POLITECNICO
DI TORINO

Master Degree in Electronic Engineering

ASIP Implementations for Polar Codes Decoding using Simplified Successive Cancellations Algorithm and Instruction Level Parallelism

Politecnico di Torino

Turin, Italy

Student:
Sebastiano Strano

Supervisor:
Guido Masera

November 8, 2018

Summary

1 Introduction

Polar Codes have been introduced by Erdal Arıkan in 2008 as a class of error-correcting codes with the possibility to reach the capacity of a discrete memoryless channel. The concept of capacity of the channel was introduced by Shannon in 1948 and it refers to the upper limit of the rate at which information can be reliably transmitted over a communication channel. Arikan developed the concept of "channel polarization" in his paper [1] where, initially, a certain independent channel is transformed into two different kind of it: a high-level noise type and a low-level noise type. By recursively applying this technique the synthesized channels increase the noise-level difference between each other until, after an infinite number of recursion, a group of completely noiseless and a group of completely noisy channels are created. Arikan exploits this concept elaborating an encoding strategy where the noisy channels were assigned to constant bits (called frozen bits) while the noiseless channels were used to send the actual information bits. On the other side, the decoder knows a priori the positions of the frozen bits and runs the so called "Successive Cancellations" (SC) algorithm in order to exploits the correlation among the source bits and correct the errors due to the noise. Successive Cancellation algorithm have been evolved into "Simplified Successive Cancellations" (SSC) and "Successive Cancellations List" (SCL) algorithms that present better performances maintaining the error-correcting capabilities.
In the present, Polar Codes are being used in the fifth generation of mobile technologies substituting the "turbo-like" code families that represented the main alternative for 3GPP, WCDMA and LTE standards.

The purpose of this dissertation is to design an application-specific in-

struction set processor (ASIP) able to efficiently decode the received symbols using the SSC decoding algorithm. At the beginning, a simple architecture based on a single arithmetic logic unit (ALU) will be implemented where the different decoding codes will be tested. Then, five designs with different parallelism degrees will be proposed. In each design several problems will be faced concerning all the different blocking elements that limit the speed-up introduced by the multiple arithmetic units. Every architecture will be characterised with the aim of offering different basis for comparisons between each design.

1.1 Why an ASIP?

An application-specific instruction set processor is basically a processor whose instruction set has been tailored to a specific application. Therefore, the main purpose of an ASIP is to provide better performance than a general purpose processor but maintaining a certain flexibility. This flexibility is given by the possibility to modify the Program memory of the device changing the algorithm run by it without changing the datapath. This is a very important feature because when a bug or a more efficient way to implement the code is found, the modifications can be done without producing a different device but simply updating the memory content.

An ASIP also provides a lower power consumption than general purpose processors. The instructions run by the ASIP are perfectly designed in order to accomplish a well define job. For this reason, if a certain operation needs several clock cycles in order to be completed by a general purpose processor the same operation can be potentially done using only one cycle if it is run by a purposely designed ASIP. Thus, if less cycles are used in order to complete an operation less energy is waisted during the computations. In addition,

an ASIP application also allows to better control the exchanges between the arithmetic unit and the memory. In consequence, if the number of memory accesses is reduced the power consumption can be decreased too.

2 Literature Review

2.1 General Transmission Scheme

Polar codes are based on a forward error-correction scheme shown in Fig. 1. This scheme is able to correct the errors due to the channel noise without re-communicating with the source but exploiting the correlation created by the encoding step.

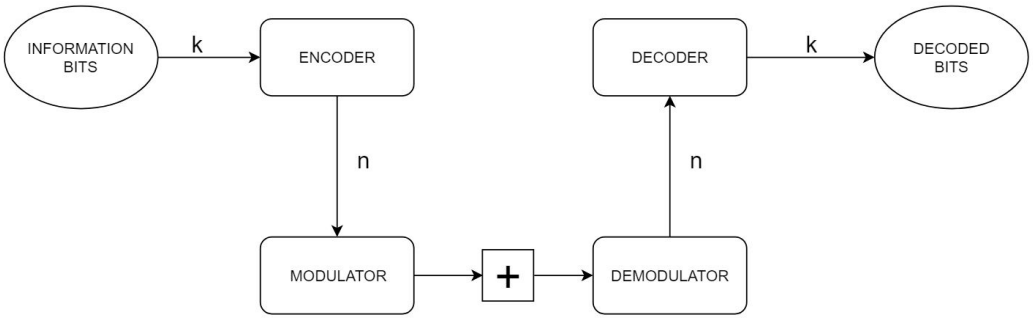

Figure 1: General transmission scheme for Polar Codes

The information bits represent the data that must be transmitted. These bits are elaborated by an encoder which returns a bit array longer than the original one. The additional bits are the result of the encoding step and are exploited in order to create the correlation. From now on "N" represents the code length while "k" represents the information length. These two variable are used to compute the code ratio "R" which is the ratio between the information bits and the code length, as shown in equation 1. This ratio is very important because it expresses the amount of "redundancy" added to the encoded bits and so the error-correcting capability.

$$R \triangleq \frac{k}{N} \tag{1}$$

Therefore, when the code rate is lower, the noise rejection will be higher because the decoder is able to exploit more redundancy in order to correct the possible errors.

After the encoding step, a modulator elaborates the symbols that are transmitted through the channel. This last element is represented by the "+" character in the scheme because when the symbols travel the channel they are summed to some random noise. At the end of the channel, the symbols are received by the demodulator which transforms them into log-likelihood ratios (LLR).

The likelihood ratio, for a binary code, represents the ratio between the probability that a single transmitted bit x_i is 0 given the received symbol y_i over the probability that the same transmitted bit is 1 (equation 2):

$$LR_{y_i} \triangleq \frac{P(y_i|x_i = 0)}{P(y_i|x_i = 1)} \tag{2}$$

$$LLR_{y_i} \triangleq log\left(\frac{P(y_i|x_i = 0)}{P(y_i|x_i = 1)}\right) \tag{3}$$

The log-likelihood ratio is basically the same thing but inside a logarithm in order to simplify some decoding operations as will be explained later.

The decoded bits are finally obtained from the LLRs computed by the decoder unit using one of the previously mentioned algorithms. The decoded bits are then ordered into a k-bits array which represents the sent information bits.

2.2 Successive Cancellation Decoding Algorithm

The SC algorithm was originally proposed by Arikan in his paper [1]. It consists in a soft/hard message passing scheme over a binary trellis which

can be extracted by the regular structure of the Polar Codes. Fig. 2 shows the trellis diagram for a code with N = 8.

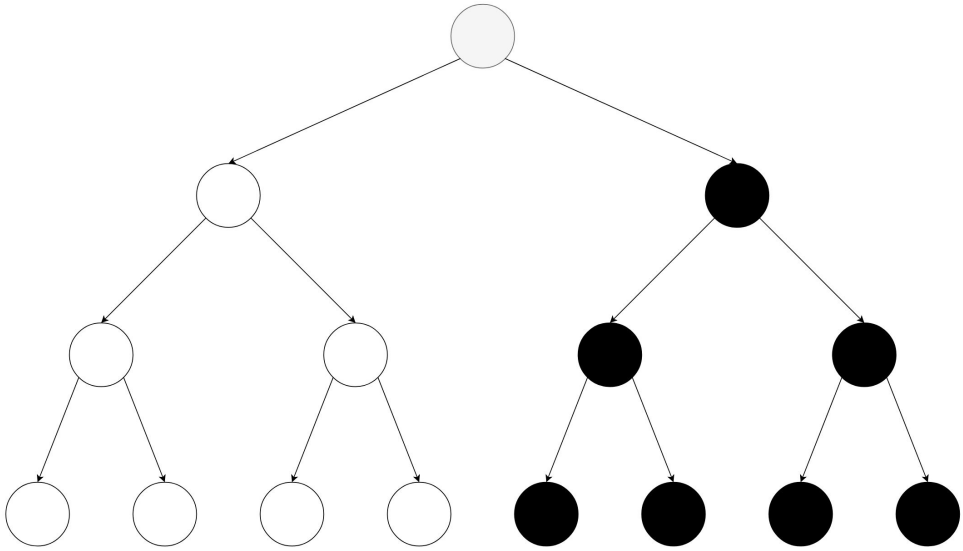

Figure 2: Trellis diagram with N = 8

The trellis presents $log_2(N) = 3$ stages where it forks until the last N leafs are obtained. The information bits can be computed performing a first forward iteration from the top of the tree to the button and a second backward iteration from the button of the tree to the top. In the forward iteration the so called "soft" inputs are computed using the soft inputs of the higher leaf while in the backward iteration an hard decision is made using the values coming from the lower leafs.

