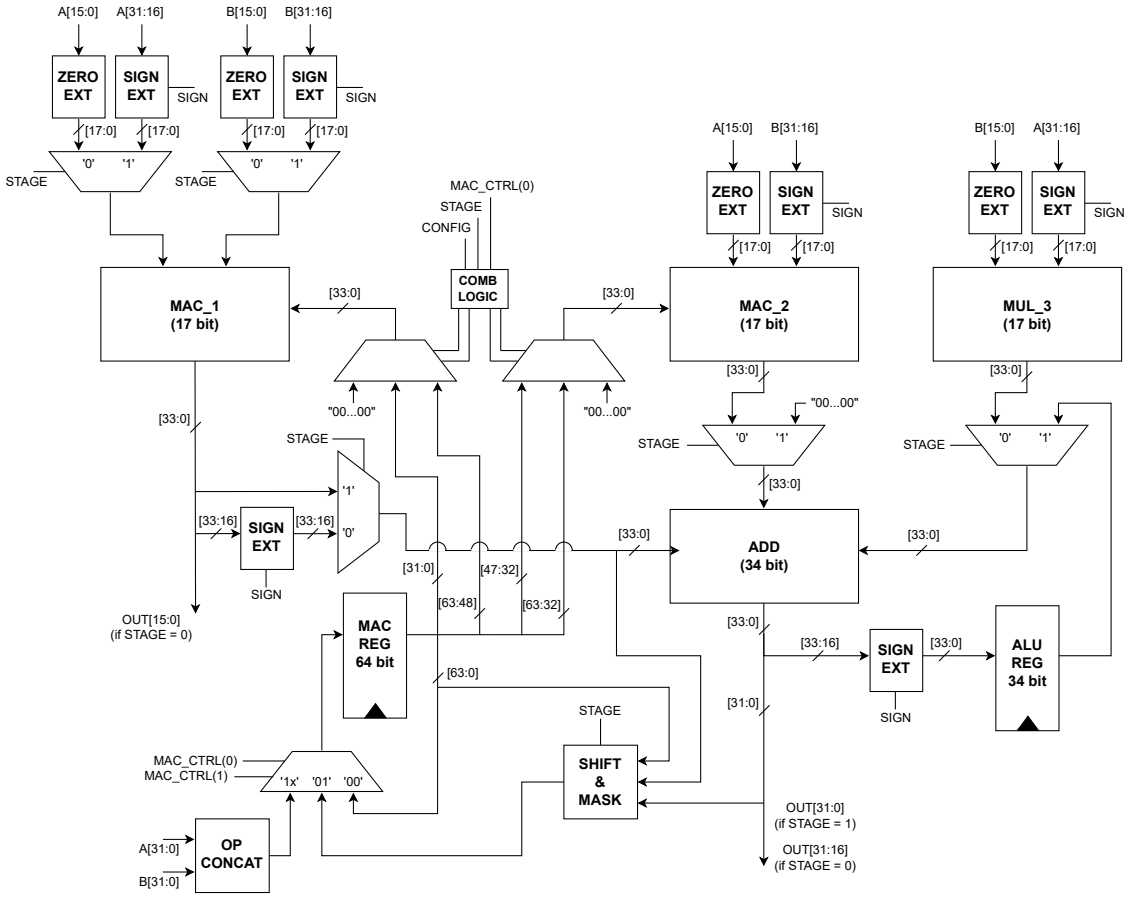


Figure 3.8: High-level circuit diagram of the behavioral PS *SingleCycle* multiplier unit.

Listing 3.2: System Verilog behavioral description of the *SingleCycle* multiplier.

```
always_comb begin
      // Default operation - MUL
      mult1_res = $signed(mult1_acc) + ($signed(mult1_op_a) * $signed(mult1_op_b));
      mult1_res_uns = $unsigned(mult1_res);
      mult2_res = $signed(mult2_acc) + ($signed(mult2_op_a) * $signed(mult2_op_b));
      mult2_res_uns = $unsigned(mult2_res);
      mult3_res = $signed(mult3_op_a) * $signed(mult3_op_b);
      mult3_res_uns = $unsigned(mult3_res);

      unique case (operator_ext_i)

        MD_OP_MUL16SS: begin  // 16-bit SS mode
          if (signed_mode_i[0]) begin
            mult1_res_uns[31:0] = $signed(int_accum[31:0]) +
                                  ($signed(mult1_op_a[15:0]) * $signed(mult1_op_b[15:0]));
            mult2_res_uns[31:0] = $signed(int_accum[63:32]) +
                                  ($signed(mult2_op_a[15:0]) * $signed(mult2_op_b[15:0]));
          end else begin
            mult1_res_uns[31:0] = $unsigned(int_accum[31:0]) +
                                  ($unsigned(mult1_op_a[15:0]) * $unsigned(mult1_op_b[15:0]));
            mult2_res_uns[31:0] = $unsigned(int_accum[63:32]) +
                                  ($unsigned(mult2_op_a[15:0]) * $unsigned(mult2_op_b[15:0]));
          end
        end

        MD_OP_MUL8SS: begin  // 8-bit SS mode
          if (signed_mode_i[0]) begin
            mult1_res_uns[15:0]  = $signed(int_accum[15:0]) +
                                   ($signed(mult1_op_a[7:0])  * $signed(mult1_op_b[7:0]));
            mult1_res_uns[31:16] = $signed(int_accum[31:16]) +
                                   ($signed(mult1_op_a[15:8]) * $signed(mult1_op_b[15:8]));
            mult2_res_uns[15:0]  = $signed(int_accum[47:32]) +
                                   ($signed(mult2_op_a[7:0])  * $signed(mult2_op_b[7:0]));
            mult2_res_uns[31:16] = $signed(int_accum[63:48])+
                                   ($signed(mult2_op_a[15:8]) * $signed(mult2_op_b[15:8]));
          end else begin
            mult1_res_uns[15:0]  = $unsigned(int_accum[15:0]) +
                                   ($unsigned(mult1_op_a[7:0])  * $unsigned(mult1_op_b[7:0]));
            mult1_res_uns[31:16] = $unsigned(int_accum[31:16]) +
                                   ($unsigned(mult1_op_a[15:8]) * $unsigned(mult1_op_b[15:8]));
            mult2_res_uns[15:0]  = $unsigned(int_accum[47:32]) +
                                   ($unsigned(mult2_op_a[7:0])  * $unsigned(mult2_op_b[7:0]));
            mult2_res_uns[31:16] = $unsigned(int_accum[63:48]) +
                                   ($unsigned(mult2_op_a[15:8]) * $unsigned(mult2_op_b[15:8]));
          end
        end

        MD_OP_MUL4SS: begin  // 4-bit SS mode
          ...
        end

        MD_OP_MUL16ST: begin  // 16-bit ST mode
          ...
        end

        MD_OP_MUL8ST: begin  // 8-bit ST mode
          if (signed_mode_i[0]) begin
            mult1_res_uns[23:0]  = $signed(int_accum[23:0]) +
                                   (($signed(mult1_op_a[7:0]) * $signed(mult1_op_b[15:8])) +
                                    ($signed(mult1_op_a[15:8]) * $signed(mult1_op_b[7:0])));
            mult2_res_uns[23:0]  = ($signed(mult2_op_a[7:0]) * $signed(mult2_op_b[15:8])) +
                                    ($signed(mult2_op_a[15:8]) * $signed(mult2_op_b[7:0]));
          end else begin
            mult1_res_uns[23:0]  = $unsigned(int_accum[23:0]) +
                                  (($unsigned(mult1_op_a[7:0]) * $unsigned(mult1_op_b[15:8])) +
                                   ($unsigned(mult1_op_a[15:8]) * $unsigned(mult1_op_b[7:0])));
            mult2_res_uns[23:0]  = ($unsigned(mult2_op_a[7:0]) * $unsigned(mult2_op_b[15:8])) +
                                   ($unsigned(mult2_op_a[15:8]) * $unsigned(mult2_op_b[7:0]));
          end
        end

        MD_OP_MUL4ST: begin  // 4-bit ST mode
          ...
        end

        default: ;
      endcase
    end
```

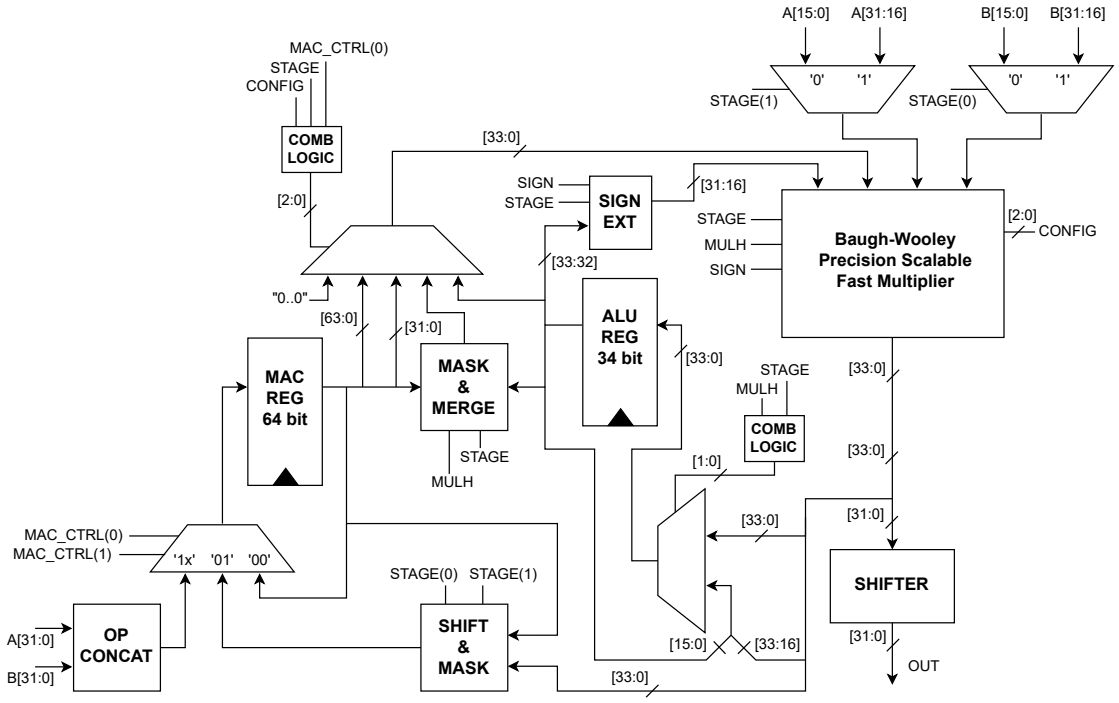


Figure 3.9: High-level circuit diagram of the gate-level PS *Fast* multiplier unit.

3.2.3 Precision scalable *Fast* gate-level multiplier

Gate-level Baugh-Wooley structure, used inside the gate-level version of the *Fast* Ibex multiplier unit, is similar to the final one derived in Section 2.3 and visible in Figure 2.18. Unfortunately, this structure is not ready to be used to compute 32 bit operations. When considering the computing paradigm adopted by the original *Fast* Ibex multiplier, sightly modified to include also the signal coming from the internal MAC register, reported in Figure 3.5, the sign extension green part depends on results produced in *Cycle 2* and *Cycle 3*. For this reason sign-extensions must be sum all together in *Cycle 4*, meaning we need to pre-compute it along cycles. Moreover, the structure of Figure 2.18 does not have any other input available on the left side where the sign-extension of previous layers should take place.

This last consideration leads to the first modification of the structure, that is, to add a computational column formed of plain FAs to then perform the sign-extension addition in parallel to the computation of *Cycle 4*; blocks added to accomplish this modification are the ones in orange visible in Figure 3.10. Each of these blocks can have as input a direct control signal making them able to sum a series of 1*s* along the column. These blocks are also used in *Cycle 2* to add the 33rd bit coming from operation performed in *Cycle 1*. Not only, from Figure 3.5b, in *Cycle 3*, the summation of the MAC input with the result of *Cycle 1* and *Cycle 2*, needs a bit-width

of 34 bit in order not to have an overflow in the result. Due to this, some FAs and an HA have to be added to extend the result up to the 33rd bit. These are the ones marked in red, in the bottom-left corner of Figure 3.10.

Now, the derived structure can be used to compute all set of operations needed. A high level view of the architecture is presented in Figure 3.9 where signal connections can be easily retrieved. This structure is similar to the behavioral one, both have the same left-branch to save the MAC result over time. Some differences are in the value saved in the ALU register, in which an embedded sign-extension and shift is not needed anymore since it is operated by the orange FA-column.

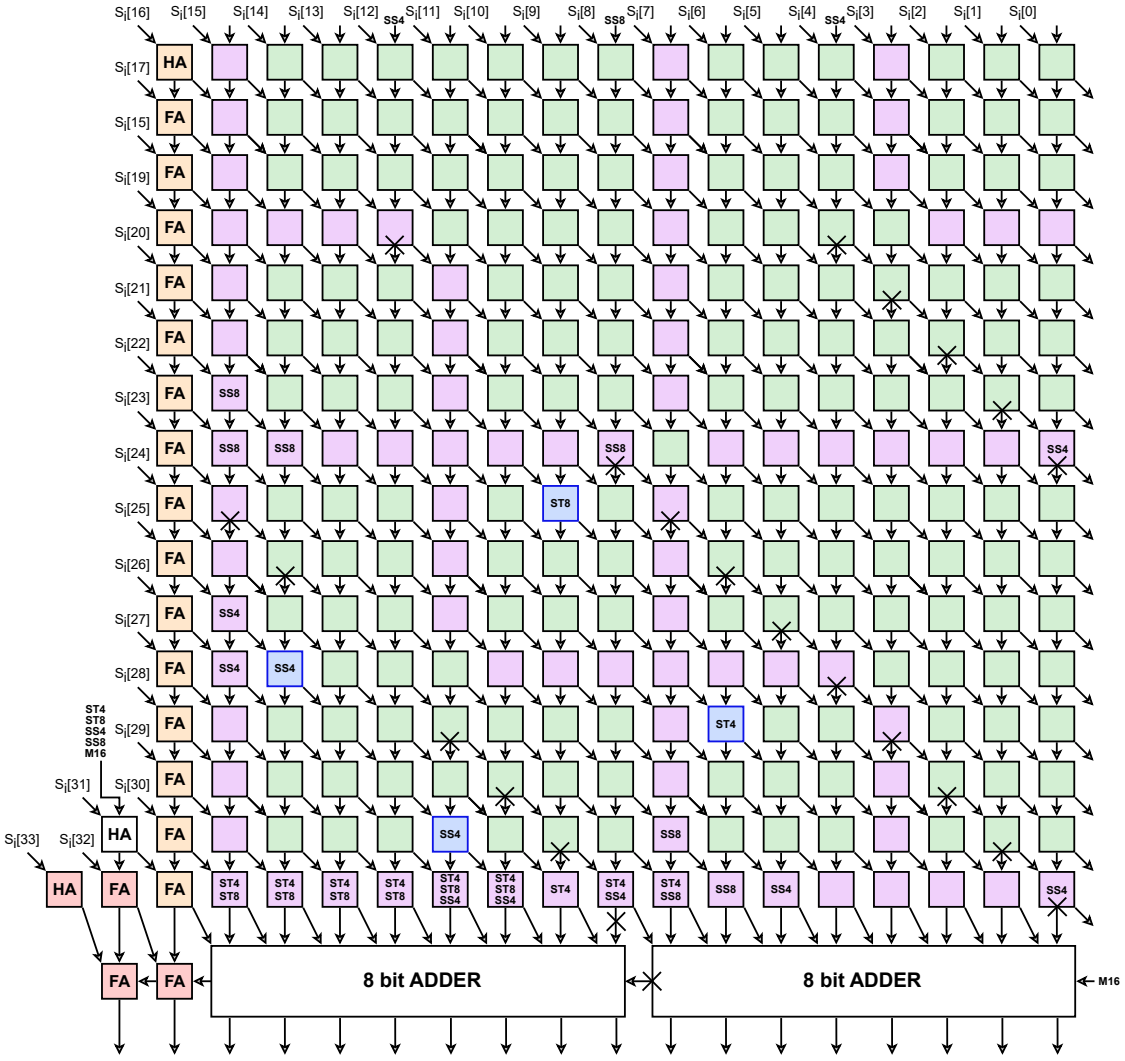


Figure 3.10: Structure of the gate-level Baugh-Wooley PS variant included in the *Fast* multiplier unit.

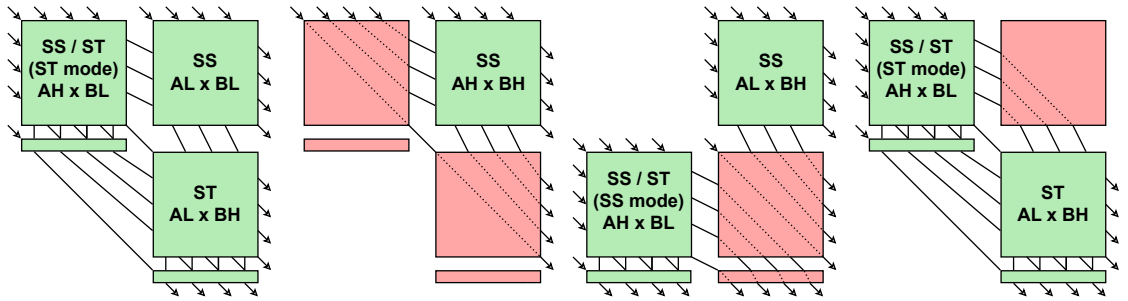
Moreover, another difference is the value provided to the BW structure to accumulate the result, where, considering the overlapping part between the ALU register, still used to save temporary values between steps, and considering the MAC register content, as already stated, we can exploit the orange FA-column at *Cycle 2*, making possible, at the same time, to perform the required summation with a single common signal also in the case of MAC operations.

To manage this, a unit called *Mask and Merge* is placed in the design. As other cases this unit is just a wrapper of some signal assignments done in the Verilog description.

- For 32-bit operations returning the 32 LSBs of the result:
  - Cycle 1:
$$\begin{aligned} OUT[31:0] &= MAC\_REG_{OUT}[31:0], \\ OUT[33:32] &= 0 \end{aligned} \tag{3.9}$$
  - Cycle 2:
$$OUT[15:0] = ALU\_REG_{OUT}[31:16] \tag{3.10}$$
  - Cycle 3:
$$OUT[15:0] = ALU\_REG_{OUT}[31:16] \tag{3.11}$$
- For 32-bit operations returning the 32 MSBs of the result:
  - Cycle 1:
$$\begin{aligned} OUT[31:0] &= MAC\_REG_{OUT}[31:0], \\ OUT[33:32] &= 0 \end{aligned} \tag{3.12}$$
  - Cycle 2:
$$\begin{aligned} OUT[15:0] &= ALU\_REG_{OUT}[31:16], \\ OUT[31:16] &= MAC\_REG_{OUT}[47:32], \\ OUT[33:32] &= 0 \end{aligned} \tag{3.13}$$
  - Cycle 3:
$$OUT[33:0] = ALU\_REG_{OUT}[33:0] \tag{3.14}$$
  - Cycle 4:
$$\begin{aligned} OUT[15:0] &= ALU\_REG_{OUT}[31:16], \\ OUT[31:16] &= MAC\_REG_{OUT}[63:48] \end{aligned} \tag{3.15}$$

From this description, to obtain the relation of the case where the operation to be performed is not a MAC operation, it is possible to just change equations, where a value coming from the MAC register is assigned to OUT, with an assignment to the logic value 0. Also, values which are not assigned are considered *don't care conditions* in the design, meaning that, new signal assignments with respect to

default ones are not performed in order to save logic. The precise selection of signals is made through some minimal combinational logic, considering the current cycle and the operation type, which controls multiplexers. All other blocks and connections, with respect to the behavioral version, remain the same.



(a) 32 bit multiplication case, first stage.

(b) 32 bit multiplication case, second stage.

(c) Sub-word multiplication in SS mode.

(d) Sub-word multiplication in ST mode.

Figure 3.11: Schematic representation of signals flow in the *SingleCycle* architecture.

3.2.4 Precision scalable *SingleCycle* gate-level multiplier

In order to increase the number of multipliers while keeping the same Baugh-Wooley structure, having as a target to compute all operations in a single cycle except the ones requiring the most significant 32 bits of the 64 bit result, which are computed in two cycles, the structure have to change as follows. First, the structure makes use of three 16 bit multipliers. Moreover, recalling Paragraph 2.2, we saw that, as a rule, to perform 16 bit SS/ST operations, given a 32 bit Baugh-Wooley multiplier just two 16 bit multipliers are needed. Since in total, in a 32 bit Baugh-Wooley structure, we have four 16 bit sub-multipliers, to reduce the number to three, it means one of these have to perform both SS and ST operations taking the place of another, leading to the final three multipliers structure. The proposed solution is to connect one 16 bit sub-multiplier of the ones present in the structure to perform both SS and ST operations; we already derived this structure as the last one of in Paragraph 2.3, that is, the one of Figure 2.18. Other two structures need to perform only the SS or only the ST operations, also these variations have been derived in Paragraph 2.3 and can be seen in Figure 2.13 and Figure 2.15 respectively. Moreover, these must be fully deactivated in the case they are not contributing to a specific computation, thus final structure includes only green blocks (Figure 2.9b), purple blocks (Figure 2.12b), and blue blocks (Figure 2.17b).

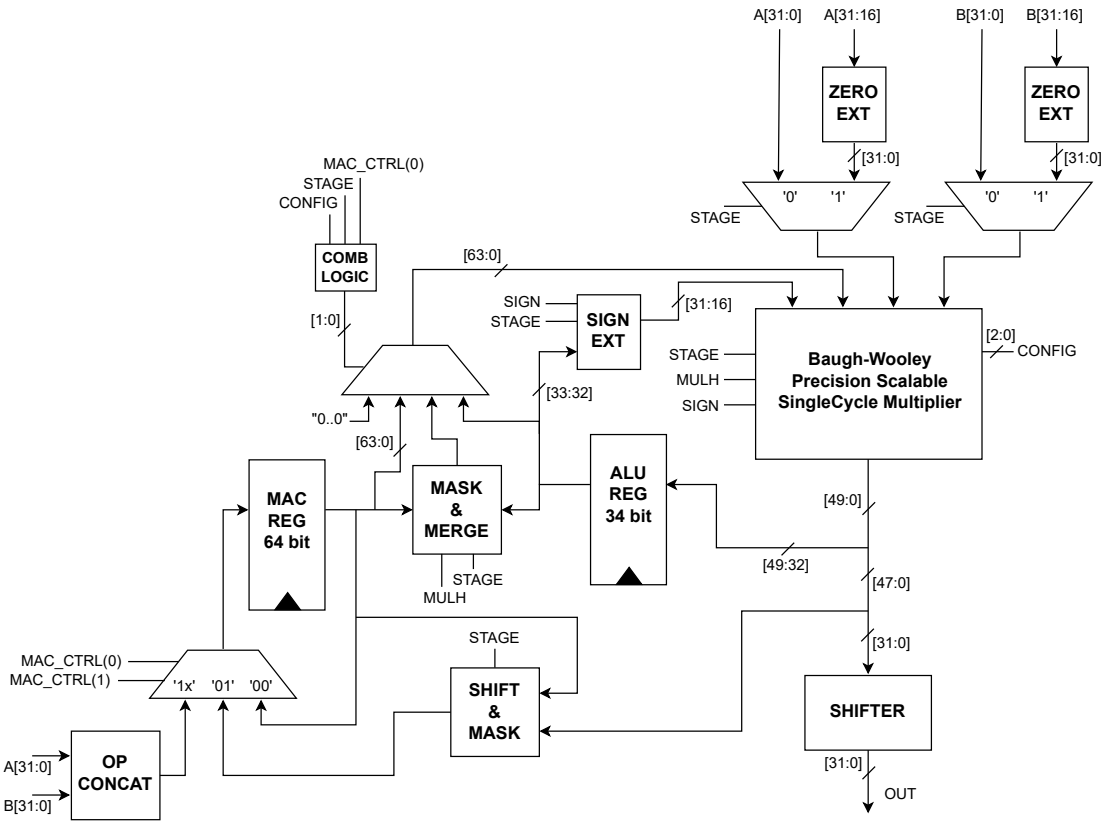


Figure 3.12: High-level circuit diagram of the gate-level PS *SingleCycle* multiplier unit.

The final scheme of this multiplier structure is visible in Figure 3.11 which also clarifies the signal connections between the three multiplier structures. In detail, for 32 bit operations the structure assumes the state depicted in Figure 3.11a and in Figure 3.11b, for the first and second cycle respectively, as illustrated in Figure 3.7b. During the first cycle all blocks have a green background, so they are all active; during the second cycle only the one in the top-right corner is active, with other blocks having a red background, meaning they do not contribute to computation. In Figure 3.11d the structure assumes a state compatible with the ST operations, while in Figure 3.11c it is made clear the alternative direct connection of the block in the top-left corner to the block in the bottom-right corner to work in parallel to compute SS operations. The top-left block has its own adder used to compute the sum bit to be added to the bottom-right block in ST configuration, the same adder is used to compute the result for SS operations having as carry-in signal the carry-out of the adder in the bottom-right corner. This usage maximizes the HW sharing among different configurations. Moreover, since to have this double behavior and double connection, the SS/ST structure have a higher complexity and

need multiplexers to route correctly the signals, input of this block is kept constant to $AH \times BL$, instead the SS block have signal reassignment to $AL \times BH$. This is the reason to have input/output relation involving a partial *cross-product* also in SS operations as previously anticipated.

The final gate-level structure is visible in Figure 3.13. Red HA/FA blocks in bottom position perform conceptually the same operations as the red ones in Figure 3.10: they allow computing partial results on 34 bit given computation requirements in Figure 3.7b. As per the sign-extension, this is computed in the very same way as the one adopted for the gate-level *Fast* case, only difference is the extra FA column is not necessary since in *Cycle 2* there is a whole 16 bit BW block deactivated, meaning that those blocks can generate the sign-extension if there is at least a blue block or a purple block along each diagonal (where the contribution to the output is the same). For diagonals in which this condition is not verified, a blue block has to take the place of a green block, in an arbitrary position. Also, from Figure 3.13 we can appreciate multiple positions in connecting the various 16 bit sub-multipliers is managed through the usage of multiplexers.

In Figure 3.12 the overall structure is visible. With respect to the *Fast* diagram in Figure 3.9, the only change is in signal parallelism and in the fewer cycles the multiplier is requested to perform. For this reason the *Mask and Merge* block changes its function accordingly, as follows:

- For 32-bit operations returning the 32 LSBs of the result:
  - Cycle 1:
$$OUT[31:0] = MAC\_REG_{OUT}[31:0] \tag{3.16}$$
- For 32-bit operations returning the 32 MSBs of the result:
  - Cycle 1:
$$OUT[47:0] = MAC\_REG_{OUT}[47:0] \tag{3.17}$$
  - Cycle 2:
$$\begin{aligned} OUT[15:0] &= ALU\_REG_{OUT}[31:16], \\ OUT[31:16] &= MAC\_REG_{OUT}[63:48] \end{aligned} \tag{3.18}$$

Also in this case to have the relation for non-MAC operations is possible to replace assignment to MAC register values with logical 0 values.

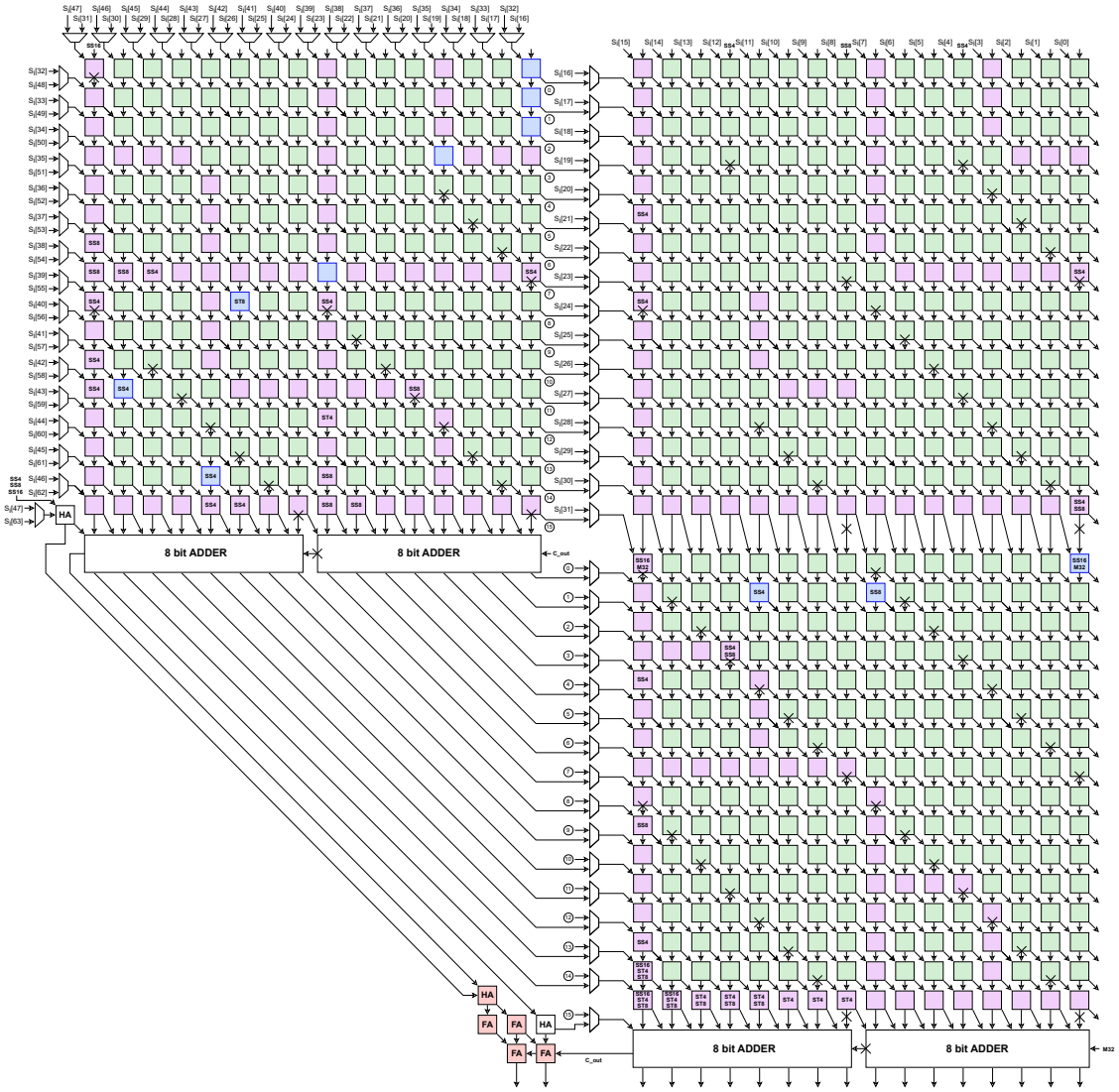


Figure 3.13: Structure of the gate-level Baugh-Wooley PS variant included in the *Fast* multiplier unit.

3.3 Synthesis results

Now that all multiplier units are derived, files containing behavioral and gate-level descriptions are included in the Ibex core adding some parameters to select the right multiplier structure from a top-level file, that is the starting point for the synthesis procedure.

Synthesis is carried out by Synopsys Design Compiler synthesis tool, targeting a 28 nm Fully-Depleted Silicon-On-Insulator (FDSOI) technology node at 0.9 V.

Synthesis results are reported in Figures 3.14 and 3.15 for *Fast* multipliers, and in Figure 3.16 and 3.17 for *SingleCycle* multipliers. Here each point is the result obtained defining a discrete target frequency for the design in a linear range of 10 steps between 100 MHz and 1 GHz. Lines present in the graphs are only to ease the reading process and have no physical meaning.

In Figure 3.14 we can see the area resulting from the synthesis process of the Ibex core, configured with the three version of the *Fast* multiplier unit. From the graph, the original *Fast* multiplier is the one having the smallest area occupation. This is an expected result, since this multiplier version have no support for the SS/ST nor MAC instructions. Between the two modified multipliers, the gate-level requires less area than the behavioral one, especially in low-frequencies. For frequencies where the gate level is synthesized without any negative slack, the area is always less than the behavioral version. This is true in the range from 100 MHz to 800 MHz. After this value of frequency, even defining a higher target frequency constraint for the synthesis tool, this is not able to correctly met that frequency thus the Ibex is stuck around 800 MHz. Given structure-independent properties in the behavioral description, for high frequencies the synthesis tool is able to select a proper multiplier structure to satisfy the design frequency constraint. This pattern for the gate-level is expected since, given its structure, the Baugh-Wooley is usually employed in low-frequency applications.
Moving now to the power consumption, in Figure 3.15, comparison between multiplier solutions shows the consumption is almost the same for low-frequencies, around 100-300 MHz. From 300 MHz, the curves start diverging. We have a lower power consumption for the gate-level with respect to the behavioral, until 700 MHz, since, after this frequency, the design of the gate level is not increasing in area, for this reason the consumption remains the same, and it overlaps the behavioral case. The behavioral description instead keeps growing trying to satisfy the target frequency constraint.

As per the *SingleCycle* multiplier versions, the area results of the synthesis process of the three Ibex cores, each one including one different multipliers, are shown in Figure 3.16. As before, the original *SingleCycle* multiplier is the one requiring less area. Near after this, for low frequencies, the gate-level is the better one, and this is true until 400 MHz. At 500 MHz the area of the gate-level version surpasses the behavioral version. Moreover, from this frequency, the synthesis tool is not able to met the target frequency for the gate-level multiplier, thus the design area remains almost the same. For the same reason as before, the behavioral version can keep increasing in area given that it tries to satisfy the design frequency constraint; this happens up to 900 MHz.

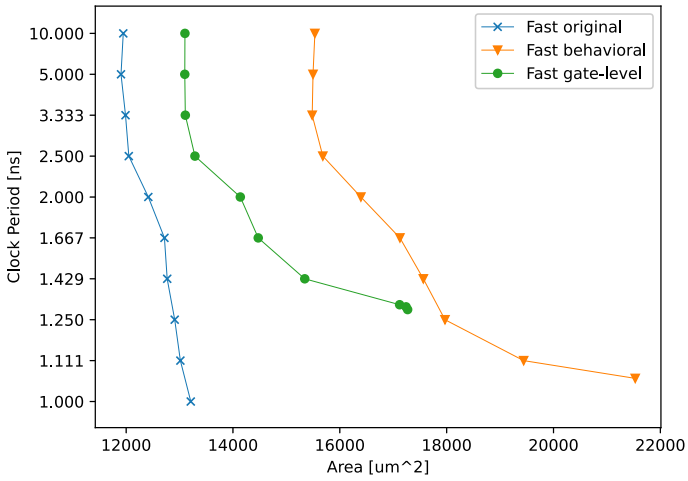


Figure 3.14: Area occupation synthesis results of the three Ibex cores *Fast*.

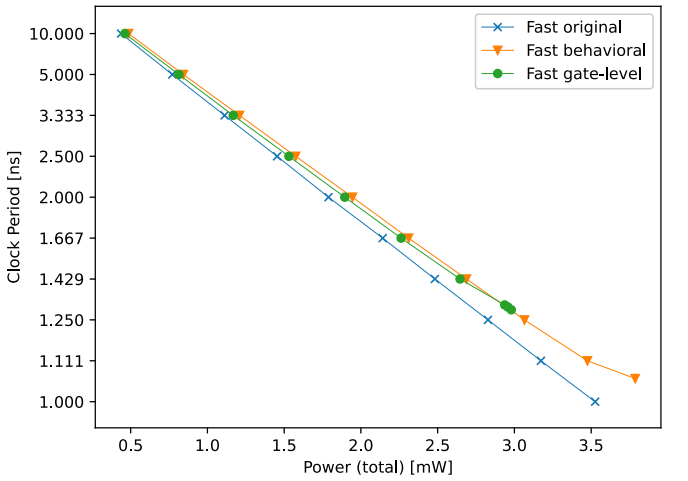


Figure 3.15: Power consumption synthesis results of the three Ibex cores *Fast*.

The power consumption reflects the area phenomena. At 100 MHz the consumption between versions is almost the same, but here the curves diverges faster than before. The gate-level variation consumes less power until 400 MHz are reached; at 500 MHz it consumes more than the behavioral version and, after that frequency it can not compete anymore. The behavioral one, instead, keeps increasing in power consumption up to 900 MHz. Also in this case these results are expected, the gate-level stop frequency is even lower than the *Fast* version due to the longer critical path present in this bigger Baugh-Wooley architecture.

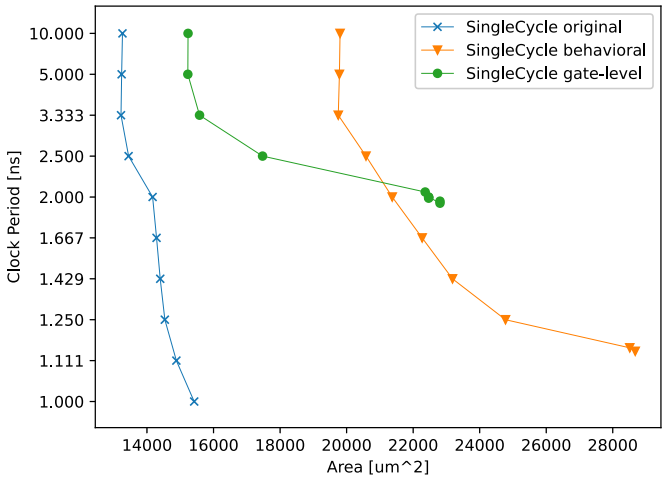


Figure 3.16: Area occupation synthesis results of the three Ibex cores *SingleCycle*.

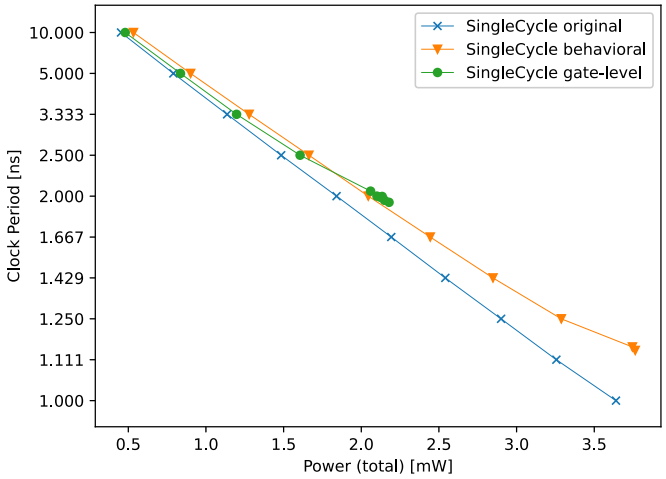


Figure 3.17: Power consumption synthesis results of the three Ibex cores *SingleCycle*.