An hardware implementation of the trellis can be obtained using three different computational nodes: a F node representing the forward operations, a G node representing the backward operations and an H node representing the hard decisions. The decoding scheme with an hardware implementation for a code with N = 8 is displayed in Fig. 4. The LLR values at the right of the scheme are the log-likelihood ratios coming from the demodulator while

the "u" values at the left of the scheme represent the decoded bits.
There are two different way to implement the SC decoding algorithm:

- Message passing: based on the structure shown in Fig. 3.
- Line SC: based on the scheme shown in Fig. 4. It is an hardware-friendly version, as mentioned before.

2.2.1 Message passing

In the Message Passing SC decoding algorithm, each node of the trellis has a parent and two children: one right child and one left child. Each parent exchanges two types of message vectors with the children : a soft input called "A" in Fig. 3 and and a binary-valued output called "B". The length of the vectors depends on the level where a certain node is located. The A_v and B_v vectors of the root node have a length equal to 2^{N_s-1} where N_s is the base two logarithm of the code length. The lower nodes have vectors length equal to 2^{N_s-1-l} where l is the level difference between the root node and the position of the same node. This means that the nodes located at the last level of the tree exchange bit vectors of length equal to one with the parent.

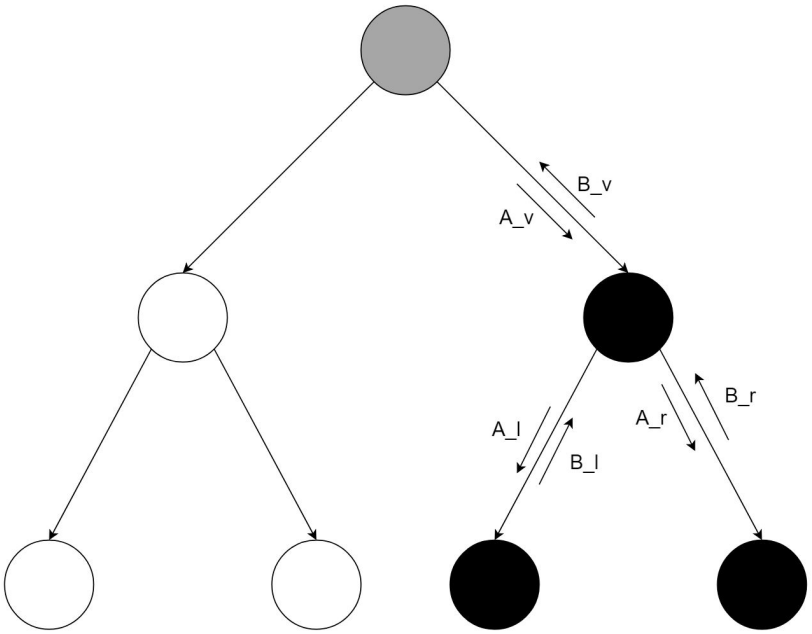

Figure 3: Decoding scheme of Message Passing SC algorithm

The A_l array is the first that can be computed once the father receives the A_v array from his parent. The A_l elements of the vector are computed using the following formula:

$$A_l[i] \triangleq 2tanh^{-1}\left(tanh\left(\frac{A_v[i]}{2}\right) \cdot tanh\left(\frac{A_v[2^{N_s-1-l}]}{2}\right)\right) \quad for\ 0 \leq i < 2^{N_s-1-l} \tag{4}$$

After the A_l vector reception, the left child returns the B_l vector with whom is possible to compute the A_r vector which is defined as follows:

$$A_r[i] \triangleq (1 - 2B_l[i]) \cdot A_v[i] + A_v[2^{N_s-1-l} + i] \quad for\ 0 \leq i < 2^{N_s-1-l} \tag{5}$$

This time the right child will return the B_r vector which is needed to compute the B_v values in the following way:

$$B_v[i] \triangleq \begin{cases} B_l[i] \oplus B_r[i] \quad for\ 0 \leq i < 2^{N_s-1-l} \\ B_l[i] \quad for\ 2^{N_s-1-l} \leq i < 2^{N_s-l} \end{cases} \tag{6}$$

where the $\oplus$ sign represents the binary XOR operation.
Different is the case for the nodes that do not have any child and so belonging to the last level of the trellis. For these nodes the B_v vector contains only one value that is computed using the single A_v value received from the parent with the following formula:

$$B_v \triangleq \begin{cases} 0 \quad when\ A_v \geq 0\ or\ i\ \epsilon\ frozen\ position \\ 1 \quad otherwise \end{cases} \tag{7}$$

It is important to check if the node belongs to a frozen position or not. If it belongs to a frozen position (the white nodes in Fig. 2) the B_v returned value will be always zero, independently from the received A_v value.
The B_v values are passed to the parent that will combine them with the values received from the other sibling in order to compute the B_v values to be passed to the higher parent. This iterative algorithm is repeated until the root node is reached. Finally, The B_v vector computed by the root node defines the decoded bits.

2.2.2 Line SC

The message passing algorithm can be implemented in a hardware-friendly version using the Line SC decoding algorithm. The Line SC decoding method is based on the structure shown in Fig. 4 where each node represents a

different type of operation (F-type, G-type or H-type) that has to be executed on the log-likelihood ratios received at the right of the scheme. As in the Message Passing case, the algorithm is based on a forward iteration from the right to the left and a backward iteration from the left to the right.
This new algorithm scheme can be built following the definitions given in [2]. First of all, the different types of nodes have to be defined according to:

$$L_{l,i} \triangleq \begin{cases} f(L_{l+1,i}; L_{l+1,i+2^{n-l-1}}) & if\ \frac{i}{2^l}\ is\ even \\ g(s_{l,z}; L_{l+1,i-2^{n-l-1}}; l_{l+1,i}) & otherwise \end{cases} \tag{8}$$

where "l" represents the state while "i" indicates the row of the trellis shown in Fig. 4. The "s" value used by the G function is the XOR operation between the hard decisions computed starting from the node which forks into the G node.

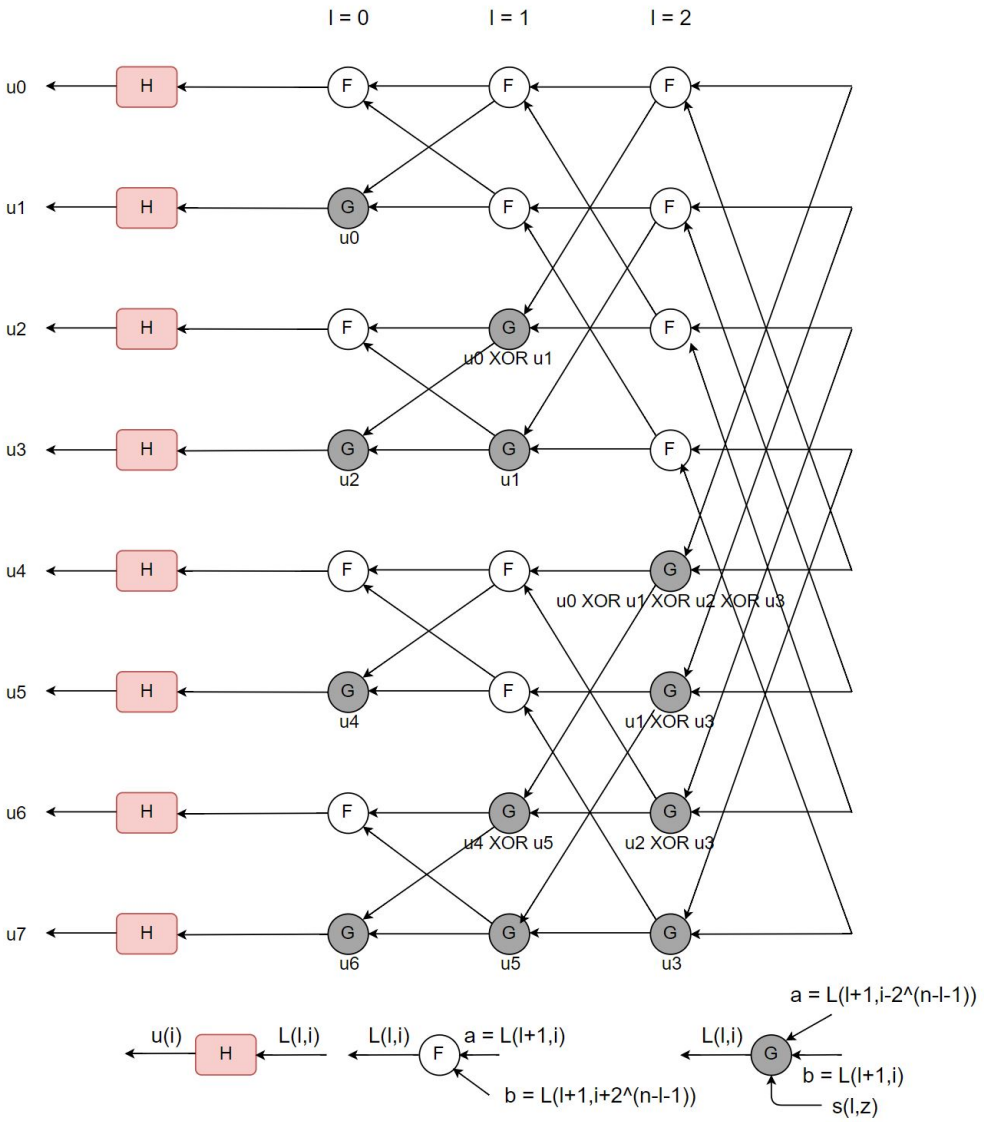

Figure 4: Decoding scheme for the implementation of the Line SC algorithm with N = 8

Regarding the F and G operations, they can be computed with equations 9 and 10 [2]:

$$f(a,b) \triangleq \frac{a \cdot b + 1}{a + b} \tag{9}$$

$$g(s,a,b) \triangleq a^{1-2s} \cdot b \tag{10}$$

However, using the likelihood ratios, the previous formulas can be rearranged in the following ones [2]:

$$f(a,b) \triangleq 2tanh^{-1}\left[tanh\left(\frac{a}{2}\right) \cdot tanh\left(\frac{b}{2}\right)\right] \tag{11}$$

$$g(s,a,b) \triangleq a \cdot (-1)^s + b \tag{12}$$

Unfortunately, this way to compute the F function is very slow for an hardware implementation that needs real time computing. For this reason the log-likelihood ratio have been used in order to exploit their properties to simplify equation 11 with the following one [2]:

$$f(a,b) = sign(a) \cdot sign(b) \cdot min(\mid a \mid, \mid b \mid) \tag{13}$$

where the vertical bars perform a normal magnitude operation while the sign function can be described as follows:

$$sign(x) \triangleq \begin{cases} -1 & if\ x < 0 \\ +1 & otherwise \end{cases} \tag{14}$$

Thanks to the log-likelihood ratios equations 12 and 13 can be performed using simple comparators and adders speeding up the operation frequency.

Finally, the hard decision is evaluated using a certain threshold and checking if it has to be done for a frozen position or not. Equation 15 [2] displays the hard decision function which can be implemented with comparators:

$$u_i \triangleq \begin{cases} 0 & if\ i\ \epsilon\ frozen\ position \\ 1 & if\ L_{0,i} < 1 \\ 0 & otherwise \end{cases} \tag{15}$$

2.3 Simplified Successive Cancellation Decoding Algorithm

The throughput of the SC decoding algorithm can be further improved using the SSC algorithm without affecting the error-correcting performance [3]. This can be archived exploiting the known a-priory positions of the frozen values in order to perform some simplifications over the trellis. Unfortunately, the SSC algorithm simplifications can be applied only to the message passing version. Using the scheme shown in Fig. 5, three different types of nodes can be identify:

- Rate-zero nodes (in white) representing the nodes whose children belong only to frozen positions
- Rate-one nodes (in black) representing the nodes whose children belong only to non-frozen positions
- Rate-R nodes (in grey) representing nodes whose children belong both to frozen positions e non-frozen ones

The rate-zero nodes have only rate-zero nodes as descendants while the rate-one nodes have only rate-one nodes as descendants.
Reference [4] pointed out that the partial sums coming from the rate-zero nodes are always zero: this means that the decoder has not to wait the hard

decisions coming from the button of the trellis in order to compute the vector to be passed to the other child or the vector to be passes to the parent. Otherwise, it can proceeding knowing a-priori that the partial sum will be zero.

Again in [4] was demonstrated that the hard decision for the root rate-one nodes can be carried out directly from the received soft input without obtaining different results then the ones computed with the standard SC algorithm.

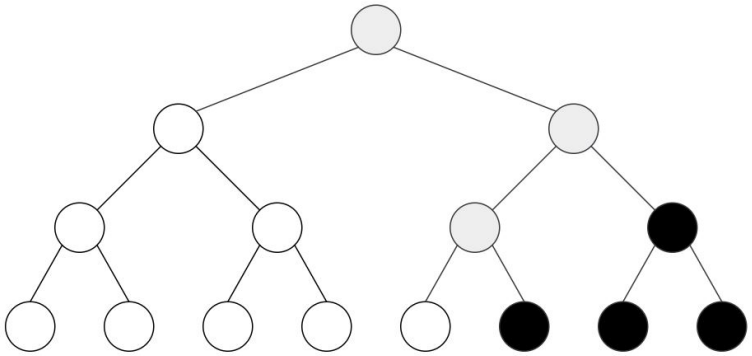

Figure 5: Trellis diagram with N = 8, k = 3 and node types

These simplifications allow to truncate the trellis as it is shown in Figs. 5 and 6. The lower number of leafs can be translated in a significant reduction of the computational steps needed to decode the receiving frames. The lower number of required computations increases the achievable throughput.

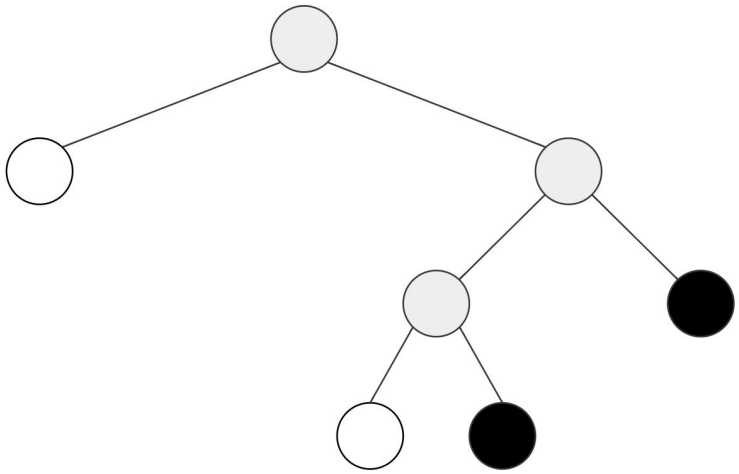

Figure 6: SSC trellis diagram with N = 8 and k = 3

3 State-Of-The-Art Examples

In this section, several decoder architectures will be presented in order to provide reliable terms of confrontation between the work that will be exposed in this thesis and other interesting designs. Unfortunately, it was not possible to find any ASIP implementation dedicated to the Polar Codes decoding. This is the reason why two different types of examples will be detailed in the following lines: three ASIC decoders dedicated to the Polar Codes and three ASIP decoders dedicated to Turbo Codes and LDPC Codes.

3.1 ASIC Decoders for Polar Codes

In [13] is presented one of the first ASIC implementation of a Polar Codes decoder. It is based on a semi-parallel architecture that is subsequently improved with some modifications to the processing elements (PEs). The design of [13] is based on four main units: an array of PEs, a LLR memory, the partial-sum update logic with the associated storage registers and a controller. The allocated PEs are N/2 where N is the code-length of the code that has to be decoded. The LLR memory stores the partial LLRs during the decoding process. It is implemented using registers connected to the PEs through some multiplexers. The partial-sum update logic manages all the different value updates when new nodes of the decoder trellis are computed. Finally, the controller is able to provide the correct control signals for the entire architecture using three different counters: one that handles the current decoded bit index, one that handles the current decoding level of the trellis and the last one which handles the node of the trellis being processed. Then, in the second part of [13] an optimization is introduced in order to improve the throughput of the ASIC. It concerns a modification of the PEs in order to

compute two values at the same time. This is archived computing both the F and G functions of two linked nodes at the same time. As it was explained in section 2.2, the G functions needs the F function output in order to be calculated but if the two possible output values of G are computed together, the right one can be selected with the F output. In this way, the parallel computation is possible with a minimal time and area overhead. The ASIC proposed in [13] occupies a area of 1.72 mm^2 (using a 180-m technology) and it is able to decode a 1024-bit code-length with a throughput of 49 Mbps.
A second ASIC implementation of a Polar codes decoder is proposed in [14] where another semi-parallel architecture is presented. The principal purpose of this semi-parallel architecture is to reduce the complexity of the decoder introducing a small latency increment. The studies detailed in [14] demonstrate that when a code with code-length N has to be decoded, there is the possibility to use simultaneously $\frac{N}{2}$ processing elements only two times for the entire decoding process. This is the reason why a lower number of PEs can be implemented reducing the architecture complexity with a small latency handicap which affects only the stages where all PEs are actually used. This type of approach is called semi-parallel architecture by the authors of [14]. The design proposed by them is based on $\frac{N}{4}$ processing elements that are fed by a random access memory (RAM) where all the temporary LLR values are stored. Since each PE requires two LLRs as input and produces one LLR as output, the RAM is designed to store a vector of LLRs coming from the processing elements and to provide two vectors of LLRs to the same processing elements at each clock cycle. Partial sum registers are also placed with the aim of storing the partial sum values needed by each G node (equation 12) to compute their outputs. This registers are uploaded at each decoding stage by a dedicated partial sum update logic module. Finally, a controller

is added to the ASIC where all the control signals of the architecture are computed at each decoding stage. With a code-length of 1024, the decoder presented in [14] reaches a throughout of 123 Mbps with an implementation area of 0.31 mm^2 using a 65-nm technology.