Overall, as expected, versions including the SS/ST, and MAC operations support, show an increase in both area and power consumption. For low frequencies the gate-level versions performs better, given the typical application for the Baugh-Wooley multiplier is the low-frequency scenario. For higher frequencies the behavioral description approach gives a structure that is less optimized, but that can target a wider range of frequencies compared to the gate-level description.

Chapter 4

RISC-V ISA extension and GCC modifications

First paragraph of this chapter recalls the GNU GCC, while the second paragraph presents modifications done to GCC to make it able to recognize and compile assembly custom instructions.

The last section is devoted as a reference of all custom assembly instructions added to the RISC-V ISA, for which the multiplier units developed in previous chapters can optimize the execution.

4.1 GCC background

The GNU Compiler Collection (GCC) [50] is one of the most famous compiler for C/C++ language. Released in the late 80s it became the standard for C compilation on UNIX systems. It supports several HW architectures making use of its flexibility to embrace also recent and novel ISA, like RISC-V, building specialized compiler targeting all variations coming from ISA extensions or custom instructions, like the one required in this thesis work.
As a lightweight explanation, GCC is organized in a backend and a frontend.

The **frontend** is used to interface from a given language to a language-independent structure called "abstract syntax tree" that could be seen as a CFG-like representation of elementary instructions.

The **backend**, starting from that language-independent abstract syntax tree, deals with the selected HW architecture producing the final machine code to run.

The objective of this chapter is to add low-level assembly instructions derived from the desired ISA extension, thus operating on the assembler part of the backend to make it able to recognize and translate those instructions. With these modifications, GCC is not able to inference custom instructions starting directly from

the provided C-code, but it recognizes only assembly instructions. For this reason who writes the code has to place directly those assembly instructions, i.e. writing assembly routines, to make use of the ISA extension.
This approach is still a good choice considering the final objective is to test modifications done at microprocessor architectural level. Expansion supporting also more abstract C-like descriptions are left as a possible future work.

4.2 Adding custom instructions to GCC

The modification to allow recognizing new assembly instructions in GCC is done operating on just two files.

Given the GCC RISC-V toolchain source code (fully available under the GNU GPL license) [53], in the directory tree organization, the first file to modify is located at `riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h`
This is a header file containing definitions of all opcodes present in the RISC-V ISA. Specifically, each instruction is defined as follows. There are two definitions, one called **MASK** and one called **MATCH**. The MASK definition is used to zero all bits related to variable parts of that instruction, like source and destination registers. The MATCH is instead the definition of the fixed pattern, it can also be seen as the encoding an instruction would have once placed in logical AND with the corresponding MASK. These are used internally by GCC to generate the proper machine-dependent code.

The first step is to define those masks. As an example, taking the instruction MUL8ST, in the file `riscv-opc.h` the following definitions have to be added:
`#define MATCH_MUL8ST 0xd0001033`, as the MATCH value.
`#define MATCH_MUL8ST 0xfe00707f`, as the MASK value.
Two things to note, the MATCH value follows each instruction definition (that is presented in the next paragraph), instead, the MASK value is the same for all instructions since all added instruction are of R-type.

Pairing process between the instruction final name and the instruction encoding is done with the following C-macro:
`DECLARE_INSN(name, MATCH, MASK)`
That in the case of the MUL8ST instruction become:
`DECLARE_INSN(mul8st, MATCH_MUL8ST, MASK_MUL8ST)`
With *mul8st* being the custom assembly instruction added.

The second file to modify is located at `riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c`.
This file contains the encoding information of given instructions, organized as a C-struct, named *riscv_opcode* containing: the name of the instruction, the length, the instruction class, both the MATCH and MACK definitions previously mentioned, and some other information, among which, if the instruction could cause hazard events to ease the instruction scheduling. Instruction encodings are organized in a vector of *riscv_opcode* struct objects, one for each instruction.

To add a custom instruction we need to add the definition of a struct object for that specific instruction. For example, taking the MUL8ST custom instruction, the struct definition is as follows:
`{"mul8st", 0, INSN_CLASS_M, "d,s,t", MATCH_MUL8ST, MASK_MUL8ST, match_opcode, 0 }`
Specifically it declares a struct having as parameters the name being *mul8st*, the instruction class, a RISC-V M-extension instruction in this case, a string indicating it is an R-type having a destination register (d), and two source registers (s and t), the MATCH and MASK definitions as before, *match_opcode* that is a function used to match that instruction, and a value 0 indicating no hazard could be generated by this instruction, as expected.

At this point, the process is to carefully write down each instruction, present in the target ISA extension, in those files with the presented criteria, and to start the GCC compilation process.

As per the GCC compilation process, a script is provided ready-to-use on the same Git repository [53]. Just setting some parameters, it allows compiling the toolchain through makefiles, and easily test the final executable.

Parameters that are essential to be indicated to the GCC compiling script are, the architecture the processor is compliant with, that is, the ***march*** parameter, and the application binary interface to be used inside compiled procedures, called ***mabi*** parameter. As per the Ibex architecture, parameters to indicate are:
`march=rv32imc`, and `mabi=32ilp`, the first indicates the RISC-V base set and the extensions supported by the Ibex core, while the second indicates the bit-width of the architecture in terms of i (integer), l (long), and p (pointer), all of them on 32 bit registers.

4.3 Reference of the RISC-V ISA extension

In this section the derivation process for new RISC-V compliant instructions is presented. The standard RISC-V M-extension [54] includes only register-to-register

(R-type) instructions, meaning all information about operands must be loaded in registers before that instruction execution. Instruction format for R-type RISC-V instructions is the following:

31 25	24 20	19 15	14 12	11 7	6 0
funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode

The encoding of the four multiplication related instruction in the RISC-V M-extension is as follows:

	funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
MUL	0000001b	-	-	000b	-	0110011b

	funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
MULH	0000001b	-	-	001b	-	0110011b

	funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
MULHSU	0000001b	-	-	010b	-	0110011b

	funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
MULHU	0000001b	-	-	011b	-	0110011b

First problem is to find a set of "missing-codes" to be used as encoding for new instructions to be added. Here I chose not to deal with all possible instruction overlappings in the overall RISC-V ISA, but only to consider instructions that the Ibex processor is supporting, thus limited to the ones actually present in its decode unit. Given this assumption, there is no-guarantee the proposed encoding of these instructions is not overlapping with other instructions in other parts of the RISC-V ISA, nor in future ratified ISA extensions.

In order to keep compatibility with instructions in the RISC-V M-extension, and to reduce modifications to the Ibex decode unit, the *Opcode* that identify actual supported multiplication instructions is left unchanged. The *Function-code* instead is changed, among supported configurations, to identify uniquely these new instructions.

Modifications to the decode unit are just some extra signal assignment for control signals used by the multiplier units.

4.3.1 SS MUL instructions group

MUL16SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	000b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. For each operation it returns the lower 16 bit part of the result, in sequence, according to the following relation:

$$\begin{aligned} R_{OUT}[15:0] &= R_{IN1}[15:0] \times R_{IN2}[31:16], \\ R_{OUT}[31:16] &= R_{IN1}[31:16] \times R_{IN2}[15:0] \end{aligned} \tag{4.1}$$

MUL16SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	100b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. For each operation it returns the upper 16 bit part of the result, the relation is the same as the one in Equation 4.1.

MUL16SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1000000b	-	-	100b	-	0110011b

It computes two separate unsigned multiplications given that input registers contain the value of two input pairs on 16 bit each. For each operation it returns the upper 16 bit part of the result, the relation is the same as the one in Equation 4.1.

MUL8SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	001b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. For each operation it returns the lower 8 bit part of the result, in sequence, according to the following relation:

$$\begin{aligned} R_{OUT}[7:0] &= R_{IN1}[7:0] \times R_{IN2}[23:16], \\ R_{OUT}[15:8] &= R_{IN1}[15:8] \times R_{IN2}[31:24], \\ R_{OUT}[23:16] &= R_{IN1}[23:16] \times R_{IN2}[7:0], \\ R_{OUT}[31:24] &= R_{IN1}[31:24] \times R_{IN2}[15:8] \end{aligned} \tag{4.2}$$

MUL8SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	101b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. For each operation it returns the upper 8 bit part of the result, the relation is the same as the one in Equation 4.2.

MUL8SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1000000b	-	-	101b	-	0110011b

It computes four separate unsigned multiplications given that input registers contain the value of four input pairs on 8 bit each. For each operation it returns the upper 8 bit part of the result, the relation is the same as the one in Equation 4.2.

MUL4SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	010b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. For each operation it returns the lower 4 bit part of the result, in sequence, according to the following relation:

$$\begin{aligned}
R_{OUT}[3:0] &= R_{IN1}[3:0] \times R_{IN2}[19:16], \\
R_{OUT}[7:4] &= R_{IN1}[7:4] \times R_{IN2}[23:30], \\
R_{OUT}[11:8] &= R_{IN1}[11:8] \times R_{IN2}[27:24], \\
R_{OUT}[15:12] &= R_{IN1}[15:12] \times R_{IN2}[31:28], \\
R_{OUT}[19:16] &= R_{IN1}[19:16] \times R_{IN2}[3:0], \\
R_{OUT}[23:20] &= R_{IN1}[23:20] \times R_{IN2}[7:4], \\
R_{OUT}[27:24] &= R_{IN1}[27:24] \times R_{IN2}[11:8], \\
R_{OUT}[31:28] &= R_{IN1}[31:28] \times R_{IN2}[15:12]
\end{aligned} \tag{4.3}$$

MUL4SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1001000b	-	-	110b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. For each operation it returns the upper 4 bit part of the result, the relation is the same as the one in Equation 4.3.

MUL4SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1000000b	-	-	110b	-	0110011b

It computes eight separate unsigned multiplications given that input registers contain the value of eight input pairs on 4 bit each. For each operation it returns the upper 4 bit part of the result, the relation is the same as the one in Equation 4.3.

4.3.2 ST MUL instructions group

MUL16ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1101000b	-	-	000b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. Each single result is then summed before returning the computed value. The input/output relation follows:

$$R_{OUT}[31:0] = R_{IN1}[15:0] \times R_{IN2}[31:16] + R_{IN1}[31:16] \times R_{IN2}[15:0] \quad (4.4)$$

MUL16STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1100000b	-	-	000b	-	0110011b

It computes two separate unsigned multiplications given that input registers contain the value of two input pairs on 16 bit each. Each single result is then summed before returning the computed value. The input/output relation is the same as the one in Equation 4.4.

MUL8ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1101000b	-	-	001b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. Each single result is then summed before returning the computed value. The input/output relation follows:

$$\begin{aligned} R_{OUT}[23:0] = {} & R_{IN1}[7:0] \times R_{IN2}[31:24] + R_{IN1}[15:8] \times R_{IN2}[23:16] \\ & + R_{IN1}[23:16] \times R_{IN2}[15:8] + R_{IN1}[31:24] \times R_{IN2}[7:0] \end{aligned} \tag{4.5}$$

MUL8STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1100000b	-	-	001b	-	0110011b

It computes four separate unsigned multiplications given that input registers contain the value of four input pairs on 8 bit each. Each single result is then summed before returning the computed value. The input/output relation is the same as the one in Equation 4.5.

MUL4ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1101000b	-	-	010b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. Each single result is then summed before returning the computed value. The input/output relation follows:

$$\begin{aligned} R_{OUT}[19:0] = {} & R_{IN1}[3:0] \times R_{IN2}[31:28] + R_{IN1}[7:4] \times R_{IN2}[27:24] \\ & + R_{IN1}[11:8] \times R_{IN2}[23:20] + R_{IN1}[15:12] \times R_{IN2}[19:16] \\ & + R_{IN1}[19:16] \times R_{IN2}[15:12] + R_{IN1}[23:20] \times R_{IN2}[11:8] \\ & + R_{IN1}[27:24] \times R_{IN2}[7:4] + R_{IN1}[31:28] \times R_{IN2}[3:0] \end{aligned} \tag{4.6}$$

MUL4STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1100000b	-	-	010b	-	0110011b

It computes eight separate unsigned multiplications given that input registers contain the value of eight input pairs on 4 bit each. Each single result is then summed before returning the computed value. The input/output relation is the same as the one in Equation 4.6.

4.3.3 MAC instructions group

For consistency on results, it is advised to first clear the ACC register value, then start computing and accumulating results, paying always attention to use the same instruction along all computation procedure; if not, results could not be consistent, since results of different instructions are blindly summed together.

MAC

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
0111000b	-	-	000b	-	0110011b

It computes the MAC operation between two signed operands on 32 bit. Internally it accumulates results over time, then it returns the lower 32 bit part of the result, according to the following relation:

$$R_{OUT}[31:0] = R_{IN1}[31:0] \times R_{IN2}[31:0] + ACC[64:0] \tag{4.7}$$

MACH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
0111000b	-	-	001b	-	0110011b

It computes the MAC operation between two signed operands on 32 bit. Internally it accumulates results over time, then it returns the upper 32 bit part of the result, the relation is the same as the one in Equation 4.7

MACHSU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
0111000b	-	-	010b	-	0110011b

It computes the MAC operation between a signed and an unsigned operand on 32 bit. Internally it accumulates results over time, then it returns the upper 32 bit part of the result, the relation is the same as the one in Equation 4.7

MACHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
0111000b	-	-	0110b	-	0110011b

It computes the MAC operation between two unsigned operands on 32 bit. Internally it accumulates results over time, then it returns the upper 32 bit part of the result, the relation is the same as the one in Equation 4.7

4.3.4 SS MAC instructions group

For consistency on results, it is advised to first clear the ACC register value, then start computing and accumulating results, paying always attention to use the same instruction along all computation procedure; if not, results could not be consistent, since results of different instructions are blindly summed together.

MAC16SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	000b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. Internally it accumulates each result over time, then it returns the lower 16 bit part of the result, in sequence, according to the following relation:

$$\begin{aligned} R_{OUT}[15:0] &= R_{IN1}[15:0] \times R_{IN2}[31:16] + ACC[31:0], \\ R_{OUT}[31:16] &= R_{IN1}[31:16] \times R_{IN2}[15:0] + ACC[63:32] \end{aligned} \tag{4.8}$$

MAC16SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	100b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. Internally it accumulates each result over time, then it returns the upper 16 bit part of the result, the relation is the same as the one in Equation 4.8.

MAC16SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1010000b	-	-	100b	-	0110011b

It computes two separate unsigned multiplications given that input registers contain the value of two input pairs on 16 bit each. Internally it accumulates each result over time, then it returns the upper 16 bit part of the result, the relation is the same as the one in Equation 4.8.

MAC8SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	001b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. Internally it accumulates each result

over time, then it returns the lower 8 bit part of the result, in sequence, according to the following relation:

$$\begin{aligned} R_{OUT}[7:0] &= R_{IN1}[7:0] \times R_{IN2}[23:16] + ACC[15:0], \\ R_{OUT}[15:8] &= R_{IN1}[15:8] \times R_{IN2}[31:24] + ACC[31:16], \\ R_{OUT}[23:16] &= R_{IN1}[23:16] \times R_{IN2}[7:0] + ACC[47:32], \\ R_{OUT}[31:24] &= R_{IN1}[31:24] \times R_{IN2}[15:8] + ACC[63:48] \end{aligned} \tag{4.9}$$

MAC8SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	101b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. Internally it accumulates each result over time, then it returns the upper 8 bit part of the result, the relation is the same as the one in Equation 4.9.

MAC8SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1010000b	-	-	101b	-	0110011b

It computes four separate unsigned multiplications given that input registers contain the value of four input pairs on 8 bit each. Internally it accumulates each result over time, then it returns the upper 8 bit part of the result, the relation is the same as the one in Equation 4.9.

MAC4SS

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	010b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. Internally it accumulates each result over time, then it returns the lower 4 bit part of the result, in sequence, according to the following relation:

$$
\begin{aligned}
R_{OUT}[3:0] &= R_{IN1}[3:0] \times R_{IN2}[19:16] + ACC[7:0], \\
R_{OUT}[7:4] &= R_{IN1}[7:4] \times R_{IN2}[23:20] + ACC[15:8], \\
R_{OUT}[11:8] &= R_{IN1}[11:8] \times R_{IN2}[27:24] + ACC[23:16], \\
R_{OUT}[15:12] &= R_{IN1}[15:12] \times R_{IN2}[31:28] + ACC[31:24], \\
R_{OUT}[19:16] &= R_{IN1}[19:16] \times R_{IN2}[3:0] + ACC[39:32], \\
R_{OUT}[23:20] &= R_{IN1}[23:20] \times R_{IN2}[7:4] + ACC[47:40], \\
R_{OUT}[27:24] &= R_{IN1}[27:24] \times R_{IN2}[11:8] + ACC[55:48], \\
R_{OUT}[31:28] &= R_{IN1}[31:28] \times R_{IN2}[15:12] + ACC[63:56]
\end{aligned}
\tag{4.10}
$$

MAC4SSH

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1011000b	-	-	110b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. Internally it accumulates each result over time, then it returns the upper 4 bit part of the result, the relation is the same as the one in Equation 4.10.

MAC4SSHU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1010000b	-	-	110b	-	0110011b

It computes eight separate unsigned multiplications given that input registers contain the value of eight input pairs on 4 bit each. Internally it accumulates each result over time, then it returns the upper 4 bit part of the result, the relation is the same as the one in Equation 4.10.

4.3.5 ST MAC instructions group

For consistency on results, it is advised to first clear the ACC register value, then start computing and accumulating results, paying always attention to use the same instruction along all computation procedure; if not, results could not be consistent, since results of different instructions are blindly summed together.

MAC16ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1111000b	-	-	000b	-	0110011b

It computes two separate signed multiplications given that input registers contain the value of two input pairs on 16 bit each. Each single result is then summed,

along with the internal accumulation register value, before returning the final result. The input/output relation follows:

$$\begin{aligned} R_{OUT}[31:0] = {} & R_{IN1}[15:0] \times R_{IN2}[31:16] \\ & + R_{IN1}[31:16] \times R_{IN2}[15:0] + ACC[47:16] \end{aligned} \tag{4.11}$$

MAC16STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1110000b	-	-	000b	-	0110011b

It computes two separate unsigned multiplications given that input registers contain the value of two input pairs on 16 bit each. Each single result is then summed, along with the internal accumulation register value, before returning the final result. The input/output relation is the same as the one in Equation 4.11.

MAC8ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1111000b	-	-	001b	-	0110011b

It computes four separate signed multiplications given that input registers contain the value of four input pairs on 8 bit each. Each single result is then summed, along with the internal accumulation register value, before returning the final result. The input/output relation follows:

$$\begin{aligned} R_{OUT}[23:0] = {} & R_{IN1}[7:0] \times R_{IN2}[31:24] + R_{IN1}[15:8] \times R_{IN2}[23:16] \\ & + R_{IN1}[23:16] \times R_{IN2}[15:8] + R_{IN1}[31:24] \times R_{IN2}[7:0] \\ & + ACC[47:24] \end{aligned} \tag{4.12}$$

MAC8STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1110000b	-	-	001b	-	0110011b

It computes four separate unsigned multiplications given that input registers contain the value of four input pairs on 8 bit each. Each single result is then summed, along with the internal accumulation register value, before returning the final result. The input/output relation is the same as the one in Equation 4.12.