The last ASIC is reported in [15]. It is a flexible Polar Code decoder that supports any code-rate and any frozen-bit positions. The peculiarity of this implementation is the possibility of choosing between three different decoding algorithms. The first one is the Successive-Cancellation algorithm (SC) explained in section 2.2. The second one is the Successive-Cancellation Flip (SCF) algorithm [17] which at the start proceeds exactly as the SC algorithm but during the final computations some simplifications are done in order to decrease the overall number of operations. When a candidate codeword is elaborated, it is checked with a cyclic redundancy check (CRC). If the outcome of the CRC is matched, the codeword is accepted, otherwise, the decoding process is restarted until a match is found. The last algorithm is the Successive-Cancellation List algorithm (SCL) [18]. It starts with a list of L codeword best candidates elaborated by itself. Then, it computes a path of reliability metric during the decoding with the aim of discarding the worst codewords of the list. At the end of the computations, the remaining codeword is picked as the decoded one. The ASIC is capable of using these three different algorithms exploiting four main units: a flexible decoder, an unrolled decoder, a clock generation unit and a test-controller unit. These units work with two different clock domains: a slower one (20 MHz) used by the clock-generator unit and the test-controller unit and a faster one used by the two decoders. Regarding the two decoders, the flexible one is used when one of the three different algorithms has to be adopted with a custom code-length and code rate. Whereas, the unrolled decoder is able to run only

the SSC algorithm with an higher speed and energy efficiency than the other decoder but giving up flexibility and the error-correction performance offered by the other algorithms. This particular decoder is called unrolled because of its pipelined architecture where the different decoding steps are computed in sequence. Regarding the clock-generator unit, it produces a fast clock using the 20 Mhz one as reference. The faster clock is produced by a frequency lock loop (FLL) unit that can be configured during the run-time. Finally, the test-controller unit is mostly composed by registers where the partial LLRs are stored during the run-time. It also includes a finite state machine which selects the desired decoder and configures both decoders and FLL. The ASIC presented by [15] produces a throughput of 306.8 Mbps exploiting the flexible decoder with the SCL algorithm which is the fastest one. On the other hand, choosing the SC algorithm, the unrolled decoder produces a throughput of 212.6 Mbps while the flexible decoder was able to reach a throughput of 187.6 Mhz. The two types of decoders where tested with a 1024-bit codeword and they were synthesized separately with a 28-nm technology. The flexible decoder occupies $0.44mm^2$ of area while the unrolled one occupies $0.3mm^2$.

3.2 ASIP Decoders

The ASIP decoder reported in [18] provides a valid standard of comparison for the ASIP developed in this thesis. The architecture detailed in [18] is fully dedicated to the Turbo Codes decoding. It is composed by two different sub-ASIP decoders connected together through buffers and multiplexers. The two sub-ASIPs work in parallel in order to decode the incoming codeword. One sub-ASIP processes the data in the natural order while the other one processes the data in the interleaved order. The complete block calculation

is handled dividing the same block into N windows that are then computed by the sub-ASIPs one by one. Each sub-ASIP has eight pipeline stages and it computes the forward recursion and the backward recursion simultaneously. The ASIP presented in [18] provides a throughput of 130 Mbps and occupies an area of $2.1mm^2$ with 90-nm technology.

The authors of [19] also propose a multi-ASIP decoder for low density parity check (LDPC) and Turbo decoding but with more sub-ASIP units. In details, eight sub-ASIPs are integrated and connected together using a Network on Chip (NoC). A ring network is also implemented in order to allow the metric exchanges between the different sub-ASIPs. A program and a configuration memory are implemented inside the ASIP. The configuration memory contains the data needed to configure all the communication parameters inside some dedicated registers implemented in each sub-ASIP unit while the program memory contains the instructions needed by each decoding step. This architecture is able to reach a maximum throughput of 312 Mbps occupying an area of 2.6 mm^2 with 90-nm technology.

Finally, the ASIP presented in [21] provides a recent example of LDPC decoder compatible with the 802.11ad standard and with an high grade of parallelism. It offers a raw-based architecture using layered scheduling with the aim of increasing both area and energy efficiency. The ASIP is capable of elaborating four different code rates and it processes a code length of 672. The authors of [21] employed 762 variable node function units (VFUs) and 42 check node function units (CFUs) in order to increase the throughput. All these units are divided into sixteen slices and each slice contains data memories, a barrel shifter and several arithmetic computation elements. The slices were designed in order to work in parallel. The ASIP provides a throughput of 7.07 Gbps and an integration area of $0.126mm^2$ using a 28-nm technology.

4 Software Overview

The design of the different ASIP versions presented in this thesis was entirely realized using ASIP Designer: a tool suite provided by Synopsys. This particular software was a easy choice because it allows a rapid exploration of architectural options offering an efficient C/C++ compiler which can generate power and area-optimized synthesizable RTL.
ASIP Designer models the different architectures starting from a nML processor description language where the instruction-set and the memory configuration of the prototype have to be indicated. The best feature offered by this description language is the possibility to model each design in a cycle- and bit-accurate way.
The nML description is compiled by the compiler provided by the tool suite. It can cope a wide range of architectural peculiarities of DSP cores as instruction-level and data-level parallelism, pipelined instructions, specialized arithmetic functions, custom data-types, specialized address generation units, heterogeneous register structures and various degree of instruction encoding. The coded instruction-set can be easily debugged using a custom assembler file or generating this one from an external C code.
The next sections will further focus the main parts of the software. All the reported information are provided by [7].

4.1 nML Description Language

The nML is an architecture description language [12] used by ASIP Designer to model the processor in a concise way. All the different instructions that compose the processor instruction-set are described hierarchically using the so called OR and AND rules:

- The OR rule is usually exploited in order to group a certain number of instructions that can not be executed together but only one at the time. This could be the case of a certain number of instructions that uses the same arithmetic unit for example. At each cycle, only one instruction of this group can be executed.

- The AND rule have the opposite purpose of the OR rule. It groups instructions that can be executed in parallel when, for example, more than one arithmetic units are available for the computations.

The nML also provides the tools able to describe the different units that compose the architecture, as memories, registers or arithmetic units, and all the connections between them.

4.2 C/C++ Compiler

The compiler is a very important tool provided by ASIP Designer. It is able to translate a C or C++ code into an assembly code which is runnable by the designed architecture. This important feature is exploited to test and verify the developed ASIP with all the operations for which the architecture is implemented. As consequence, the writing of the assembly code becomes optional. A C code is enough in order to make the software generate all the instructions that have to be tested. The compiler also manages all the memory initializations and operations in order to correctly store all the variable used into the C/C++ code. The link between the code operators and functions with the ASIP operations is done listing all the connections into the so called "chess primitive header" [14]. The compiler reads this file every time a compilation has to be done. An example is provided in Listing 1 where the connection between the "+" operator of the C/C++ code and the sum

operation of the ASIP is reported. The "int" indicates the type of variable for which the operation is possible while the "word" indicates the type of data in which the variable has to be translate inside the ASIP architecture.

Listing 1: Chess primative header declaration for sum operator

```
promotion int operator+(int, int) = word sum(word, word)
```

4.3 Instruction-set Simulator

The instruction-set simulator [11] is a fundamental tool for the architecture debugging. It allows to simulate the C/C++ code cycle by cycle with the aim of monitoring all the instructions executed by the ASIP and finding the possible problems. The simulator also provides a microcode where the executed assembly code can be displayed. By clicking on one line of the microcode it is possible to see the corresponding lines of the C/C++ code and vice versa. The simulator also provides the possibility to watch the memories and register content at each cycle. Furthermore, the simulator implements some profiling tools able to generate very interesting information after the simulations. For example, it is possible to view the usage of the different units integrated into the ASIP or the percentage of the different instructions run during the simulation. These tools can be very useful when a bottleneck has to be found or the instructions have to be improved. Finally, files containing the data stored into well defined memory ranges can be printed so that the results of a given simulation can be compared with the right ones.

4.4 RTL Generator

The RTL generator [12] is a fundamental tool of ASIP Designer. It is able to create a RTL version of the implemented architecture in both VHDL and Verilog description languages. When the RTL generator is executed, a configuration file has to be indicated. Inside the configuration file all the features provided by the tool can be activated. Some of these features are listed below:

- Generate an HDL testbench for the architecture
- Create synthesis script for architecture implementation
- Use existing hardware blocks to integrate the RTL Design
- Enable low-power design optimizations such as selective clock gating

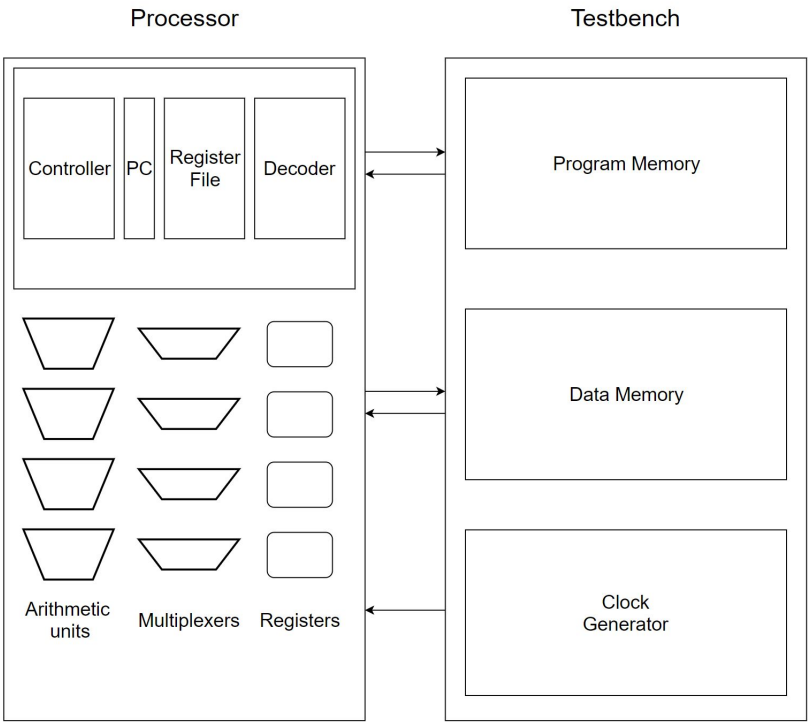

Figure 7: Generated RTL structure

Fig. 7 reports a scheme where the generated RTL code is defined. Only the processor logic is actually implemented with real gates while the memories and the clock generation unit are only coded inside the testbech. The testbech elements can be used only for test purposes but they will not be included for the synthesis.

5 Code Overview

The C code used in order to implement the decoding algorithm as machine language is very important because it directly affects the final performance of the ASIP. In this section, only the SSC decoding algorithm code will be presented because it is the code used for the simulations of all the different ASIP architectures. Three versions of the recursive function exploited to implement the different decoding nodes will be detailed:

- First version of the decoding function
- Second version of the decoding function
- Vector version of the decoding function

5.1 First version of the decoding function

The first version of the decoding function was implemented following the message passing decoding scheme. Each time the decoding function is evoked, five different parameters have to be indicated:

- Current state: it indicates the level of the tree where the decoding node is located. This is needed to fix the vector length of the bits that have to be received and computed. It is also used to compute the position in the Tree vector where all the node types are listed in order to use the SSC tree simplifications, as it was explained in section 2.3
- A_v vector: it contains the bits elaborated by the parent node
- B_v vector: in contains the bits elaborated by the current node after it receives the hard decisions vector from its descendants

- Tree vector: it is the vector which contains the node type of each node. It is already saved into the ASIP memory at the start-up of the system and it is only read to check the SSC simplifications

- Position: it is an integer value indicating the horizontal position of the current node. It is read to compute the node position inside the Tree vector

The first operation done by the decoding function is to check the Tree vector so that the type of computations to be performed are selected.
If the decoding node is a Rate-R node, it starts to compute the A_l bits to be passed to its left child. Then, it evokes another decoding function where the A_l bits are indicated, the current state is decremented by one and the new position is computed using a dedicated function. This new decoding function returns the B_l bits that are used to compute the A_r vector. Again, the parameter to be passed to a new decoding function representing the right child node are computed and the B_r vector is received. The B_l and B_r vectors are exploited in order to compute the B_v vector which is finally returned to the function that evoked the decoding function.
Otherwise, if the decoding node is not Rate-R type it can directly compute the B_v vector from the received A_v vector in two different way, depending on its type:

- Making the hard decision operation for all the received bits if the node is R-1 type

- Returning a vector with all frozen value if the node is R-0 type

5.2 Second version of the decoding function

The second version of the decoding function implements a lot of modifications in order to improve the arithmetic units usage of the ASIP and decrease the number of instructions needed to translate the decoding algorithm into machine instructions. This particular version of the decoding function will be used starting from the Design 2 implementation, reported in section 6.2. The first important change regards the way in which three computational functions are implemented: minimum, XOR and hard decision. These three functions were initial computed using nested ifs in order to compute the output value. These if statements limit the parallelism of the ASIP and introduce a lot of nop instructions that increase the cycles needed for the decoding, as it is explained in section 6.1. These problems were easily solved by creating some dedicated operations inside the arithmetic unit of the ASIP in order to compute the needed elements in a single cycle, as it is further explained in section 6.2. The C code was updated in order to integrate these new functions by only adding the computational function syntax to the chess primitive header so that a link between the new operations implemented into the architecture and the corresponding C functions is created by the compiler.

In addition, this second version also introduced another big improvement which decreased the number of operations needed for the decoding. The functions used to compute the node position inside the Tree vector and inside the decoding trellis are substituted by two simple counters that are incremented each time a tree position or a trellis position is requested. The decoding algorithm always computes the different nodes of the tree with the same order so if the values computed by both the Tree position and the trellis position functions are stored in two ordered vectors it is possible to read the

desired value by only using the counter as index. When one of these vectors is read, the counter is incremented in order to prepare the index for the next reading.
The last modification concerns the temporary value storing. In this second version of the decoding function all the trellis values computed by two or more operations were divided and all the single results needed to complete a single value computation were stored in dedicated variable. Thanks to this division, register renaming was applied to this version of the decoding function in order to better exploit the architecture parallelism from Design 2 on wards. The different variables where the temporary results are stored in the C code are translated as register file locations from ASIP Designer, avoiding data dependencies between the values computed by the different arithmetic units and decresing the number of memory accesses needed to complete the computations of the trellis values .
All the mentioned modifications will improve the performance of the ASIP as it is detailed in section 6.

5.3 Vector version of the decoding function

The vector version of the decoding function is implemented in order to exploit the vector unit introduced in section 6.5 for Design 5.
The parallel version only substitutes the previous function when the node for which the decoding function is evoked computes vectors with a length equal or multiple of the vector length processed by the vector unit. The choice between the evocation of a parallel function or a normal one is done through an if statement before the function declaration. If the trellis level of the child node is high enough to process vector of bits with the requested length the parallel decoding function will be evoked. Otherwise, the normal one will be

chosen.

The parameters of the parallel decoding function are the same of the previous one. The pointers to the A_v and B_v vectors are copied in a vector type pointer so that the single bits that compose the two vectors are seen as vector operators from the ASIP Designer compiler. The vector unit processes the vectors in the same way the single values are processed by the normal arithmetic units. The only difference is that the vector units processes all the values contained into the vector operators in one single cycle.

Furthermore, all the memory locations dedicated to the storing of both received LLR values and decoded bits have to be correctly aligned in memory. This is needed because the vector unit can access to the vector mode of the Data memory only by using addresses multiple of the vector size, as it is detailed in section 6.5. This problem is easily managed by the "chess_storage" command, an ASIP Designer command which indicates to the compiler when the data has to be properly aligned for vector access.

6 Hardware Implementation

6.1 Design Overview (Design 1)

In this section the base implementation of the ASIP will be presented. The structure shown in Fig. 8 is the starting architecture which will be improved in the following sections. The base ASIP is composed by a 2^{16} X 16 bit Data memory, 2^{10} X 20 bit Program memory, a single 16-bit ALU and a 3-bit addressed register file. The length of the instruction word processed by Design 1 is 20 bits.