MAC4ST

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1111000b	-	-	010b	-	0110011b

It computes eight separate signed multiplications given that input registers contain the value of eight input pairs on 4 bit each. Each single result is then summed, along with the internal accumulation register value, before returning the final result. The input/output relation follows:

$$\begin{aligned} R_{OUT}[19:0] = {} & R_{IN1}[3:0] \times R_{IN2}[31:28] + R_{IN1}[7:4] \times R_{IN2}[27:24] \\ & + R_{IN1}[11:8] \times R_{IN2}[23:20] + R_{IN1}[15:12] \times R_{IN2}[19:16] \\ & + R_{IN1}[19:16] \times R_{IN2}[15:12] + R_{IN1}[23:20] \times R_{IN2}[11:8] \\ & + R_{IN1}[27:24] \times R_{IN2}[7:4] + R_{IN1}[31:28] \times R_{IN2}[3:0] \\ & + ACC[47:28] \end{aligned} \tag{4.13}$$

MAC4STU

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1110000b	-	-	010b	-	0110011b

It computes eight separate unsigned multiplications given that input registers contain the value of eight input pairs on 4 bit each. Each single result is then summed, along with the internal accumulation register value, before returning the final result. The input/output relation is the same as the one in Equation 4.13.

4.3.6 MAC special instructions

MACSET

funct[9:3]	rs2	rs1	funct[2:0]	rd	opcode
1111000b	-	-	111b	-	0110011b

This is a special instruction to set an arbitrary value in the internal accumulation register used by MAC instructions. The MAC register is on 64 bit, its value is set from a binary concatenation on the two input registers content each on 32 bit:

$$MAC[63:0] = R_{IN2}[31:0] \ \& \ R_{IN1}[31:0] \tag{4.14}$$

Chapter 5

QNN benchmarks and execution time results

This chapter exposes benchmarks developed to verify final SoC performance of structures derived in Chapter 3, with respect to the two baseline Ibex structures, and presents the performance results. First section introduces the three ML benchmarks defined, one for each of the three most common QNN layers in the edge domain. Second section exposes the test methodology to execute these benchmarks on the Ibex core. Execution involves the usage of an FPGA board programmed with the ESP [56] toolchain allowing integrating, verifying and executing software routines on supported cores. In the last section an analysis of results make evident advantages of the sub-word parallel approach when dealing with the QNNs execution, and provides some metrics to quantify these improvements.

5.1 Target QNN algorithm description

Each of the following subsections targets a single QNN algorithm. All of them are subsequently run on the Ibex core to evaluate performance improvements of custom multipliers with respect to the baseline. In the first subsection, regarding the FC layer, five different algorithms are exposed, two of them include only instructions present in the original Ibex core (mainly 32 bit MUL and ADD), other two make use of the 32 bit MAC operations to isolate the contribution of MAC operations with respect to the precision-scalable approach, while the last algorithm is the most optimized one in executing an FC layer, making use of only precision-scalable ST custom instructions. As per the other subsections, targeting CNN and DW-CNN layers, I decided to focus only on the advantages coming from the usage of precision-scalable operations, with the only baseline considered being the versions using the 32 bit MAC operations.

Listing 5.1: FC algorithm using MUL and ADD instructions.

```
for (int out_pos = 0; out_pos < output_lenght; ++out_pos) {
  // reset accumulation variable
  int32_t acc = 0;

  for (int in_pos = 0; in_pos < input_lenght; ++in_pos) {
    // load the input value
    int input_idx = in_pos;
    int8_t input_val = input_data[input_idx];
    // load the filter value
    int filter_idx = out_pos*input_lenght + in_pos;
    int8_t filter_val = filter_data[filter_idx];

    int32_t mul_res;
    // multiply the input and filter values
    asm volatile("mul %0, %1, %2\n":"=r"(mul_res):"r"(input_val),"r"(filter_val):);
    // accumulate in the accumulation variable
    asm volatile("add %0, %1, %2\n":"=r"(acc):"r"(acc),"r"(mul_res):);
  }

  // store the result
  int output_idx = out_x_pos;
  output_data[output_idx] = acc;
}
```

Listing 5.2: FC algorithm using MAC instructions.

```
for (int out_pos = 0; out_pos < output_lenght; ++out_pos) {
  int acc;

  // reset the internal accumulation register
  asm volatile("macset x0, x0, x0");

  for (int in_pos = 0; in_pos < input_lenght; ++in_pos) {
    // load the input value
    int input_idx = in_pos;
    int8_t input_val = input_data[input_idx];
    // load the filter value
    int filter_idx = out_pos*input_lenght + in_pos;
    int8_t filter_val = filter_data[filter_idx];

    // perform the MAC operation saving result on the accumulation variable
    asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(input_val),"r"(filter_val):);
  }

  // store the result
  int output_idx = out_x_pos;
  output_data[output_idx] = acc;
}
```

5.1.1 Fully-connected layer benchmark

Fully-connected layer, also called dense layer, feedforward NN, or, in honor of its origin, single-layer Perceptron, is one of the oldest algorithms in the ML domain. Conceived in the 40s, this approach was not initially considered having good performance in classification tasks. In subsequent years, the idea of building NNs staking layers one after the other, what is called a multi-layer Perceptron, revealed its real potential. This NN layer has a given number of inputs and a given number of outputs. From the name fully-connected, it follows each input has a connection with each output, each connection corresponds to a parameter that, once multiplied with the corresponding input value, it is accumulated contributing to the result of that specific output. This leads to a highly parametrized structure. Indeed, since this layer has a higher number of parameters, it also has higher computational and memory requirements than other approaches like the convolutional one.

Listing 5.3: FC algorithm using MUL and ADD instructions with packed memory access.

```
for (int out_pos = 0; out_pos < output_lenght; ++out_pos){
  // reset accumulation variable
  int32_t acc = 0;

  for (int in_pos = 0; in_pos < input_lenght/4; ++in_pos) {
    // load the input packet
    int input_idx = in_pos;
    int32_t input_val = input_data[input_idx];
    // load the filter packet
    int filter_idx = out_pos*(input_lenght/4) + in_pos;
    int32_t filter_val = filter_data[filter_idx];

    int32_t mul_res;

    // shift and mask
    int32_t in_val_1 = (int16_t)(input_val & 0x000000FF);
    int32_t fil_val_1 = (int16_t)(filter_val & 0x000000FF);
    asm volatile("mul %0, %1, %2\n":"=r"(mul_res):"r"(in_val_1),"r"(fil_val_1):);
    asm volatile("add %0, %1, %2\n":"=r"(acc):"r"(acc),"r"(mul_res):);

    int32_t in_val_2 = (int16_t)((input_val >> 8) & 0x000000FF);
    int32_t fil_val_2 = (int16_t)((filter_val >> 8) & 0x000000FF);
    asm volatile("mul %0, %1, %2\n":"=r"(mul_res):"r"(in_val_2),"r"(fil_val_2):);
    asm volatile("add %0, %1, %2\n":"=r"(acc):"r"(acc),"r"(mul_res):);

    int32_t in_val_3 = (int16_t)((input_val >> 16) & 0x000000FF);
    int32_t fil_val_3 = (int16_t)((filter_val >> 16) & 0x000000FF);
    asm volatile("mul %0, %1, %2\n":"=r"(mul_res):"r"(in_val_3),"r"(fil_val_3):);
    asm volatile("add %0, %1, %2\n":"=r"(acc):"r"(acc),"r"(mul_res):);

    int32_t in_val_4 = (int16_t)((input_val >> 24) & 0x000000FF);
    int32_t fil_val_4 = (int16_t)((filter_val >> 24) & 0x000000FF);
    asm volatile("mul %0, %1, %2\n":"=r"(mul_res):"r"(in_val_4),"r"(fil_val_4):);
    asm volatile("add %0, %1, %2\n":"=r"(acc):"r"(acc),"r"(mul_res):);
  }

  // store the result
  int output_idx = out_pos;
  output_data[output_idx] = acc;
}
```

The algorithm of a fully-connected layer computes a number of MAC operations for each output equal to the number of inputs, where every input is multiplied with a dedicated parameter. A C-like pseudo-code of this algorithm for the 8 bit case is presented in Listing 5.1 and Listing 5.2. From these pieces of code, we can see the number of MAC operations (either implemented by the concatenation of a MUL and an ADD instructions, or carried out by a single MAC instruction) is equal $I_{size} \cdot O_{size}$, where I_{size} is the number of inputs while O_{size} is the number of outputs. The number of parameters follows the relation given by the same formula. All of this means every improvement in the MAC execution flow leads to improvements on execution time, thus directly on NN inference latency.

The standard algorithms in Listing 5.1 and 5.2, perform a memory access each time a MAC operation need to be done. Since optimized instructions deal directly with the parameter packed in groups equal to the word length of the Ibex core, the influence of performing fewer load operations from memory, and making fewer loop cycles could lead to performance improvements not directly related to the multiplier unit. For this reason variations of these standard algorithms have been developed.

Listing 5.4: FC algorithm using MAC instructions with packed memory access.

```
for (int out_pos = 0; out_pos < output_lenght; ++out_pos){
  int32_t acc;

  // reset the internal accumulation register
  asm volatile("macset x0, x0, x0");

  for (int in_pos = 0; in_pos < input_lenght/4; ++in_pos) {
    // load the input packet
    int input_idx = in_pos;
    int32_t input_val = input_data[input_idx];
    // load the filter packet
    int filter_idx = out_pos*(input_lenght/4) + in_pos;
    int32_t filter_val = filter_data[filter_idx];

    // shift and mask
    int32_t in_val_1 = (int16_t)(input_val & 0x000000FF);
    int32_t fil_val_1 = (int16_t)(filter_val & 0x000000FF);
    asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_1),"r"(fil_val_1):);

    int32_t in_val_2 = (int16_t)((input_val >> 8) & 0x000000FF);
    int32_t fil_val_2 = (int16_t)((filter_val >> 8) & 0x000000FF);
    asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_2),"r"(fil_val_2):);

    int32_t in_val_3 = (int16_t)((input_val >> 16) & 0x000000FF);
    int32_t fil_val_3 = (int16_t)((filter_val >> 16) & 0x000000FF);
    asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_3),"r"(fil_val_3):);

    int32_t in_val_4 = (int16_t)((input_val >> 24) & 0x000000FF);
    int32_t fil_val_4 = (int16_t)((filter_val >> 24) & 0x000000FF);
    asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_4),"r"(fil_val_4):);
  }

  // store the result
  int output_idx = out_pos;
  output_data[output_idx] = acc;
}
```

Listing 5.5: FC algorithm using custom MACST instructions.

```
for (int out_pos = 0; out_pos < output_lenght; ++out_pos) {
  int acc;

  // reset the internal accumulation register
  asm volatile("macset x0, x0, x0");

  for (int in_pos = 0; in_pos < input_lenght/4; ++in_pos) {
    // load the input packet
    const int input_idx = in_pos;
    const int32_t input_val = input_data[input_idx];
    // load the filter packet
    const int filter_idx = out_pos*(input_lenght/4) + in_pos;
    const int32_t filter_val = filter_data[filter_idx];

    // perform the MAC operation of the four inputs in parallel
    asm volatile("mac8st %0, %1, %2\n":"=r"(acc):"r"(input_val),"r"(filter_val):);
  }

  // store the result
  const int output_idx = out_pos;
  output_data[output_idx] = acc;
}
```

They access a packet of data, and then with some shift and mask operations, that are likely to cost less than a memory access or loop cycling, it restores the original data content. However, in order not to add an if-check condition at each operation interrupting the computational flow to check if all data in a packet are valid or not, once a packet is loaded all possible operations are executed in every case. The

algorithm definition for the 8 bit case is presented in Listing 5.3, using MUL and ADD operations, and in Listing 5.4, for the one using MAC operations. Proposed versions are unrolled and, of course, they have the side effect of requiring a higher code size.

The third variant is the one making use of the sub-word parallel MAC instructions, thus operating less machine cycles to compute the result. This variant is visible in Listing 5.5.

Fully connected layers are usually placed as the very last layers of a network, thus the input and output range is limited. For this reason the considered range for testing include a number of inputs from 128 to 256, and output number limited to 8, 16 and 32, even if, it is clear, the theoretical execution time increases linearly with the number of outputs.

5.1.2 Convolutional layer benchmark

CNN layers [27] are of different types depending on input and filter dimensionality. Hereinafter I refer to the 2-Dimensional variant.
Convolutional layer are inspired both from the biological visual cortex, but also on typical convolution of filters used in DSP applications for image processing. Usually, the 2D convolution layer takes a 3-Dimensional filter and convolves it over a 3D input along the first two dimensions, which are called *height* and *width*. This filter has usually the same length for the third dimension, called *depth* or more often *channels*. For each output position, all overlapping values between the input and the filter are multiplied and accumulated before moving to the next filter position. Once the filter have assumed all positions, the accumulated value is written to output. Filter values are the parameter of the NN layer, different filters produce different outputs, which are stacked along the third-dimension forming the channels of the output, the output then becomes the input of the following layer and so on. Figure 5.1 represents visually the convolution process.

Given a fixed number of inputs, since the filter dimension is smaller than the input dimension, this NN layer have a reduced number of parameters, thus lower memory requirements compared to the fully connected one. Moreover, for the same reason, given also a fixed number of outputs, this NN layer have reduced computational requirements. The memory requirement is limited by the filter dimension, while the computational one is limited also by the output dimension. Output dimension is determined by some parameters like the filter dimension, the stride of the convolution process, the presence of a padding in the input, a dilation factor for the filter and others. Since these considerations are out of the scope of the benchmark developed, I decided to cut this explanation.

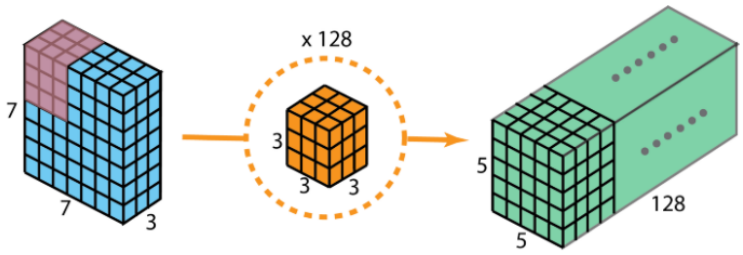


Figure 5.1: The convolution process: each of the 128 filters convolves the input producing one of the 128 output channels.

Listing 5.6: CNN algorithm using MAC instructions.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < output_depth; ++out_ch_pos) {
      int32_t acc;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          for (int fil_ch_pos = 0; fil_ch_pos < input_depth; ++fil_ch_pos) {
            // load the input value
            int input_idx = in_y_pos*(input_depth*input_width)
                                          + in_x_pos*(input_depth) + fil_ch_pos;
            int8_t input_val = input_data[input_idx];
            // load the filter value
            int filter_idx = out_ch_pos*(input_depth*filter_width*filter_height)
                                         + fil_y_pos*(input_depth*filter_width)
                                         + fil_x_pos*(input_depth) + fil_ch_pos;
            int8_t filter_val = filter_data[filter_idx];

            // perform the MAC operation saving result on the accumulation variable
          asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(input_val),"r"(filter_val):);
          }
        }
      }

      // store the result
      int output_idx = out_y_pos*(output_depth*output_width)
                                        + out_x_pos*(output_depth) + out_ch_pos;
      output_data[output_idx] = acc;
    }
  }
}
```

From the previous description, the algorithm used for this CNN benchmark, for the 8 bit quantization case, is the one in Listing 5.6, which, as anticipated, directly uses only 32 bit MAC instructions.

Given the implementation in Listing 5.6, we can derive the number of parameter is equal to $F_{size} \cdot N_{out\_ch}$, where F_{size} is the filter size while N_{out_ch} is the number of output channel. The total number of MAC operations instead is equal to $F_{size} \cdot O_{size}$, again F_{size} is the filter size, while O_{size} is the total number of outputs.

Listing 5.7: CNN algorithm using MAC instructions with packed memory access.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < output_depth; ++out_ch_pos) {
      int32_t acc;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          for (int fil_ch_pos = 0; fil_ch_pos < input_depth/4; ++fil_ch_pos) {
            // load the input packet
            int input_idx = in_y_pos*((input_depth/4)*input_width)
                                    + in_x_pos*((input_depth/4)) + fil_ch_pos;
            int32_t input_val = input_data[input_idx];
            // load the filter packet
            int filter_idx = out_ch_pos*((input_depth/4)*filter_width*filter_height)
                                      + fil_y_pos*((input_depth/4)*filter_width)
                                      + fil_x_pos*((input_depth/4)) + fil_ch_pos;
            int32_t filter_val = filter_data[filter_idx];

            // shift and mask
            int32_t in_val_1 = (int8_t)(input_val & 0x000000FF);
            int32_t fil_val_1 = (int8_t)(filter_val & 0x000000FF);
            asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_1),"r"(fil_val_1):);

            int32_t in_val_2 = (int8_t)((input_val >> 8) & 0x000000FF);
            int32_t fil_val_2 = (int8_t)((filter_val >> 8) & 0x000000FF);
            asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_2),"r"(fil_val_2):);

            int32_t in_val_3 = (int8_t)((input_val >> 16) & 0x000000FF);
            int32_t fil_val_3 = (int8_t)((filter_val >> 16) & 0x000000FF);
            asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_3),"r"(fil_val_3):);

            int32_t in_val_4 = (int8_t)((input_val >> 24) & 0x000000FF);
            int32_t fil_val_4 = (int8_t)((filter_val >> 24) & 0x000000FF);
            asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(in_val_4),"r"(fil_val_4):);
          }
        }
      }

      // store the result
      int output_idx = out_y_pos*(output_depth*output_width)
                                      + out_x_pos*(output_depth) + out_ch_pos;
      output_data[output_idx] = acc;
    }
  }
}
```

As previously done, a variation to see if the number of memory accesses makes the difference is also considered in Listing 5.7.

The last version, the one in Listing 5.8, exploits the parallelization capabilities offered by custom reduced-precision instructions, specifically MAC8ST for the 8 bit benchmark.