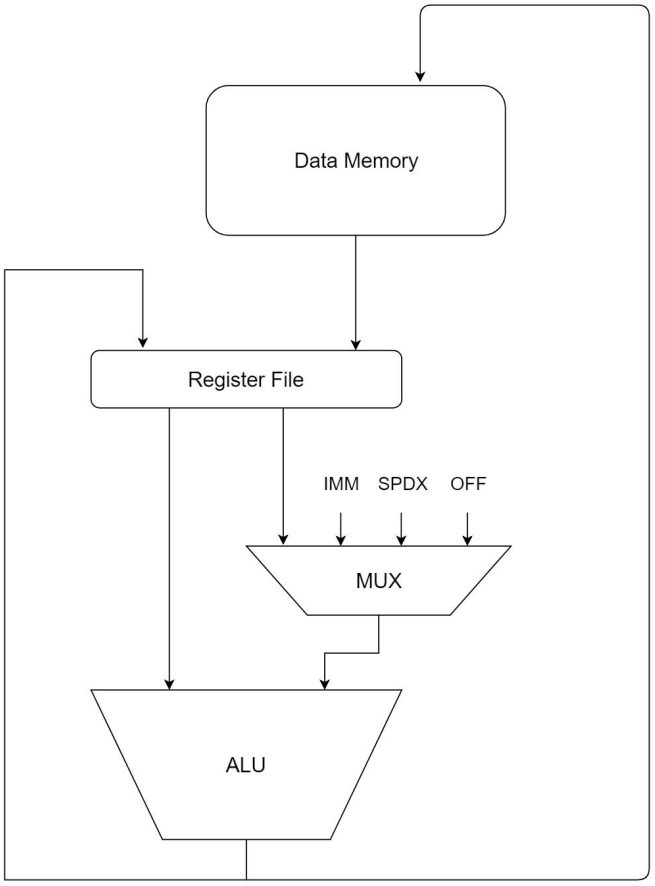

Figure 8: Basic ASIP implementation

The processor works with a 16-bit parallelism for both data and addresses. Theoretically, the models of the symbols transmission through a channel affected by noise [5] show that with a realistic SNR, the number of bits used for the LLRs representation is not crucial for the BER performance. However, it is important to set a proper saturation range in order to minimize the BER, as explained in [5]. Since the BER performance is not one of the topics of this thesis, a saturation for the elaborated data will be not applied.
Regarding the data memory, it is not only used to store the received frames and the decoded bits but it is also exploited to implement a stack. The stack implementation is required because the software, during the assembly code compilation, has to store the temporary values used inside the iterative functions implemented within the C code. Each time a function is evoked, a context switch has to be performed. All the parameters needed by the new function are stored in a portion of the stack appositely assigned. When the function is terminated, the stack portion will be freed and made available for other uses. This is the reason why a 16 bit addressed memory was requisite.
In a first attempt, a processor with two different computational units was implemented: a 8 bit arithmetic unit exploited for the decoding computations and a 16 bit address unit used only for the address computations. This choice was driven by the desire to reduce the area of the ASIP for both memory side and logic side (especially when the instruction level parallelism will be introduced) but it was discarded because of all the bypasses needed between the address and the arithmetic units and the slowdowns introduced by the manipulation of the 16 bit addresses in order to be loaded/stored in a 8-bit memory. These slowdowns occur very often since each time a value has to be loaded from the memory or stored into the memory an address computation has to be executed. In addition, the implementation of several units able to

compute both data and addresses increases the system flexibility and so the performances when the ILP will be introduced. These assumptions are also verified in section 6.2. At last, a 16 bit parallelism was adopted for each part of the design in order to avoid all the previous mentioned problems.
All the ASIP versions will be implemented with the following four pipe stages:

1. IF: instruction fetch, during this stage the address of the next instruction is sampled by the program counter and used with the aim of extracting the next instruction code-word from the Program memory

2. ID: instruction decode, in this stage the instruction code-word extracted from the Program memory is read from the control unit and the Register File in order to respectively set the control signals for the following stages and access to the indicated data

3. EX1: execution 1, during this stage the operation indicated by the instruction is executed by the arithmetic unit

4. EX2: execution 2, this is the final stage where the data is stored into the register file or the Data memory loads/stores the indicated data

The ALU is able to read the operands from the two ports of the register file or, according on the current instruction, a MUX can extract the needed bits from the instruction code. Three types of values can be delivered to the ALU:

- IMM: immediate value used to indicate an operand directly with the instruction code

- SPDX: stack offset value exploited to compute the stack pointer of the needed memory location

- OFF: memory address offset which is used to set the correct address for the data load/store operations

Once the architecture is set, the next step requests to design an instruction-set where all the possible operations done by the ASIP are mapped. This step is particularly important because from the instruction-set depends the efficiency with which the C code, containing all the different steps needed for the decoding process, will be implemented in the machine language. Fig. 9 shows the first instruction-set implementation.
In the following lines the instructions syntax is explained:

- ALU_OPN: instruction that regulates the addition, subtraction, AND, OR and XOR operations performed by the ALU. The A and B fields are used to indicate the register file addresses of the two operands while the D field contains the register file destination address where the result will be stored. The type of operation is selected by the OP code.
- COMPARE_OPN: instruction that regulates the compare operations performed by the ALU. The OP filed selects one compare operation between the following ones:
  - Less than
  - Less or equal than
  - Grater than
  - Grater or equal than
  - Equal
  - Different

The result is stored in a dedicated register called SREG. This register is read when the direction of a conditional jump has to be selected.

- REG_MOVE: instruction able to move a value from a cell of the register file (SRC) to another one (D) using the input and output ports of the arithmetic unit.
- U_JUMP: instruction that regulates normal jumps. The address contained into the program counter is added to the value contained into the OFFSET field in order to compute the jump address.
- C_JUMP: instruction that regulates conditional jumps. The jump occurs only if the value stored into the SREG is matched. The jump address is computed as in the U_JUMP case.
- BSR: call instruction that saves the return address in a register file cell called LR and stores the target address inside another register file cell indicated by the A field. In the next cycle, the target address is fetched by the program counter.
- RTS: return from subroutine instruction. It sets the address previously saved into the LR register as the next address to be fetched from the program counter.
- NOPE: nop instruction. The hardware is stalled when this particular instruction is executed. The nop instructions are issued when the direction of a jump has to be computed, for example.
- IMM_OPN: instruction which regulates the immediate additions and subtractions. The B value is incremented/decremented by the value contained into the VALUE field. The type of operation is selected by the OP field.

- SIGN_OPN: instruction which regulates the ABS and SIGN operations needed for the decoding algorithm. Only one operand is indicated by the B field.
- COMPLEX_OPN: instruction that regulates the multiplication, division and modulo operations performed by the ALU.
- LOAD: instruction that regulates the data load from the memory to the register file. The data memory is addressed by the value contained in the ADD location of the register file. The extracted value is stored in the D location of the register file.
- STORE: instruction that regulates the data store from the register file to the data memory. The value contained in the SRC location of the register file is stored in the data memory location indicated by the address contained in the ADD location.
- LOAD_SP: instruction that sums the value contained in the OFFSET field to the current stack pointer. The result is used as address to load a value from the memory. This instruction and the following one are used to interact with the values contained in the stack.
- STORE_SP: same as LOAD_SP but the result is used to store the value in the data memory.
- SH_OPN: instruction which regulates the shift operations performed by the ALU. The A field contains the address of the operand while the B field contains the address of the number of shifts to be performed.
- RESET_REG: writes a zero into the register file location addressed by D

- GEN_WORD_UP/DOWN: these two instructions generate a value inside the register file writing in two different moments the first 8 bits and the second ones of the word.

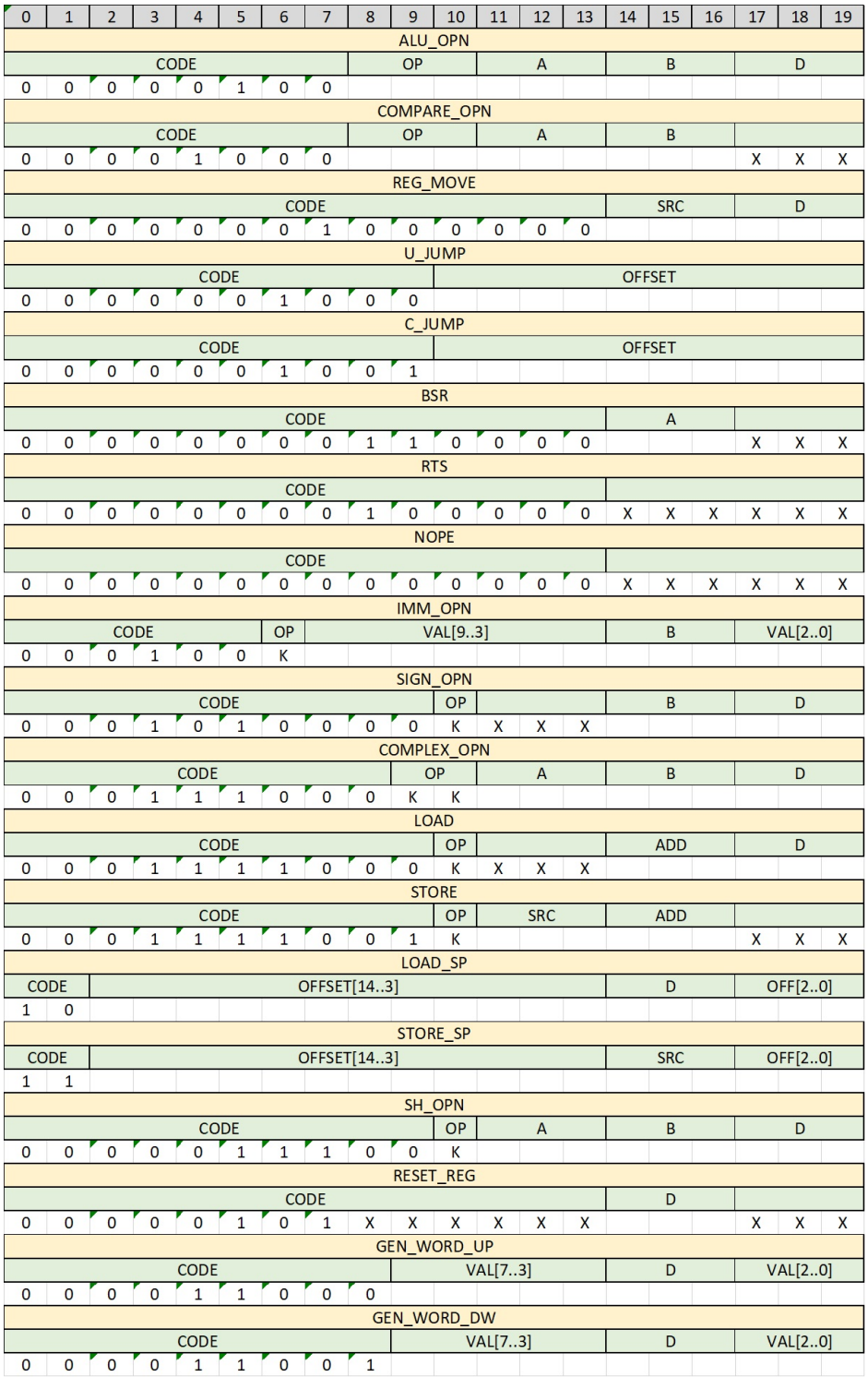

Figure 9: First instruction-set implementation

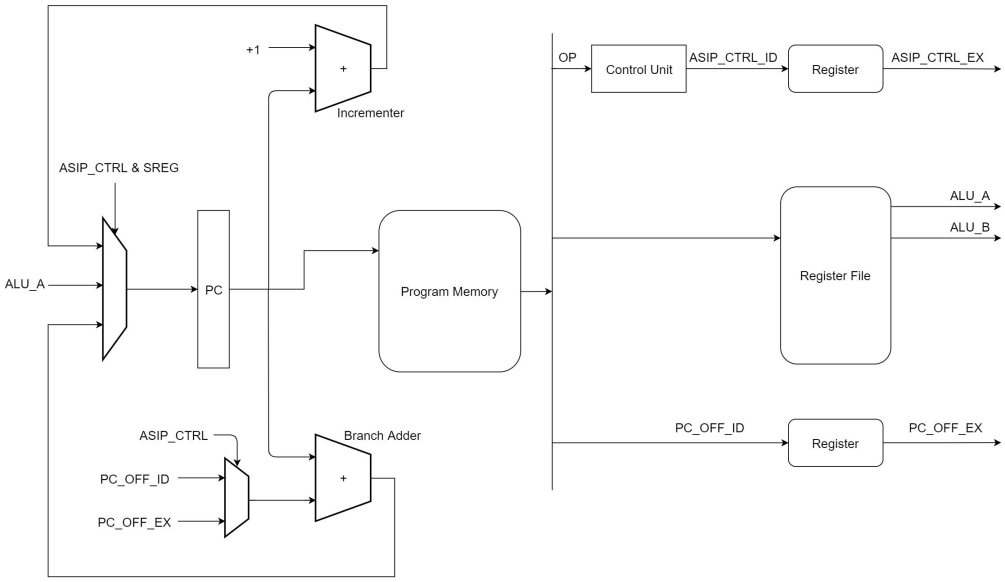

Figure 10: ASIP instruction fetch focus

Regarding the instruction fetch, a focus on the hardware dedicated to this job is shown in Fig. 10. The address fetched by the program counter can be selected between three sources:

- Incrementer: it increments the current fetched address by one in order to continue the normal instruction sequence
- ALU_A: it vehicles the address stored inside the register file when a BSR or RTS instruction is executed
- Branch Adder: it selects the address computed with the current fetched address and an offset that can be provided by both jump and conditional jump instructions

The selection is chosen by the ASIP_CTRL signals that are the control unit signals at both ID and EX stages merged together and the SREG output which is important when a conditional jump occurs. Both the control signal

in the two stages are needed because a normal jump is executed when it is located in the decode stage while a conditional jump is executed only when the instruction reaches the execution stage where the branch direction is computed and stored inside the SREG register. In the same way, the Branch Adder has to be able to select between two different offsets to be added to the current fetched address. The offset located in the decode stage has to be chosen when a normal jump is executed, otherwise, the offset in the execute stage has to be selected when a conditional jump is executed. This selection is again managed by the ASIP_CTRL signals. Every time a jump of both types has to executed, the compiler also introduces a sequence of nop instructions that allows the computation of the branch address without fetching wrong instructions in the meantime. The part of the architecture dedicated to the address fetch is never changed in the following designs exception made for Design 6 where the implementation of the hardware loop requires a little modification of the fetch circuit.

6.1.1 Algorithms comparison

Two simulations are run using the same architecture but compiling two different algorithms: the SC message passing and the SSC message passing algorithms. This test is carried out in order to appreciate the different performances between the two decoding algorithms. The code to be decoded has a code-length equal to 1024 and a code rate of 0.5. This means that the algorithms will extract 512 information bits from the received frame. The simulation results are shown in Table 1

Algorithm	SC	SSC
Cycles	1332480	749345
ALU usage	59.9%	59.75%
DM usage	36.96%	36.77%
PM size	417	498

Table 1: SC vs SSC algorithm simulation results

The SSC algorithm is 43.79% faster than the SC one. The simplifications made to the tree bring a big speed-up in terms of performance. The only handicap is given by a slightly increment of the program memory size. This is due to the additional if statements present in the SSC code. They are needed to discriminate the different types of nodes previously mentioned in section 2.3

The assembly code of both the SC and SSC simulations showed a big number of nop instructions generated by the conditional jumps with whom the if statements are translated. This means that the performance could be further improved decreasing the number of if statements in the code.

Three operation types are implemented with nested if statements that can be easily substituted by three specific ALU operations. The operations under discussion are:

- minimum operation, used in equation 14
- word XOR, used in equation 6
- hard decision, used in equation 7

These modifications will be applied to the next design.

6.1.2 Synthesis

In this section the Synopsys tool suite is exploited in order to synthesize the Design 1 architecture. The synthesis gives a feedback regarding the predicted

needed area for the silicon implementation and the critical path delay. This last information is very important because it is one of the variables that will determine the working frequency of the ASIP.
The technology used for the synthesis is called UMC 65. It is a 65-nm technology with some interesting features like multiple VT options, for power consuption and performance balancing, and retrograde twin well for parasitics and leakage current reduction. For more details refer to [6].
The results of the Design 1 synthesis are shown in Table 2. These results were obtained with the simplest synthesis available without using any optimization offered by the Synopsys suite. For this reason, the synthesis data will be used only to compare the different Designs proposed in this thesis not in absolute terms but in relative ones.
This logic synthesis and all the following ones are carried out without including the Data and the Program memories. Memories have to be generated by different tools and their sizes are usually not planned by the designer but they are strictly connected to the current market availability.

Design 1	
Combinational Area (μm^2)	15913
Noncombinational Area (μm^2)	1976
Total Area (μm^2)	17886
Critical path (ns)	4.8
Throughput (Mbps)	0.142

Table 2: Design 1 synthesis results

The combinational area value expresses the total area dedicated to the combinational logic while the noncombinational area represents all the area dedicated to the logic which has nothing to do with the combinational units as, for example, registers or multiplexers. All the area values do not take into account the area needed for the routing process. The obtained throughput

with this first version of the ASIP is 0.142 Mbps, very low compared to the examples presented in section 3. The architecture needs to be improved in order to obtain results comparable with the state-of-the-art examples.