To conclude, given the wide range of use cases for the CNN layer, in order not to have long execution time testing all possible configurations, I decided to define two major use case and set the layer hyperparameters accordingly. A CNN layer, either it happens to be used in the first layers of a NN, meaning both input (thus also

Listing 5.8: CNN algorithm using custom MACST instructions.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < output_depth; ++out_ch_pos) {
      int acc;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          for (int fil_ch_pos = 0; fil_ch_pos < input_depth/4; ++fil_ch_pos) {
            // load the input packet
            int input_idx = in_y_pos*((input_depth/4)*input_width)
                                    + in_x_pos*((input_depth/4)) + fil_ch_pos;
            int32_t input_val = input_data[input_idx];
            // load the filter packet
            int filter_idx = out_ch_pos*((input_depth/4)*filter_width*filter_height)
                                      + fil_y_pos*((input_depth/4)*filter_width)
                                      + fil_x_pos*((input_depth/4)) + fil_ch_pos;
            int32_t filter_val = filter_data[filter_idx];

            // perform the MAC operation of the four inputs in parallel
            asm volatile("mac8st %0, %1, %2\n":"=r"(acc):"r"(input_val), "r"(filter_val):);
          }
        }
      }

      // store the result
      int output_idx = out_y_pos*(output_depth*output_width)
                                    + out_x_pos*(output_depth) + out_ch_pos;
      output_data[output_idx] = acc;
    }
  }
}
```

filters) and output have a higher height and width, paired with a reduced number of channels, and the filter 2D-window could be bigger; or it is used as an almost final layer of the NN having smaller input and output height and width, but a higher number of channels, and the filter dimension could be smaller. In the first case, considering edge applications, input height and width could be from 16x16 [1] to even 512x512 (depending on the application, here I stop to 16x16) and a number of channels varying from 1 to at most 16; the filter size is usually of 3x3, or more rarely of 5x5. In the second case the considered input shape is in a range from 4x4 to 8x8, with a number of channels from 32 to 128, and a filter shape that is usually 3x3, but not rarely, 1x1 in the so called *313 CNN layer* [2] or as a second step of the *separable convolutional NN*, which is called *point-wise convolution*.

[1] Hereinafter the notation used is of the form *heightXwidht*, meaning that the notation 3x3 refers to an input having height = 3 and width = 3.

[2] This configuration is organized as a convolutional layer with 3x3 filters, followed by a convolutional layer with 1x1 filters, and again followed by another convolutional layer with 3x3 filters.

To summarize, the range considered, for these two test scenarios, is the following:

- **First layer case:**
  - Input shape = {16}.
  - Input channels = [1, 16].
  - Filter shape = {3, 5}.
- **Last layer case:**
  - Input shape = {4, 8}.
  - Input channels = [32, 128].
  - Filter shape = {1, 3}.

5.1.3 Depth-wise convolutional layer benchmark

Depth-wise convolution has been first introduced in the definition of *MobileNets* [9]. The objective in mind was to optimize both computation and memory requirements of CNN deployed on edge devices, that are often *mobile* devices. CNN computation and memory requirements directly depends on the filter size. The filter size could be reduced up to a 1x1 filter, but in this case each output would be dependent only on channels of the corresponding input. This way of using 1x1 filters is also called *point-wise convolution*. A way to counteract this phenomenon is to add a step before the point-wise convolution treating each input channel independently with a dedicated 3x3, or rarely 5x5, filter having one single channel. Each input channel convoluted with its filter produces an output channel, meaning that, the number of output channels is the same as the number of inputs. This approach of convolution is called *depth-wise convolution*. In Figure 5.2 the overall process is represented.

Figure 5.2: The depth-wise convolution process: each filter channel independently convolves with the corresponding input channel producing an output channel.

Listing 5.9: Depth-wise CNN standard algorithm.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < input_depth; ++out_ch_pos) {
      int32_t acc;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          // load the input value
          int input_idx = out_ch_pos*(input_depth*input_width)
                                        + in_y_pos*(input_width) + in_x_pos + i;
          int8_t input_val = input_data[input_idx];
          // load the filter value
          int filter_idx = out_ch_pos*(input_depth*filter_width)
                                    + fil_y_pos*(filter_width) + fil_x_pos + i;
          int8_t filter_val = filter_data[filter_idx];

          // perform the MAC operation saving result on the accumulation variable
          asm volatile("mac %0, %1, %2\n":"=r"(acc):"r"(input_val),"r"(filter_val):);
        }
      }

      // store the results
      int output_idx = out_y_pos*(input_depth*output_width)
                          + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack;
      output_data[output_idx] = acc;

    }
  }
}
```

Connecting a depth-wise convolutional NN layer, and a point-wise convolutional NN layer, forms the so called *separable depth-wise convolution*, that is, the main layer in *MobileNets* [9].
As in a normal convolution, also in this case the output is influenced by many factors like stride, padding, dilation, etc., but these have not been considered in the development process of the overall benchmark, and are linear in the computational requirements in almost every case.

Memory access management in this layer is a little more complicated. As also analyzed in [20], given that each input channel is independently contributing to distinct output channels, where each one has its own filter, and given the filter dimension being limited in size, usually 3x3 filters are used, with a total number of parameters equal to 9, thus leading to just 9 operations for each input-filter couple; for these reasons it is difficult to parallelize operations exploiting the ST approach. A way to limit this problem could be to multiply and accumulate separated filter computations in parallel, meaning that the parallelism does not occur at filter level, but rather it occurs at channel level, where more than one independent filter computation could be carried out at the same time.

Listing 5.10: Depth-wise CNN algorithm with packed memory access.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < input_depth/4; ++out_ch_pos) {
      int32_t acc_1, acc_2, acc_3, acc_4;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          // load the input packet
          int input_idx = out_ch_pos*((input_depth/4)*input_width)
                                              + in_y_pos*(input_width) + in_x_pos;
          int32_t input_val = input_data[input_idx];
          // load the filter packet
          int filter_idx = out_ch_pos*((input_depth/4)*input_width)
                                            + fil_y_pos*(input_width) + fil_x_pos;
          int32_t filter_val = filter_data[filter_idx];

          // shift and mask
          int32_t in_val_1 = (int8_t)(input_val & 0x000000FF);
          int32_t fil_val_1 = (int8_t)(filter_val & 0x000000FF);
          asm volatile("mac %0, %1, %2\n":"=r"(acc_1):"r"(in_val_1),"r"(fil_val_1):);

          int32_t in_val_2 = (int8_t)((input_val >> 8) & 0x000000FF);
          int32_t fil_val_2 = (int8_t)((filter_val >> 8) & 0x000000FF);
          asm volatile("mac %0, %1, %2\n":"=r"(acc_2):"r"(in_val_2),"r"(fil_val_2):);

          int32_t in_val_3 = (int8_t)((input_val >> 16) & 0x000000FF);
          int32_t fil_val_3 = (int8_t)((filter_val >> 16) & 0x000000FF);
          asm volatile("mac %0, %1, %2\n":"=r"(acc_3):"r"(in_val_3),"r"(fil_val_3):);

          int32_t in_val_4 = (int8_t)((input_val >> 24) & 0x000000FF);
          int32_t fil_val_4 = (int8_t)((filter_val >> 24) & 0x000000FF);
          asm volatile("mac %0, %1, %2\n":"=r"(acc_4):"r"(in_val_4),"r"(fil_val_4):);
        }
      }
      // store the results
      int output_idx_1 = out_y_pos*(input_depth*output_width)
                           + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack;
      output_data[output_idx_1] = acc_1;
      int output_idx_2 = out_y_pos*(input_depth*output_width)
                         + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 1;
      output_data[output_idx_2] = acc_2;
      int output_idx_3 = out_y_pos*(input_depth*output_width)
                         + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 2;
      output_data[output_idx_3] = acc_3;
      int output_idx_4 = out_y_pos*(input_depth*output_width)
                         + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 3;
      output_data[output_idx_4] = acc_4;
    }
  }
}
```

Of course, this means, in compiled code, we have to keep more than one partial result on which the accumulation take place in time, if the data organization have to be fixed as for all algorithms in this benchmark. An optimized architecture, having SS instructions support, like the one developed for this thesis, could exploit these instructions to accumulate separated values while multiplying them.

Given last considerations, the standard algorithm for the depth-wise convolution employed is the one in Listing 5.9, also in this case I consider only the one using 32 bit MAC instructions directly.

Listing 5.11: Depth-wise CNN algorithm using custom MACSS instructions.

```
for (int out_y_pos = 0; out_y_pos < output_height; ++out_y_pos) {
  for (int out_x_pos = 0; out_x_pos < output_width; ++out_x_pos) {
    for (int out_ch_pos = 0; out_ch_pos < input_depth/4; ++out_ch_pos) {
      int32_t acc_L, acc_H;

      // reset the internal accumulation register
      asm volatile("macset x0, x0, x0");

      for (int fil_y_pos = 0; fil_y_pos < filter_height; ++fil_y_pos) {
        int in_y_pos = out_y_pos + fil_y_pos;

        for (int fil_x_pos = 0; fil_x_pos < filter_width; ++fil_x_pos) {
          int in_x_pos = out_x_pos + fil_x_pos;

          // load the input packet
          int input_idx = out_ch_pos*((input_depth/4)*input_width)
                                          + in_y_pos*(input_width) + in_x_pos;
          int32_t input_val = input_data[input_idx];
          // load the filter packet
          int filter_idx = out_ch_pos*((input_depth/4)*input_width)
                                        + fil_y_pos*(input_width) + fil_x_pos;
          int32_t filter_val = filter_data[filter_idx];

          // perform the MAC operation - get the most-significant part of results
          asm volatile("mac8ssh %0, %1, %2\n":"=r"(acc_H):"r"(input_val), "r"(filter_val):);
        }
      }

      // performa a fake MAC operation - get the least-significant part of results
      asm volatile("mac8ss %0, x0, x0\n":"=r"(acc_L):);

      // store the results - with shift and mask overhead to restore results
      int output_idx_1 = out_y_pos*(input_depth*output_width)
                          + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack;
      output_data[output_idx_1] = (int16_t) ((acc_H << 8) & 0x0000FF00)
                                                    | (acc_L & 0x000000FF);
      int output_idx_2 = out_y_pos*(input_depth*output_width)
                        + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 1;
      output_data[output_idx_2] = (int16_t) (acc_H & 0x0000FF00)
                                                  | ((acc_L >> 8) & 0x000000FF);
      int output_idx_3 = out_y_pos*(input_depth*output_width)
                        + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 2;
      output_data[output_idx_3] = (int16_t) ((acc_H >> 8) & 0x0000FF00)
                                                  | ((acc_L >> 16) & 0x000000FF);
      int output_idx_4 = out_y_pos*(input_depth*output_width)
                        + out_x_pos*(input_depth) + out_ch_pos*n_data_in_a_pack + 3;
      output_data[output_idx_4] = (int16_t) ((acc_H >> 16) & 0x0000FF00)
                                                  | ((acc_L >> 24) & 0x000000FF);
    }
  }
}
```

Analyzing this code, the number of parameter is equal to $F_{size} \cdot N_{in\_ch}$, where F_{size} is the size of the 2-dimensional filter, and N_{in_ch} is the input number of channels. The number of operations is equal to $F_{size} \cdot O_{size}$, where F_{size} is again the filter size, while O_{size} is the output size.

A version loading an entire data packet, having thus the same memory accesses then the version using custom MACSS instructions is presented in Listing 5.10.

In Listing 5.11 there is the last version, the one making use of the custom MACSS instructions.

In the version using custom instructions we can notice, given the structure of data packet returned by the MACxxSS instructions, still it is necessary to perform

masking and shifting before writing a result, meaning that some advantages of using a single instruction to parallelize the execution is lost.

Given the constraint output channels must be equal to input channels, this varies from 16 to 64 channels; The considered input shape is also small, and it is 16x16 or 8x8, this choice is to speed up computation time, but also because computational time is increasing linearly between different algorithm parameters.

5.2 Test platform and methodology

The test platform, on which benchmarks are executed, makes use of the Embedded Scalable Platform (ESP) [29] [56]. ESP is a platform developed by the System-Level Design Group at Columbia University. It allows to easily integrate different Intellectual Properties (IP), amid CPU and accelerators, with the objective to make the development of SoC straightforward. ESP have been conceived for heterogeneous architectures, thus, where computing units of different type interact together to carry out computation for a target task. In this view, the ESP organization makes use of *Tiles*. A tile is a computing unit connected to a Network-on-Chip (Noc) used to exchange information between the main processor and different accelerators, or between different accelerators itself, which are then able to read/write shared memory positions (properly synchronized). Also, the memory management unit is interfaced as a tile in the SoC. Among examples making use of the ESP as a development basis, some are [30] [31]. Specifically, the processor tile is configurable with different RISC-V processors, among which also the Ibex core is included. After the configuration phase the ESP platform provides scripts to simulate, with cycle accurate precision, the overall SoC using HDL simulators, like Mentor Questa Sim, or to target FPGA-synthesis through Xilinx Vivado. This makes ESP suitable to test in a fast way the previously presented Ibex modifications, to both validate functionality and performance.

Since the objective of this thesis is restricted to the processor core development, I decided to skip the description and usage of ESP targeting the accelerator integration. As an informative compendium, ESP can be used to design accelerators at RTL [57], or exploiting HLS with many HLS tools, as for example [58], integrating third-party IP [59], or even using accelerators automatically designed for a given NN [60] where the design steps include the usage of the HLS4ML compiler [44].

To test each modified version of the Ibex core, it is sufficient to follow the setup documentation [61] and the one related on how to configure ESP to integrate a single-core in the final SoC [62].

The ESP configuration is done as follows. We need first to move into the FPGA

directory `esp_root/socs/<fpga_name>/`, then issue the `make esp-xconfig` command. Now a page with the tile organization of the SoC is shown, in this view there are four tiles by default (more can be added). Here we need to set a tile for external memory management, a tile for the I/O interface and a tile for the CPU. This configuration is already the default one. As per the CPU, using the parameters in the corresponding section, there is the possibility to select the Ibex core, which is instantiated starting from files in position `esp_root/rtl/cores/ibex/`. Finally, carefully unselect the cache being included in the SoC design, since, to evaluate performance increments coming only from architectural benefits we need to ensure deterministic behavior.

Once the SoC is set, to start a simulation, we need to issue the `make qsim` command, which sets and opens Questa Sim HDL simulator. From Questa Sim it is possible to analyze the cycle level accurate behavior of the Ibex core executing a specific routine. By default, this is located at `esp_root/socs/<fpga_name>/systest.c`, it is a simple C program that is compiled before setting Questa Sim. To compile it with a compiler including custom instruction in assembly format, we need to change the global variable pointing to the RISC-V GCC compiler, and make it pointing to our modified GCC including modifications presented in Chapter 4. The compilation process of the file `systest.c` can be started also by typing `make soft`.

The used FPGA for testing and performance analysis purpose is the Xilinx Virtex proFPGA XC7V2000T [45]. In order to compile the bitstream, starting from the FPGA directory, that in this case is `esp_root/socs/profpga-xc7v2000t/`, we can issue the command `make vivado-syn`. Once the Xilinx Vivado synthesis has completed, with the command `make fpga-program` the bitstream can be loaded into the FPGA. Then, typing `make fpga-run`, the same file `systest.c` is compiled with our custom compiler, and it is then executed by the core running on the FPGA. ESP provides already in the compilation flow the SW interface to make classical `printf()` printing on the serial monitor, which can be captured by a program like `minicom`.

For the sake of simplicity, I built a single parametric file, replacing the content of `systest.c`, that once compiled calls a specific class of benchmarks and reports on the output console the execution time in terms of machine cycles.

To measure performance in executing the computational loops of the three benchmarks previously presented, an internal counter can be used. In the RISC-V documentation the *Control and Status register file* organization is presented, amid these registers, one called `mcycle` fulfill the machine cycle counter functionality. It is located at address `0xB00`. The same counter is present in the Ibex documentation, thus it is available to be used. This register can both be red and written, and the counting process is controlled by an enable-bit, active-low, in the `mcountinhibit`

register at address `0x320`. Unfortunately, due to an incompatibility between the GCC version compiled and the Ibex core, the assembler cannot associate correctly the name to the corresponding register, for this reason the following parts make use of the register address directly instead.

Listing 5.12: Routine to read the mcycle register.

```
static inline uint32_t mcycle_read()
{
    uint32_t counter;

    // mcycle address is 0xb00
    asm volatile ("csrr t0, 0xb00\n");
    asm volatile ("mv %0, t0\n":"=r"(counter):);

    return counter;
}
```

Listing 5.13: Routine to reset the mcycle register.

```
static inline void mcycle_reset()
{
    // mcycle address is 0xb00
    asm volatile ("csrw 0xb00, x0\n");

    return;
}
```

Listing 5.14: Routine to disable the mcycle counter.

```
static inline void mcycle_disable()
{
    // mcountinhibit is at address 0x320
    asm volatile ("csrr t0, 0x320\n");
    // the bit to set is the LSB
    asm volatile ("ori t0, t0, 0x01\n");
    // write back the register content
    asm volatile ("csrw 0x320, t0\n");

    return;
}
```

Listing 5.15: Routine to enable the mcycle counter.

```
static inline void mcycle_enable()
{
    // mcountinhibit is at address 0x320
    asm volatile ("csrr t0, 0x320\n");
    // the bit to reset is the LSB
    asm volatile ("andi t0, t0, 0xfffffffe\n");
    // write back the register content
    asm volatile ("csrw 0x320, t0\n");

    return;
}
```

Due to length of overall execution of these benchmarks, in order not to deal with overflow phenomena, the `mcycle` counter is always reset before usage. C-like routines used to manage the machine cycle register are presented in Listing 5.12 to read the mcycle register, in Listing 5.13 to reset the mcycle register, in Listing 5.14 to disable the counting process, and in Listing 5.15 to enable the counting process.

5.3 Analysis of results

Moving to the benchmark execution results, these are presented with the following criteria. For each subsection one single benchmark of the three in Section 5.1 is considered. In the first subsection, the first comparison targets performance improvements of MAC instructions with respect to instructions supported by the original Ibex core (32 bit MUL and ADD), while the second comparison selects as a baseline the execution time of algorithms using 32 bit MAC instructions, comparing them with the custom MAC SS/ST instructions. For the other two subsections, instead, the baseline becomes only the algorithm using MAC instructions. All comparisons are made with the execution time of algorithms exposed in Section 5.1, and between multiplier units having comparable performance, specifically, the *Fast* and the *SingleCycle* versions are compared independently.

5.3.1 Fully-connected layer benchmark results

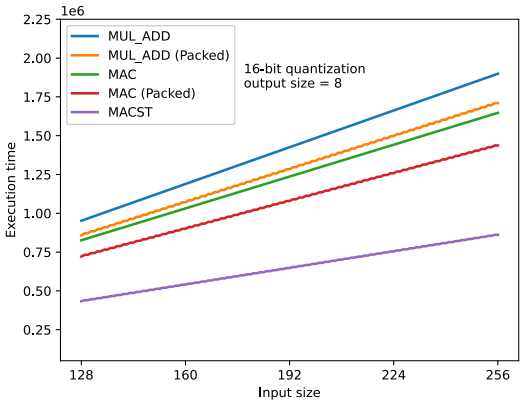
Figure 5.3 includes the execution time results coming from the execution of FC layer algorithms on the Ibex core using standard RISC-V M-extension instructions, MAC custom instructions, and sub-word parallel MAC instructions which support comes from the gate-level precision-scalable multiplier unit. On the vertical axis we have the execution time, while on the horizontal axis we have the number of inputs. The number of outputs is kept fixed in each graph, and is equal to 8 for Figures 5.3a-b-c, while is equal to 32 for Figures 5.3d-e-f. As can be seen, execution time proportionally increases as the number of output is increased, for this reason graphs with the output number equal to 16 have not been included.

In Figure 5.3a and Figure 5.3d, reporting execution time for the 16 bit quantization case, we can consider the baseline being the *MUL_ADD* blue line, thus appreciating the performance increment coming both from the usage of MAC instructions with the *MAC* green line, and from the memory access performed in packets, with the *MUL_ADD (Packed)* orange line. The red line, *MAC (Packed)*, refers instead to a union of these two strategies by using both a packed memory access strategy and MAC operations. Considering now the MAC-related algorithm (orange and red lines) as a baseline, we can derive the increment of using the custom MAC ST instructions. The purple line, being the *MACST* algorithm, have an average decrement of execution time of 1.91× with respect to MAC version (green line), and of 1.66× with respect to the MAC with memory accessed in packets (red

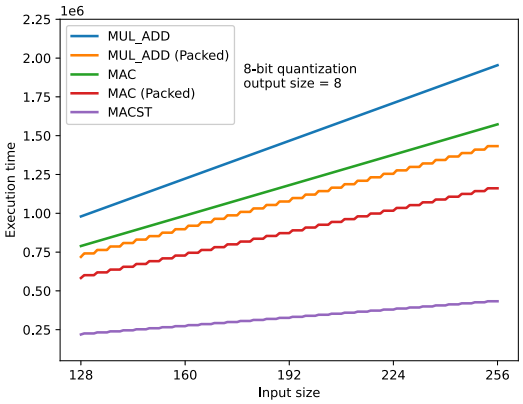
line). The theoretical increment at 16 bit coming from ST operations is equal to 2, but, having each instruction an execution ratio of 3/2 between the MAC and the MAC ST instruction, leads to a theoretical improvement equal to $2 \cdot (3/2) = 3$. Clearly, in the execution of the algorithms, there are not only MAC related instructions, but some complexity comes also from load, shift and mask operations, from the increment operation of the loop variable, and from the loop exit condition check, all of which are instructions that, of course, have not been accelerated, and that contribute in lowering the relative performance increment. From the graph, we can also see the increment in execution time between different output neurons numbers is linear.

Moving to Figure 5.3b, which present results for the 8 bit quantization case, we see now, loading memory in packets gives a higher improvement than the one given by MAC instructions, even considering the overhead to unpack the loaded words.

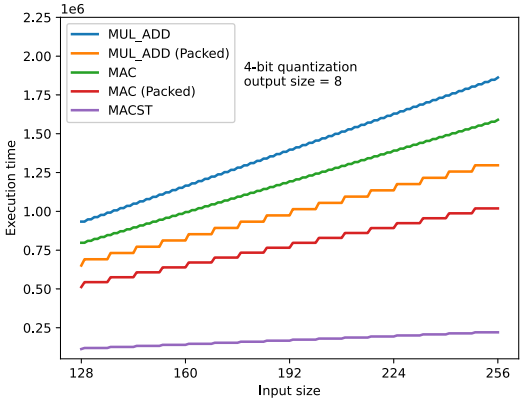
Figure 5.3



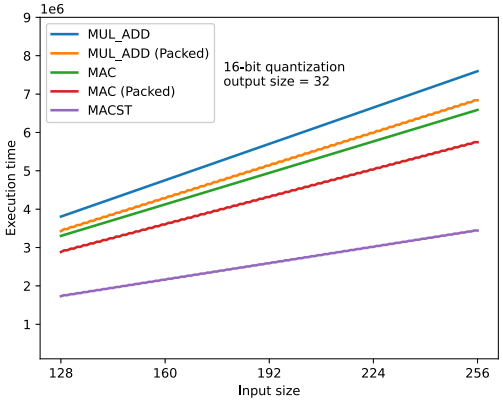
(a) FC layer execution time, *Fast*, 16 bit quantization, 8 outputs.



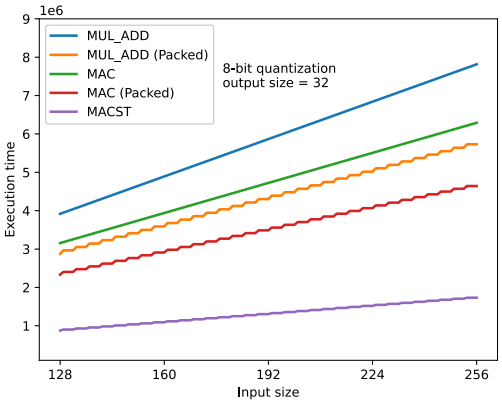
(b) FC layer execution time, *Fast*, 8 bit quantization, 8 outputs.



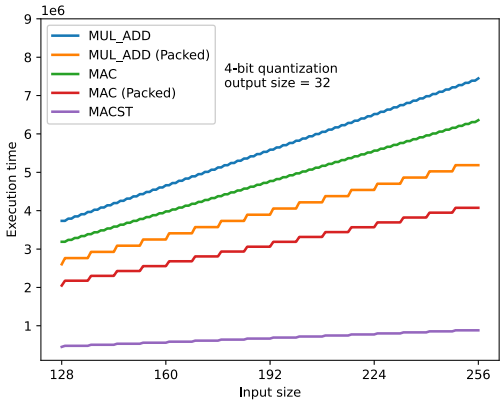
(c) FC layer execution time, *Fast*, 4 bit quantization, 8 outputs.



(d) FC layer execution time, *Fast*, 16 bit quantization, 32 outputs.



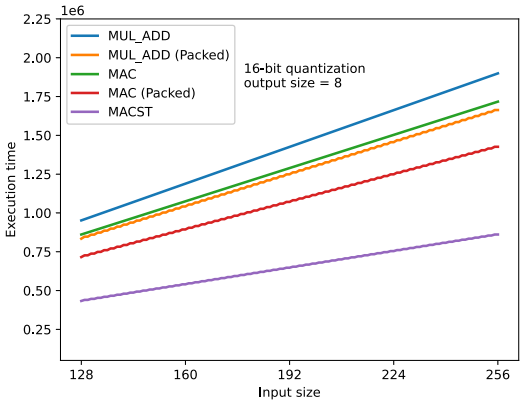
(e) FC layer execution time, *Fast*, 8 bit quantization, 32 outputs.



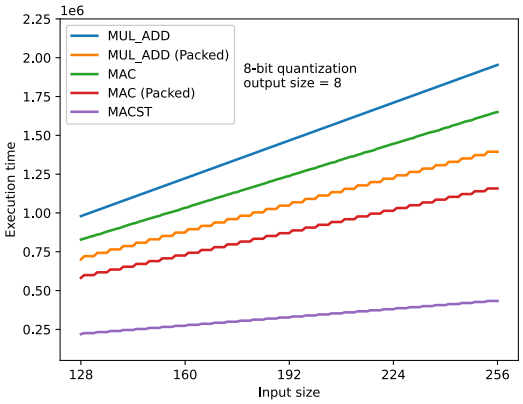
(f) FC layer execution time, *Fast*, 4 bit quantization, 32 outputs.

Theoretical improvement for this case derives from the execution of four operations in parallel, plus the addition 3/2 execution rate, leading to a result of $4 \cdot (1/2) = 6$. Considering as a baseline the red line of the *MAC (Packed)* algorithm, by using custom MAC ST instructions we achieve an improvement of 3.6× without packet loads (green line), and 2.68× with packet loads (red line). Moreover, here it is easier than the previous case to see the different behavior in accessing the memory by 8 bit each, with respect to accessing the memory in packets, specifically, we see we have an increment in the execution time only when the input size became a multiple of the number of operands contained in a packet. For example, in the *MACST* algorithm, varying the number of layers from 129 to 132 leads to the same execution time, while for 133 up to 136 it requires one more cycle. This is the reason of some lines being similar to a step-function.

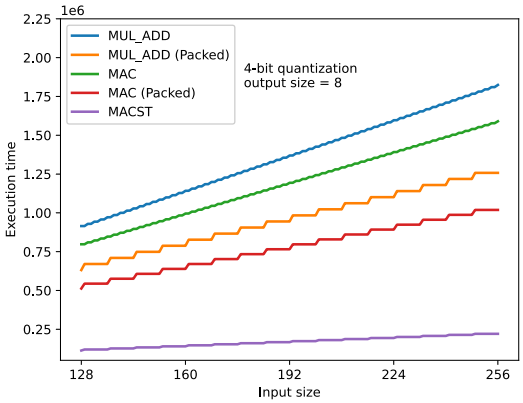
Figure 5.4



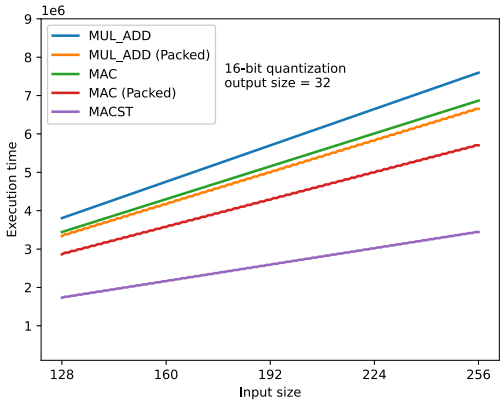
(a) FC layer execution time, *SingleCycle*, 16 bit quantization, 8 outputs.



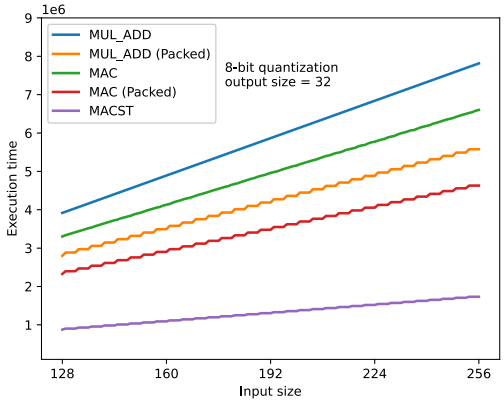
(b) FC layer execution time, *SingleCycle*, 8 bit quantization, 8 outputs.



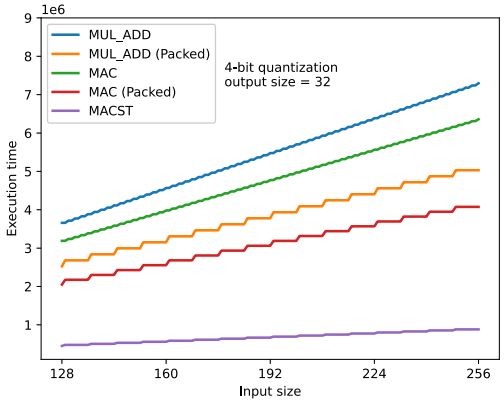
(c) FC layer execution time, *SingleCycle*, 4 bit quantization, 8 outputs.



(d) FC layer execution time, *SingleCycle*, 16 bit quantization, 32 outputs.



(e) FC layer execution time, *SingleCycle*, 8 bit quantization, 32 outputs.



(f) FC layer execution time, *SingleCycle*, 4 bit quantization, 32 outputs.

Finally, by looking at image 5.3c, the theoretical improvement is $8 \cdot (3/2) = 12$, but, as expected, the final improvement for this layer is lower, with the *MACST* line (in purple), performing 7.14× better than the *MAC* line (in green) and 4.58× better than the *MAC (Packed)* line (in red). Moreover, here, given the minimum memory access being 8 bit, also the *MUL_ADD* and the *MAC* variations access a single 8 bit packet unpacking it then, thus a little "step-function" behavior is visible. Figures 5.3e and 5.3f behaves the same as the ones with 8 output neurons.

Moving to the execution of FC layer algorithms on the *SingleCycle* multipliers, we obtain similar graphs. In Figure 5.4, proportionally the same considerations could be done. Specifically, by looking at Figure 5.4a (16 bit case), performance improvement is 1.65× in average with respect to the red line. The same happens also in Figure 5.4b (8 bit case) where it shows a 2.66× improvement, while for 5.4c (4 bit case) the improvement is 4.58 ×, all of them with respect to the red line. Moreover, the *SingleCycle* multiplier executes ST operations in one clock cycles,

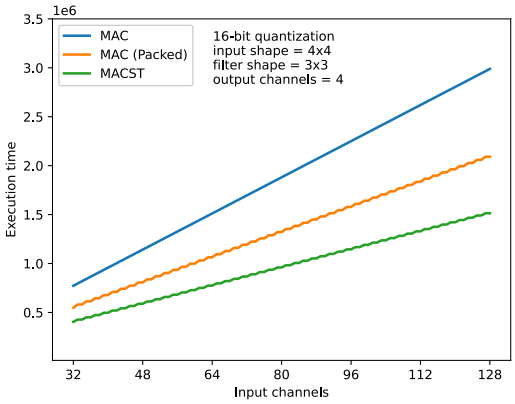
the same as a 32 bit operation, so there is no-more the 1.5 ratio between execution of these two types of instructions.

5.3.2 Convolutional layer benchmark results

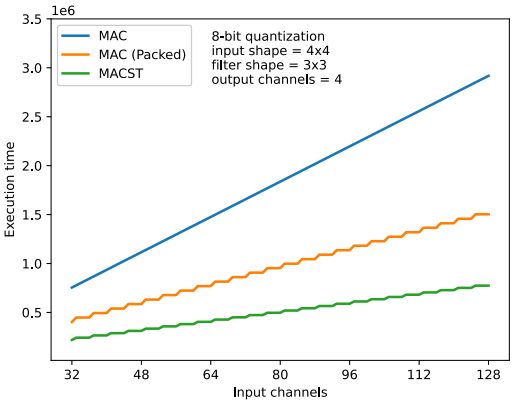
In this subsection results coming from the execution of the CNN layer algorithms is exposed. Only three algorithms are considered, the *MAC* algorithm making use of MAC operations, the same version with memory accessed in packets, and finally the version optimized to use custom precision-scalable ST instructions.

In Figure 5.5 we have results of the three algorithms running on the *Fast* multiplier. Specifically, in Figure 5.5a, that is the 16 bit quantization case, with an input feature map of 4x4 and a filter of 3x3, we can see the *MACST* algorithm having a 1.37× increment in performance, this is less than the FC case, but having the convolutional layer algorithm a higher number of nested loops, the non-accelerated overhead is less negligible than before.

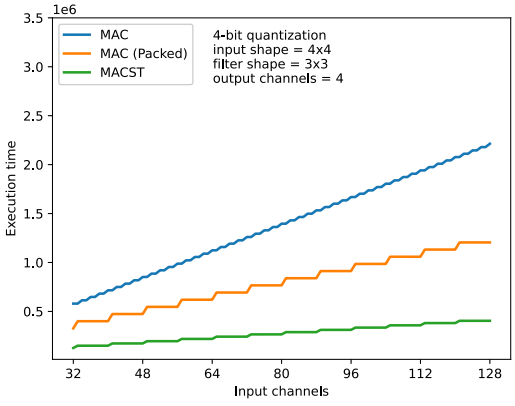
Figure 5.5



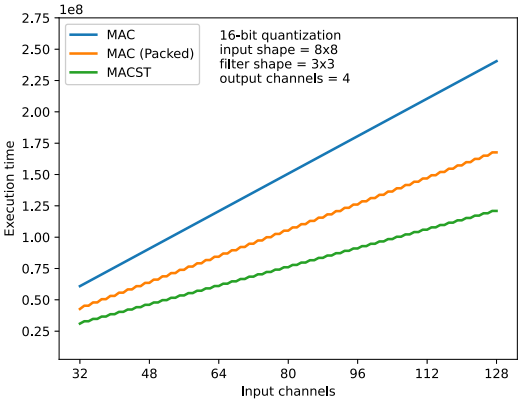
(a) CNN layer execution time, *Fast*, quant = 16 bit, in = 4x4, fil = 3x3.



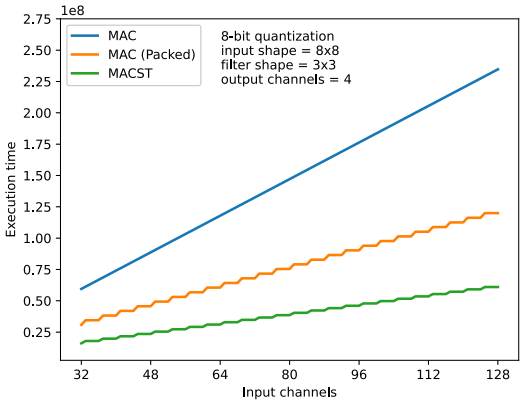
(b) CNN layer execution time, *Fast*, quant = 8 bit, in = 4x4, fil = 3x3.



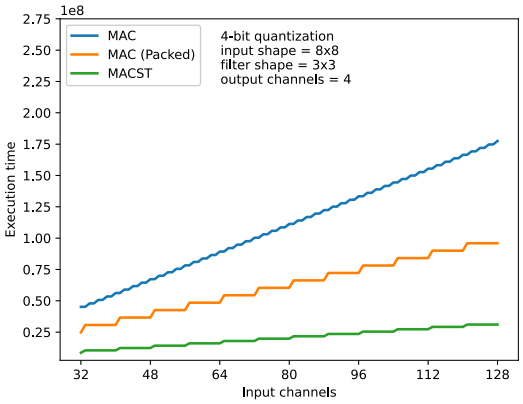
(c) CNN layer execution time, *Fast*, quant = 4 bit, in = 4x4, fil = 3x3.



(d) CNN layer execution time, *Fast*, quant = 16 bit, in = 8x8, fil = 3x3.

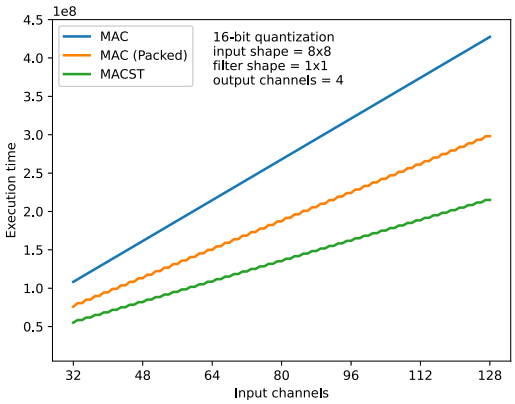


(e) CNN layer execution time, *Fast*, quant = 8 bit, in = 8x8, fil = 3x3.

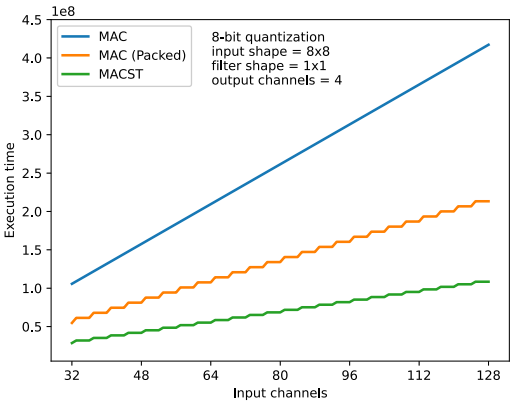


(f) CNN layer execution time, *Fast*, quant = 4 bit, in = 8x8, fil = 3x3.

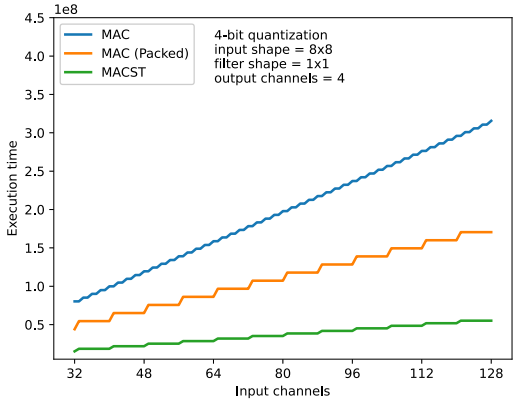
Figure 5.6



(a) CNN layer execution time, *Fast*, quant = 16 bit, in = 8x8, fil = 1x1.



(b) CNN layer execution time, *Fast*, quant = 8 bit, in = 8x8, fil = 1x1.

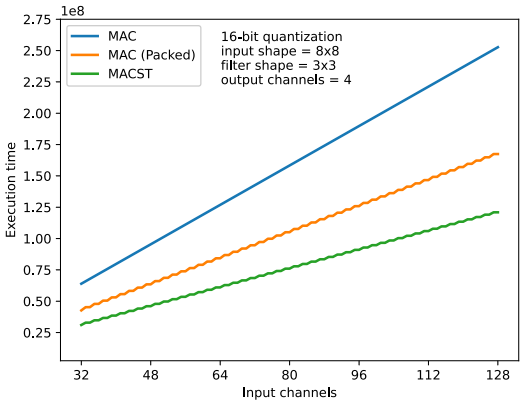


(c) CNN layer execution time, *Fast*, quant = 4 bit, in = 8x8, fil = 1x1.

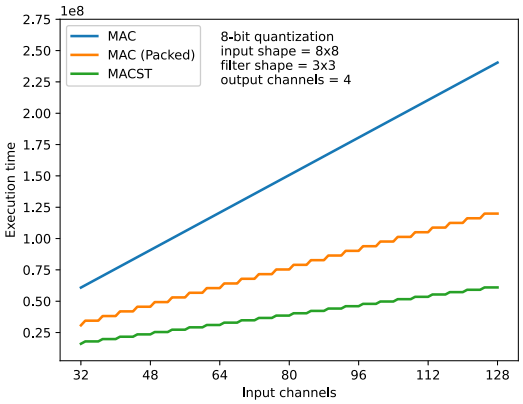
A very similar ratio of 1.38× is present in Figure 5.5d, which changes the input feature to 8x8, meaning that, for bigger inputs, the overhead is almost constant in changing the loop cycle number. In Figure 5.5b, analyzing the case of 8 bit quantization, we have an improvement of 1.88×, while for 5.5c the ratio grows up to 2.98× (the minimum is 2.57×). From these two cases we can see the improvement is less sharp than the FC layer one, in more aggressive quantization cases. Finally, as already discussed, increasing the input shape leads to proportionally similar performance, and this is the case of Figures 5.5e and 5.5f.

Another tested configuration of hyperparameters of the CNN layer, sets the filter shape to 1x1. In this case, results are visible in Figure 5.6. In detail Figure 5.6a, for the 16 bit quantization, shows a performance increment of 1.38×, the same as the 3x3 case. Figures 5.6b and 5.6c, for the 8 bit and 4 bit quantization cases, shows a performance increment up to 1.96× and 3.01×, sightly better than before.

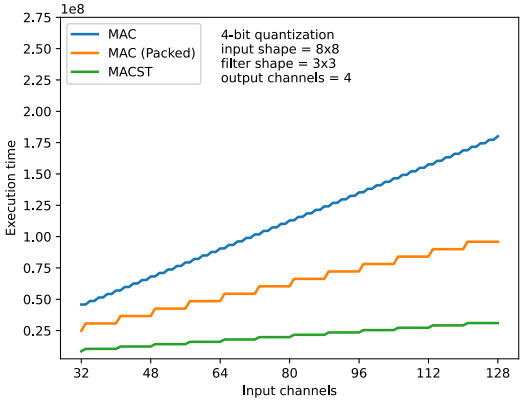
Figure 5.7



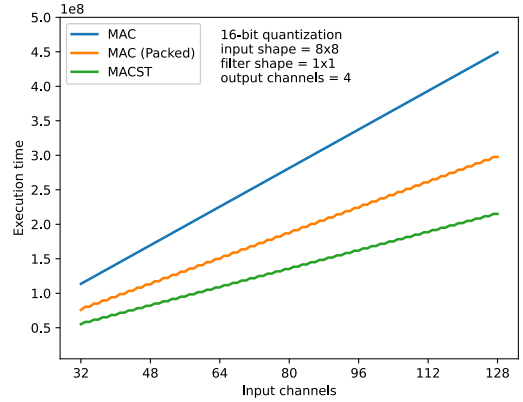
(a) CNN layer execution time, *SingleCycle*, quant = 16 bit, in = 8x8, fil = 3x3.



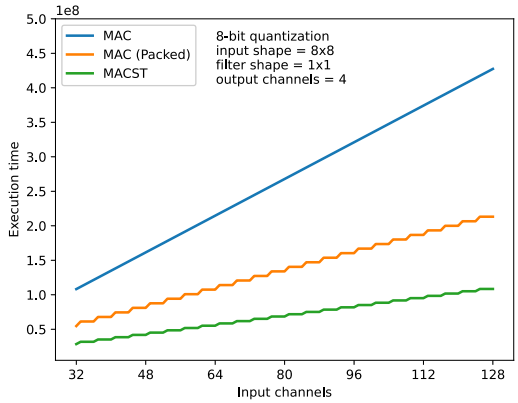
(b) CNN layer execution time, *SingleCycle*, quant = 8 bit, in = 8x8, fil = 3x3.



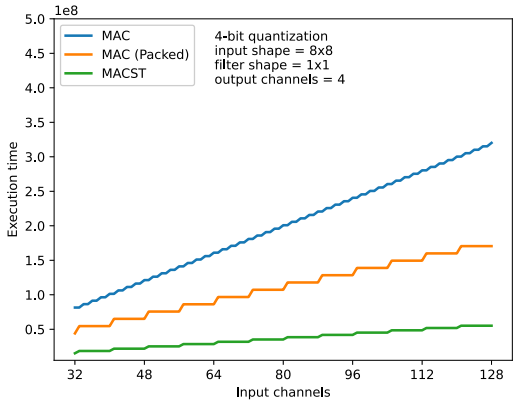
(c) CNN layer execution time, *SingleCycle*, quant = 4 bit, in = 8x8, fil = 3x3.



(d) CNN layer execution time, *SingleCycle*, quant = 16 bit, in = 8x8, fil = 1x1.



(e) CNN layer execution time, *SingleCycle*, quant = 8 bit, in = 8x8, fil = 1x1.

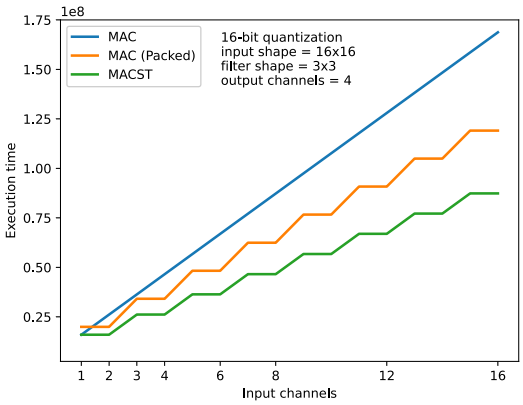


(f) CNN layer execution time, *SingleCycle*, quant = 4 bit, in = 8x8, fil = 1x1.

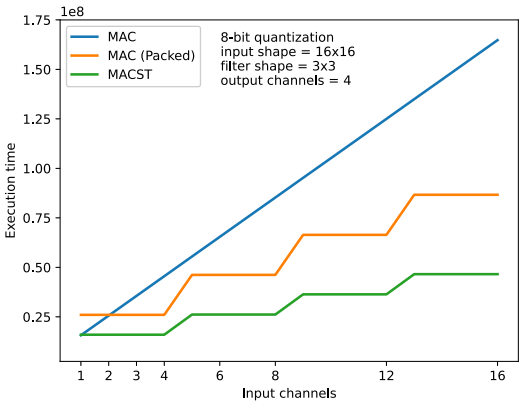
Moving now to the execution on the *SingleCycle* multiplier unit, only results having an input shape of 8x8 are reported in Figure 5.7. The 16 bit quantization strategy shows an improvement of 1.36× considering both Figure 5.7a for the 3x3 filter case and Figure 5.7d for the 1x1 filter case. As per the 8 bit quantization, Figures 5.7b and 5.7e, the ratio is a 1.93× performance improvement. Last, in Figure 5.7c and Figure 5.7f, representing the 4 bit case, we have a performance increment of 2.98×.

Previous results are related to the case of a CNN layer being used as a layer in middle/end part of a network. Now I present an analysis of what happens if the number of channels is reduced. The following results consider a number of channels

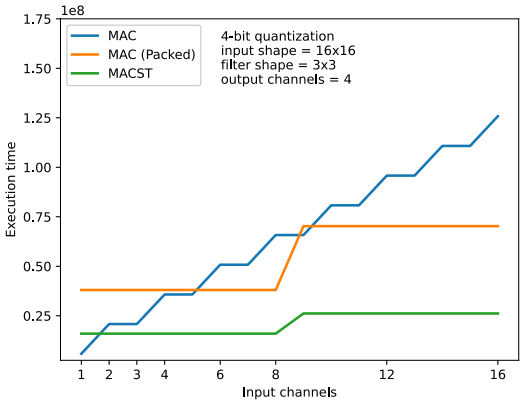
Figure 5.8



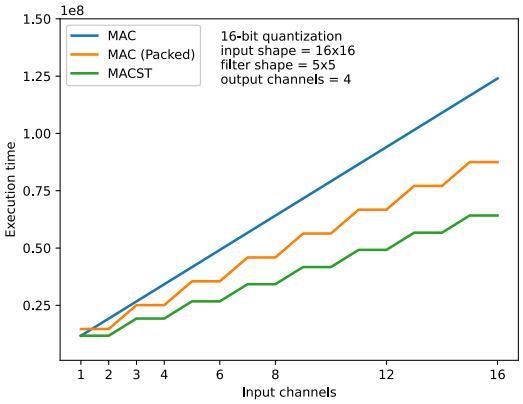
(a) CNN layer execution time, *Fast*, quant = 16 bit, in = 16x16, fil = 3x3.



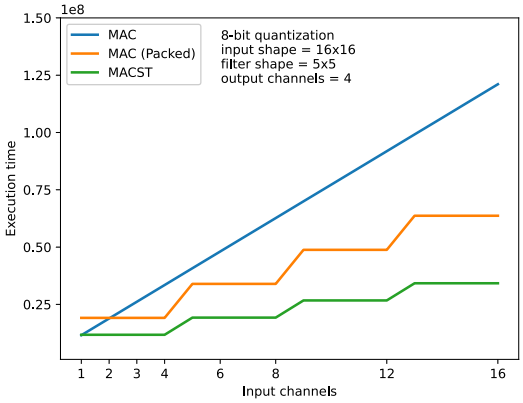
(b) CNN layer execution time, *Fast*, quant = 8 bit, in = 16x16, fil = 3x3.



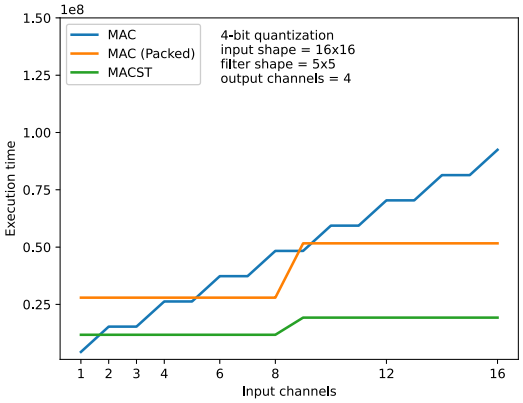
(c) CNN layer execution time, *Fast*, quant = 4 bit, in = 16x16, fil = 3x3.



(d) CNN layer execution time, *Fast*, quant = 16 bit, in = 16x16, fil = 5x5.



(e) CNN layer execution time, *Fast*, quant = 8 bit, in = 16x16, fil = 5x5.



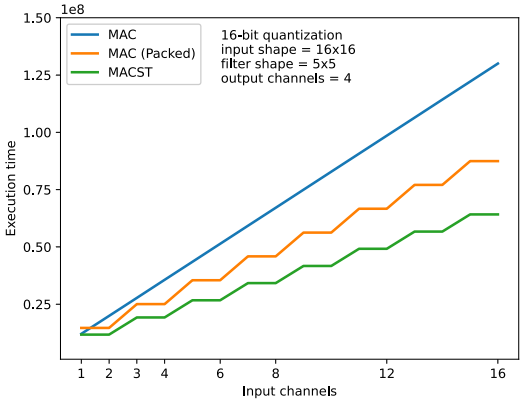
(f) CNN layer execution time, *Fast*, quant = 4 bit, in = 16x16, fil = 5x5.

from 1 to 16, although having an input feature map relatively small and limited to 16x16. This comes from the fact a 32x32 input shape leads already to an overflow of the internal counter when reading the lower 32 bit as done in previous cases; the upper 32 bit of the mcycle counter can be retrieved as well, but changing the test methodology for conditions in which we already saw, in many cases, the execution time increment grows proportionally it is not worth.

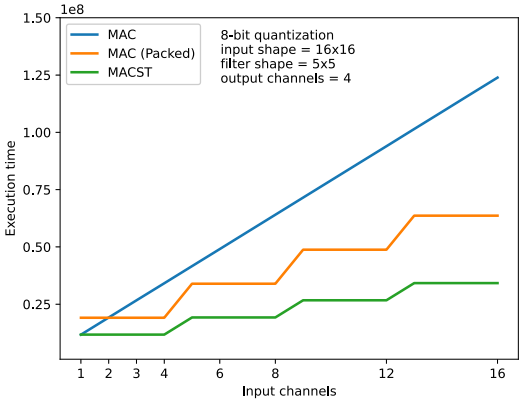
Having considered this, Figure 5.8 shows results of the execution on the *Fast* multiplier unit. In particular, from Figure 5.8a, we can see the case of few channels give a reduced performance increment, which is even 1× with a single channel, compared with algorithms not using parallel execution at all. Indeed, for a single channel there is no-parallelism possible along the channel dimension. Moreover, the loop overhead is almost null, since having a single channel means the number of loop cycles is equal to 1 as well. Increasing the channel number to 2 we can already appreciate an increment for the *MAC* algorithm having to perform two times the number of cycles as before; by the time channels increases up to 16 we can appreciate

an increment in the order of 1.36×. Also in Figure 5.8b, for the 8 bit quantization, we have a similar behavior, here the increment is more evident due to the fact parallelization takes 4 operations at most. Moreover, from this figure we can see the increment is at maximum if the network structure definition, mainly the number of channels, follows the HW computation capabilities. I want to add also, the unrolling amount for *Packed* algorithms have been fixed to the same number of operations performed in parallel, but this have been done by hand, it is unlikely it would be done by a compiler without giving it additional constraints. Another interesting case is the 4 bit quantization in Figure 5.8c, where the standard *MAC* algorithm without packet loads (blue line), in the single channel case, performs even better than the optimized *MACST* algorithm using custom instructions. Fortunately, these extreme cases are very rare and, as already stated, they are present only in the very first layer of a given NN.

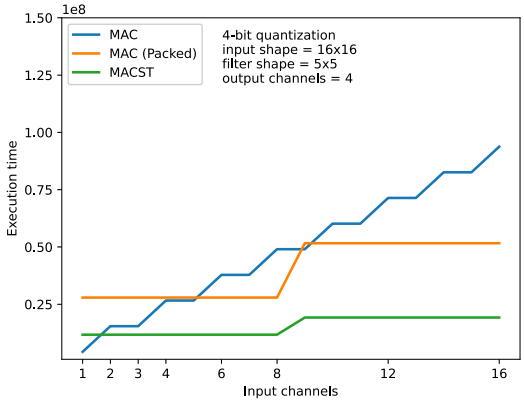
Figure 5.9



(a) CNN layer execution time, *SingleCycle*, quant = 16 bit, in = 16x16, fil = 5x5.



(b) CNN layer execution time, *SingleCycle*, quant = 8 bit, in = 16x16, fil = 5x5.



(c) CNN layer execution time, *SingleCycle*, quant = 4 bit, in = 16x16, fil = 5x5.

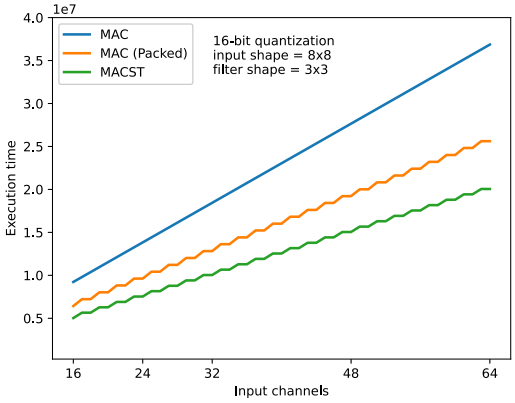
Finally, execution time with a 5x5 filter shape (Figures 5.8d, 5.8e, and 5.8f) shows perfectly comparable performance with respect to the 3x3 case.

Moving then to execution results on the *SingleCycle* multiplier, we have exactly the same behavior as with the *Fast* variant. In Figure 5.9 are reported the execution time results for the three quantization cases only with a 5x5 filter shape.

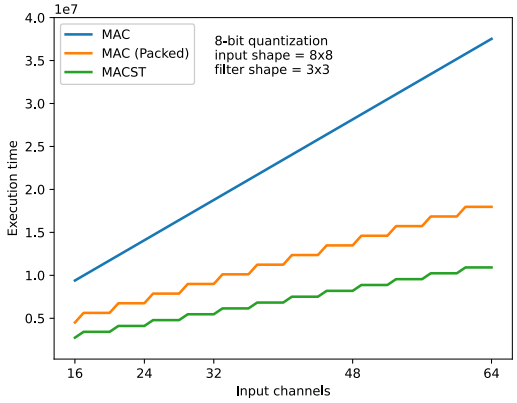
5.3.3 Depth-wise convolutional layer benchmark results

This subsection analyzes the execution time results of the DW-CNN layer. In Figure 5.10 graphs of the execution on the *Fast* multiplier unit are exposed. Figures 5.10a and 5.10d, having an input feature map of 8x8 and 16x16 respectively, include the execution time for the 16 bit quantization case, and both have a performance increment of 1.28× in average.

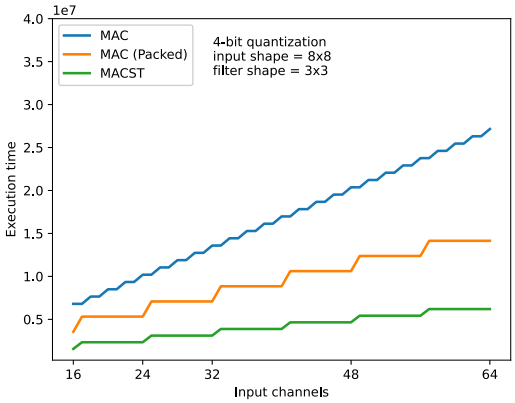
Figure 5.10



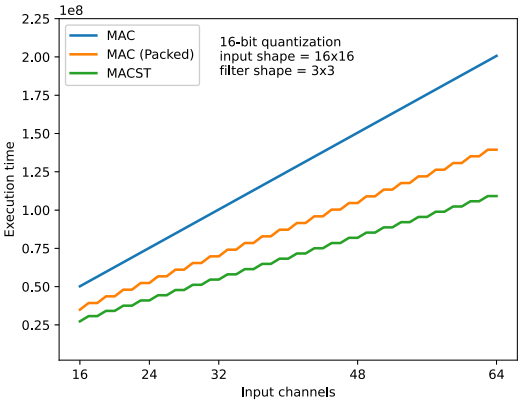
(a) DW-CNN layer execution time, *Fast*, quant = 16 bit, in = 8x8, fil = 3x3.



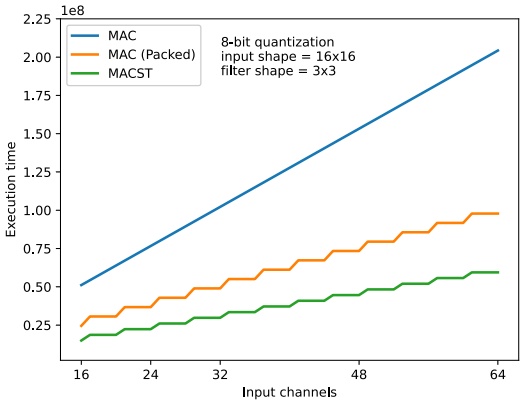
(b) DW-CNN layer execution time, *Fast*, quant = 8 bit, in = 8x8, fil = 3x3.



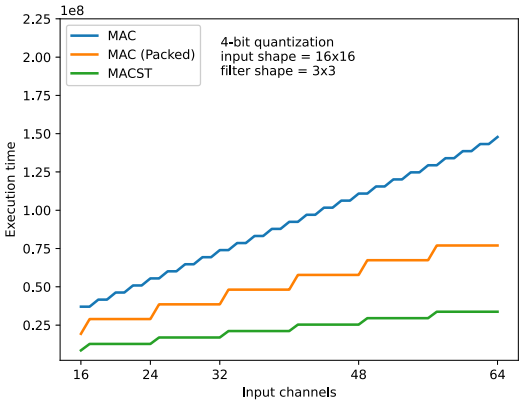
(c) DW-CNN layer execution time, *Fast*, quant = 4 bit, in = 8x8, fil = 3x3.



(d) DW-CNN layer execution time, *Fast*, quant = 16 bit, in = 16x16, fil = 3x3.



(e) DW-CNN layer execution time, *Fast*, quant = 8 bit, in = 16x16, fil = 3x3.

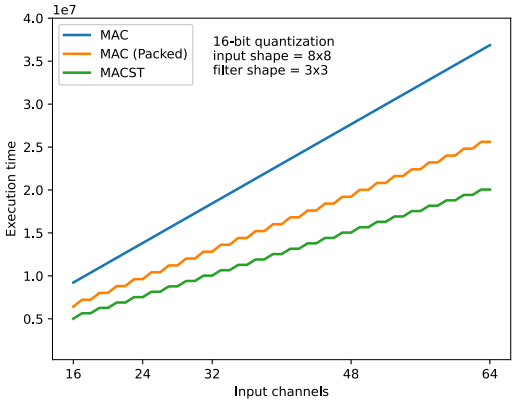


(f) DW-CNN layer execution time, *Fast*, quant = 4 bit, in = 16x16, fil = 3x3.

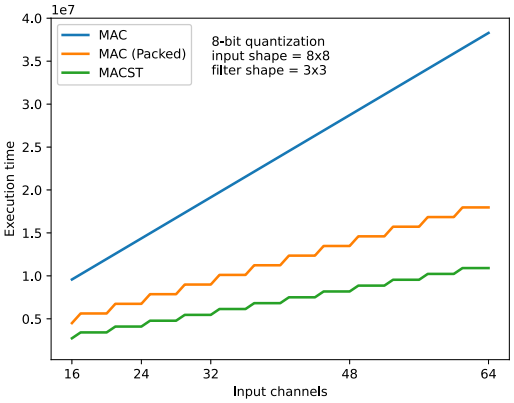
Figures 5.10b and 5.10e show instead results for the 8 bit quantization, also here increment is almost the same between these two runs, and it is in average 1.64×. Moving to Figures 5.10c and 5.10f, showing results of the 4 bit case, the performance increment is sharper, and is in average 2.28×.

As anticipated in Section 5.1.3 performance increment for this kind of layer is expected to be smaller, both due to keeping separated results of each kernel execution, but mainly due to the additional logic required using SS instructions in a way that tries to mimic the expected behavior for a hypothetical SIMD RISC-V compliant instruction, not considering instead requirements proper of this algorithm structure. This problem and possible countermeasures are left as a future work.

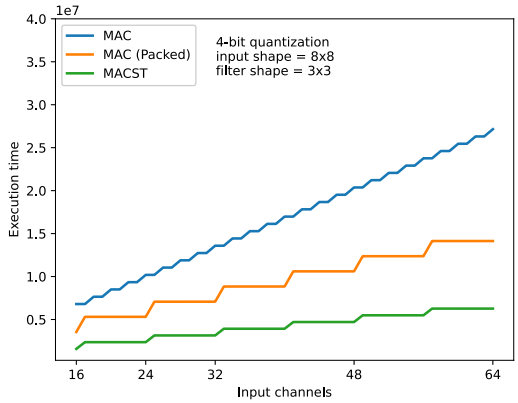
Figure 5.11



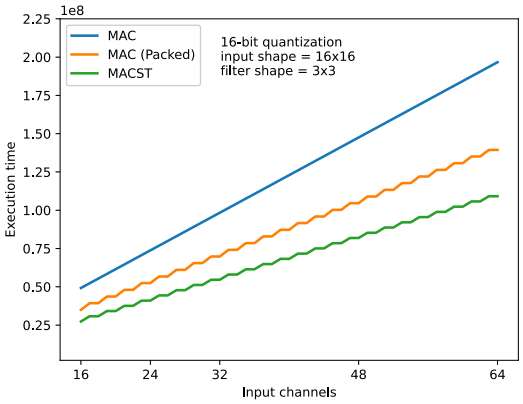
(a) DW-CNN layer execution time, *SingleCycle*, quant = 16 bit, in = 8x8, fil = 3x3.



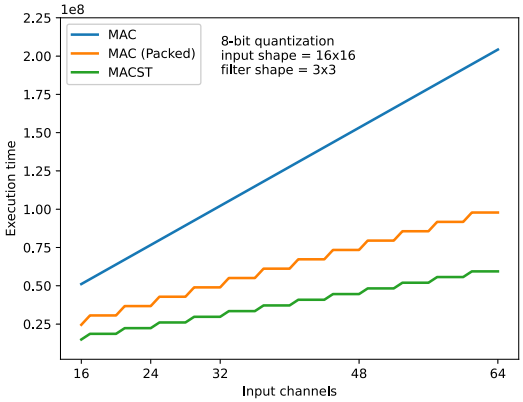
(b) DW-CNN layer execution time, *SingleCycle*, quant = 8 bit, in = 8x8, fil = 3x3.



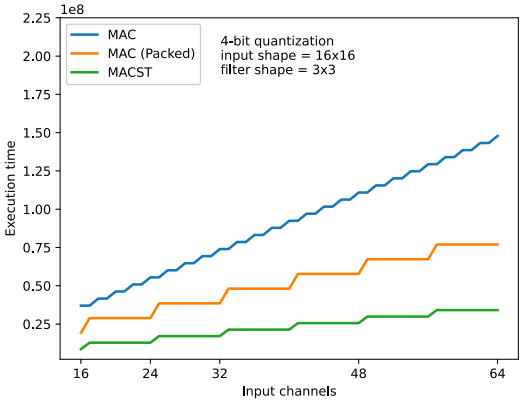
(c) DW-CNN layer execution time, *SingleCycle*, quant = 4 bit, in = 8x8, fil = 3x3.



(d) DW-CNN layer execution time, *SingleCycle*, quant = 16 bit, in = 16x16, fil = 3x3.



(e) DW-CNN layer execution time, *SingleCycle*, quant = 8 bit, in = 16x16, fil = 3x3.



(f) DW-CNN layer execution time, *SingleCycle*, quant = 4 bit, in = 16x16, fil = 3x3.

Finally, in Figure 5.11 the same graphs for the *SingleCycle* are reported. For each of the quantization and each of the input shape in analysis results are comparable with the previous case.

Chapter 6

Conclusions and future work

The first paragraph of this chapter illustrates some known-issues present in the actual design, evaluating meaning of these issues with respect of results obtained. The second paragraph is devoted to an overview and a conclusive analysis of this thesis work. The last section, concluding the thesis document, presents some possible interesting future work.

6.1 Known issues

The correctness analysis performed on the behavioral multiplier units using the HDL simulator Mentor Questa Sim gave a positive result. This is a cycle-accurate simulation of the entire SoC generated by ESP, including the modified Ibex core. Moving then to the same correctness analysis done on the AMD Xilinx Virtex proFPGA XC7V2000T, it showed non-correct values for some specific operations. Reasons behind this issue could be a different interpretation of the architecture description done by the HDL simulator Questa Sim, and by the Vivado synthesis tool, maybe on some interpretation boundaries left in the System Verilog language definition. This kind of problem have not shown in the two gate-level versions, which are cycle-accurate correct in both the FPGA and in the HDL simulation.

This is the main reason why, in Paragraph 5.3, the execution time of benchmarks running on behavioral multiplier units have not been presented. Even if the result was not correct, the execution time of the three benchmarks have been taken anyway. Results show an execution time that, in every case, overlaps with the one of the gate-level variant. In this view, the overall timing results can be considered coherent also in this situation, due to the fact that, even if results of the multiplication is wrong, the number of cycles to compute them on the FPGA is correct; thus even having a correct final result, the benchmark execution would very likely require the same amount of machine cycles.

Of course, this problem could also lightly affect synthesis results of the behavioral multiplier descriptions presented in Section 3.3, but, given the correctness being verified by the HDL cycle-accurate simulation, at least the actual trends and considerations done in that section could be considered valid.

6.2 Overall result analysis and considerations

In this thesis I developed a novel precision-scalable 16 bit multiplier based on the Baugh-Wooley structure, capable of accomplish computation required by Sum-Separate and Sum-Together operations from 16 bit down to 4 bit. SS and ST instructions are two different approaches to sub-word computation, and parallel execution of instructions using a single HW structure. Additions to the original Baugh-Wooley architecture have been step-by-step derived, to support both signed and unsigned operations, the previously mentioned precision-scalable instructions, and also MAC support for all multiplication instructions.

This structure has been then integrated in the low-power RISC-V Ibex core, replacing the two originally present multiplier units, with two gate-level units based on this precision-scalable design. I also developed two behaviorally described equivalent multiplier units for comparison. All the six multipliers, the two baseline, and the four modified, have been verified with an HDL simulator, and on FPGA (although with some non-consistent result for the behavioral ones). Moreover, these six versions have been synthesized targeting a 28 nm FDSOI technology node at 0.9 V, obtaining a metric on the cost, in terms of Power-Performance-Area trade-offs, at which the parallel execution comes. To summarize increments in the area for the gate-level *Fast* Ibex multiplier with respect to the original *Fast* version is around 10% from 100 MHz to 400 MHz, 13% between 500 MHz and 600 MHz and 20% at 700 MHz; the power consumption is between 5% and 6%, for frequencies up to 700 MHz. As per the gate-level *SingleCycle* the area increment is 15% from 100 MHz to 300 MHz compared to the original *SingleCycle* design, while it increases rapidly to 30% at 400 MHz and to 57% at 500 MHz, after which, for higher frequencies, the synthesis tool can not infer a compliant structure; as per the power consumption increment, it is around 5% up to 300 MHz, increasing to 8% and 12% for 400 MHz and 500 MHz respectively. The behavioral versions require both more area and power, specifically, the *Fast* one occupies between 30% and 40% more area than the original *Fast* design, power consumption is between 8% and 11%, while the *SingleCycle* version requires instead between 50% and 60% more area than the baseline *SingleCycle* one, and consumes between 11% and 17% more power along the frequency range considered.

The integration process in the Ibex core includes defining custom RISC-V instructions. These instructions have been added as an extension of the RISC-V

ISA, meaning that they have both been added in the Ibex core, modifying its decode unit, and in a compiler, modifying the GCC compiler for an assembly-level support of custom instructions. Moreover, both Ibex and GCC have to agree on the instruction format, which also have not to overlap with any other instruction already present in the supported RISC-V ISA parts (in this work, only instructions already supported by the Ibex core have been considered for possible overlappings).

To evaluate performance of each precision-scalable multiplier unit, three benchmarks have been developed, each one defining computational routines of the three most common NN layers for the edge domain, namely, Fully Connected layer, Convolutional layer, and Depth-Wise Convolutional layer. Execution time of these benchmarks have been taken, and results put in evidence advantages coming from the usage of custom instructions supported by precision-scalable descriptions. To isolate the precision-scalable contribution in performance increments with respect to the MAC operation addition, the main baseline for tests becomes routines making use of only custom 32 bit MAC operations. Results of execution shows an improvement for the FC layer up to 7.14× over an algorithm making use of the 32-bit MAC instructions, and 4.58× considering the same algorithm having optimizations such as memory accessed in word-length packets, and loop unrolling to mimic parallelism. The CNN layer computation shows instead a performance increment up to 5.80× with respect to the 32 bit MAC-based algorithm, while an increment up to 2.98× if compared with the unrolled version. Finally, for the DW-CNN layer an improvement up to 4.38× with respect to the MAC algorithm, and 2.28× considering the unrolled version of the same algorithm.

6.3 Future work

The first future work is to fix the issue present in the first paragraph of this chapter.

From the analysis of the synthesis reports, especially considering the area parameter, in both gate-level descriptions the synthesis tool does not manage to keep optimizing over a certain frequency that is below the target range considered from 100 MHz to 1000 MHz. A correction to this problem is reducing the critical path of the architecture. A good contribution to the critical path in Baugh-Wooley architectures comes from the final adder description. A countermeasure could be defining a behavioral adder as the final adder summing S_i and C_i signals in the last layer. A behavioral description could be optimized for a high-frequency scenario, by selecting an adder structure, from a library, that performs better than the ripple-carry adder, among which, parallel-prefix adders or others.

Following this solution, the second one is defining a gate-level parallel-prefix adder which precisely satisfies the separable requirement, both increasing performances and optimizing the area in the high-frequency domain with respect to behaviorally described solutions.

Talking about high frequency scenarios, there are different multiplier structures which could be also adapted to the precision-scalable approach. One example have been done in [21], in which ST operations support is added.

Amid RISC-V proposed extensions, the P-extension [63] targets 16 bit and 8 bit SIMD instructions, and have not been ratified yet. First, this work can be made compliant with instructions so far defined in the P-extension. Moreover, a proposal to expand this extension to the 4 bit SIMD instructions could be done, indicating trends in the QNN computation domain, which could requires this kind of instructions as a standard in near future.

Finally, a remarkable work could be using the proposed multiplier inside QNN dedicated ASIC accelerators, which have not to deal with strictly instruction formats of a standard microprocessor. Moreover, this kind of accelerators could target different QNN layer optimization strategies all in ones (exploiting both SS and ST operations), being more energy efficient, but paying the price of loosing some flexibility coming from the removal of general purpose computational parts.

List of Figures

Bibliography

[1] Reuther, Albert, et al. "AI and ML Accelerator Survey and Trends." 2022 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2022.

[2] Bishop, Christopher M., and Nasser M. Nasrabadi. Pattern recognition and machine learning. Vol. 4. No. 4. New York: springer, 2006.

[3] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

[4] Sze, Vivienne, et al. "Efficient processing of deep neural networks: A tutorial and survey." Proceedings of the IEEE 105.12 (2017): 2295-2329.

[5] Wu, Hao, et al. "Integer quantization for deep learning inference: Principles and empirical evaluation." arXiv preprint arXiv:2004.09602 (2020).

[6] Gholami, Amir, et al. "A survey of quantization methods for efficient neural network inference." arXiv preprint arXiv:2103.13630 (2021).

[7] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[8] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[9] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[10] Yvinec, Edouard, Arnaud Dapogny, and Kevin Bailly. "To fold or not to fold: a necessary and sufficient condition on batch-normalization layers folding." arXiv preprint arXiv:2203.14646 (2022)

[11] Song Han, Jeff Pool, John Tran, William J. Dally (2015). Learning both Weights and Connections for Efficient Neural Networks.

[12] Javier Fernandez-Marques, Paul N. Whatmough, Andrew Mundy, Matthew Mattina (2020). Searching for Winograd-aware Quantized Networks.

[13] Tom B. Brown et al. (2020). Language Models are Few-Shot Learners.

[14] N. H. E. Weste and D. M. Harris. "CMOS VLSI Design", 4th ed. Reading, MA: Addison-Wesley, 2011, ch. 11.

[15] Tu, Jin-Hao, and Lan-Da Van. "Power-efficient pipelined reconfigurable fixed-width Baugh-Wooley multipliers." IEEE transactions on computers 58.10 (2009): 1346-1355

[16] Lin, Rong. "Reconfigurable parallel inner product processor architectures."

IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9.2 (2001): 261-272.

[17] Camus, Vincent, et al. "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing." IEEE Journal on Emerging and Selected Topics in Circuits and Systems 9.4 (2019): 697-711.

[18] Mei, Linyan, et al. "Sub-word parallel precision-scalable MAC engines for efficient embedded DNN inference." 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, 2019.

[19] Sharma, Hardik, et al. "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network." 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2018.

[20] Urbinati, Luca, and Mario R. Casu. "A Reconfigurable Depth-Wise Convolution Module for Heterogeneously Quantized DNNs." 2022 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2022.

[21] Urbinati, Luca, and Mario R. Casu. "A Reconfigurable Multiplier/Dot-Product Unit for Precision-Scalable Deep Learning Applications." Proceedings of SIE 2022: 53rd Annual Meeting of the Italian Electronics Society. Cham: Springer Nature Switzerland, 2023.

[22] Urbinati, Luca, and Mario R. Casu. "A reconfigurable 2d-convolution accelerator for dnns quantized with mixed-precision." Applications in Electronics Pervading Industry, Environment and Society: APPLEPIES 2022. Cham: Springer Nature Switzerland, 2023. 210-215.

[23] Peemen, Maurice, et al. "Memory-centric accelerator design for convolutional neural networks." 2013 IEEE 31st international conference on computer design (ICCD). IEEE, 2013.

[24] Mittal, Sparsh. "A survey of FPGA-based accelerators for convolutional neural networks." Neural computing and applications 32.4 (2020): 1109-1139.

[25] Schiavone, Pasquale Davide, et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications." 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE, 2017.

[26] Garofalo, Angelo, et al. "Xpulpnn: Enabling energy efficient and flexible inference of quantized neural networks on risc-v based iot end nodes." IEEE Transactions on Emerging Topics in Computing 9.3 (2021): 1489-1505.

[27] O'Shea, Keiron, and Ryan Nash. "An introduction to convolutional neural networks." arXiv preprint arXiv:1511.08458 (2015).

[28] Duarte J. et al. (2018). Fast inference of deep neural networks in FPGAs for particle physics.

[29] Mantovani, Paolo, et al. "Agile SoC development with open ESP." Proceedings of the 39th International Conference on Computer-Aided Design. 2020.

[30] Eichler, Guy, et al. "MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces." 2021 IEEE 39th International Conference on Computer Design (ICCD). IEEE, 2021.
[31] Jia, Tianyu, et al. "A 12nm Agile-Designed SoC for Swarm-Based Perception with Heterogeneous IP Blocks, a Reconfigurable Memory Hierarchy, and an 800MHz Multi-Plane NoC." ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC). IEEE, 2022.
[32] Choquette, Jack, et al. "Nvidia a100 tensor core gpu: Performance and innovation." IEEE Micro 41.2 (2021): 29-35.
[33] https://developer.nvidia.com/machine-learning
[34] https://docs.openvino.ai/latest/home.html
[35] https://www.amd.com/en/graphics/servers-solutions-rocm-ml
[36] https://www.tensorflow.org/
[37] https://pytorch.org/
[38] https://openai.com/
[39] https://cloud.google.com/tpu
[40] https://cloud.google.com/edge-tpu
[41] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
[42] http://nvdla.org/
[43] https://xilinx.github.io/finn/
[44] https://fastmachinelearning.org/hls4ml/
[45] https://www.xilinx.com/products/boards-and-kits/1-4jkbxs.html
[46] https://www.arm.com/products/silicon-ip-cpu?families=cortex-m
[47] https://lowrisc.org/
[48] https://github.com/lowRISC/ibex
[49] https://ibex-core.readthedocs.io/en/latest/
[50] https://gcc.gnu.org/
[51] https://riscv.org/
[52] https://www.pulp-platform.org/
[53] https://github.com/riscv-collab/riscv-gnu-toolchain
[54] https://github.com/riscv/riscv-isa-manual/releases/
[55] https://github.com/riscv/riscv-bitmanip/releases/
[56] https://esp.cs.columbia.edu/
[57] https://esp.cs.columbia.edu/docs/rtl_acc/
[58] https://esp.cs.columbia.edu/docs/cpp_acc/
[59] https://esp.cs.columbia.edu/docs/thirdparty_acc/
[60] https://esp.cs.columbia.edu/docs/hls4ml/
[61] https://esp.cs.columbia.edu/docs/setup/
[62] https://esp.cs.columbia.edu/docs/singlecore/
[63] https://github.com/riscv/riscv-p-spec/