6.2 Design 2

The Design 2 is a modified version of Design 1 where all the operations that previously required the usage of if statements are substituted by dedicated operation performed directly by the arithmetic unit. The syntax of the new implemented instructions is shown in Fig. 11.

- The IF_OPN instruction selects through the OP field one operation between the minimum operation and the word XOR
- The HARD_DEC instruction performs the hard decision on the B operator

These additional instructions requested an extra bit for the code fields bringing the instruction word length up to 21 bits.

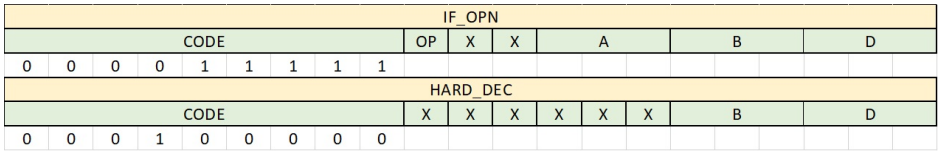

Figure 11: Design 2 additional instructions

The simulation performed in section 6.1.1 is repeated for Design 2 using only the SSC algorithm. The results are reported in Table 3:

Design	1	2
Cycles	749345	280994
ALU usage	59.75%	68.53%
DM usage	36.77%	36.17%
PM size	498	235

Table 3: SSC simulation results for Design 1 and Design 2

The Design 2 improved its performance by 62.5% over the Design 1. Also the ALU usage increased because of all the conditional jumps introduced by

the if statements were substituted by dedicated computations performed by the arithmetic unit. On the same basis, the Program memory size decreased by 52.81%.

These results are perfectly aligned with the expected ones. Each time an if statement is used, the ASIP processor stalls the instruction fetch for two cycles in order to wait the direction computation without executing the next instructions. Thanks to the dedicated arithmetic operations presented in Fig. 11 this big handicap is avoided with a little increment of the arithmetic unit complexity.

Secondly, the decrement of the if statements brings also a decrement for the conditional jumps. This will surely be a good point when the arithmetic units will be duplicated since the conditional jumps have to be computed by only one arithmetic unit while the other ones has to wait until the correct address is fetched. The conditional jump decrement will increase the percentage of instructions that can be run in parallel when the instruction level parallelism will be introduced.

6.2.1 Synthesis

The Design 2 synthesis results are displayed in Table 4. As expected, the Design 2 computational area increased respect to Design 1. This is due to the additional logic needed inside the arithmetic unit in order to substitute the if statements with the dedicated instructions shown in Fig. 11. Whereas, the noncomputational area do not increase because of all the logic outside the arithmetic unit was not modified.

Design	1	2
Computational Area (μm^2)	15913	17226
Noncomputational Area (μm^2)	1976	1970
Total Area (μm^2)	17886	19195
Critical Path (ns)	4.8	4.8
Throughput (Mbps)	0.142	0.38

Table 4: Design 1 vs Design 2 synthesis

In conclusion, the Design 2 modifications increased the total area of the ASIP by 7.3% compared to the Design 1 implementation. This is an acceptable handicap taking into account the speed-up which the Design 2 brought. Indeed, the throughput of Design 2 increased proportionally with the decrease of the cycles needed to complete a frame decoding.

6.3 Design 3

The Design 3 implementation exploits instruction level parallelism in order to speed-up the operations. Two computational units are implemented, each one with its own register file, as it is shown in Fig 12. The first computation unit is dedicated to the arithmetic computations while the second one computes the memory addresses and the stack pointer modifications, as it is shown by the instruction-set displayed in Fig. 13.

The instruction level parallelism introduced a big handicap for the Program memory word size: the instruction bit-length grew to 36 bits. Thus, it is not practical to implement a single stack memory with such a word length but it is more practical to implement several memory banks with a smaller word length and then access them with the same address in parallel. This concept remains true for the next ASIP designs where the instruction length will further grow.

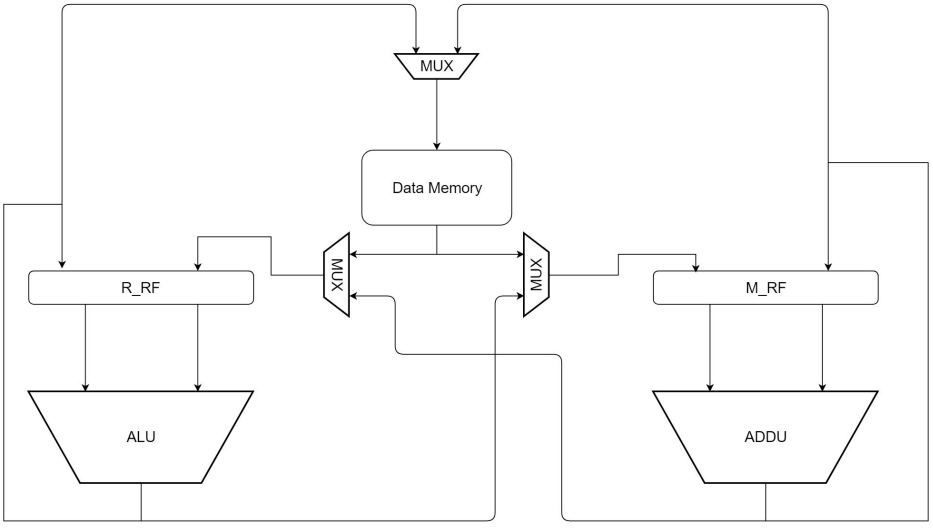

Figure 12: Design 3 datapath

The two register files are implemented with two bypasses able to link

them and allow register moves between the two different arithmetic units. These bypasses have been inserted in order to increase the flexibility of the system and allowing the address computation with the ALU when the ADDU is busy. The bypasses allow to directly load values without using the Data memory. This saves one cycle each time a move between the two register files is required and should increase the performance of the device because the instruction level parallelism is more efficiently exploited. The bypasses are connected to a multiplexer which selects if the data has to be loaded from the other register file or from the Data memory. It is important to underline that the hardware of the two arithmetic units is not the same but the ADDU is a simpler version of the ALU where only sums and subtractions can be performed. Regarding the SREG, it was not doubled because only the ALU includes the hardware needed to compute the jump conditions. In any case, as it was stated in the previous sections, only one arithmetic unit at the time can work when the direction of a jump has to be computed.

The instruction-set has been adapted to the datapath parallelization. Fig. 13 displays the two different modes in which the Design 3 ASIP can work: the parallel mode is able to execute two instructions in parallel with the aim of exploiting the two arithmetic units while the immediate mode is used to execute the instructions containing an immediate value or the instructions that modify the flow of the program counter (like jumps or rts). The two different modes have been discriminated by the mask bits at the start of the instruction code (displayed in dark yellow by Fig. 12). The immediate mode is able to execute only a single instruction at the time. Unfortunately, the immediate mode and the parallel mode were not merged because of the different lengths of their instructions. Indeed, if a parallel mode was introduced for the immediate instruction too, the instruction word length would

be further increased.
In addition, the LOAD/STORE_SP_SMALL instructions have been implemented into the parallel mode: when the stack pointer has to be modified by a little value it can be done by the parallel mode exploiting the SP_SMALL instructions instead of the LOAD/STORE_SP instructions of the immediate mode. This represents a big speed-up for the overall performances since the largest part of instructions that modify the stack pointer need to move the pointer by little ranges that can be managed by the SMALL_SP instructions. If the SMALL_SP instructions were not introduced, all the stack pointer manipulations would be implemented with the LOAD/STORE_SP instructions of the immediate mode losing the opportunity to use both the arithmetic units at the same time. The LOAD/STORE_SP instructions are only used when the ASIP switches from one function of the C code to another one while all the stack modifications inside the same C function are managed by the SMALL_SP instructions.
Another speed-up is introduced by the merger of the GEN_WORD_UP/DW instructions. In Design 1 and 2, a word was generated using these two instructions in order to respectively manipulate the upper and lower bits of a register file location using an 8-bit immediate. However, Design 3 implementation offers the possibility to use 16-bit immediate value making the word generation in one single cycles possible. Unfortunately, the GEN_WORD instruction of Design 3 can not be executed in parallel mode.
Regarding the REG_MOVE instructions shown in Fig. 13, they are able to move the word contained into one register file to the other one exploiting the bypasses shown in Fig. 12

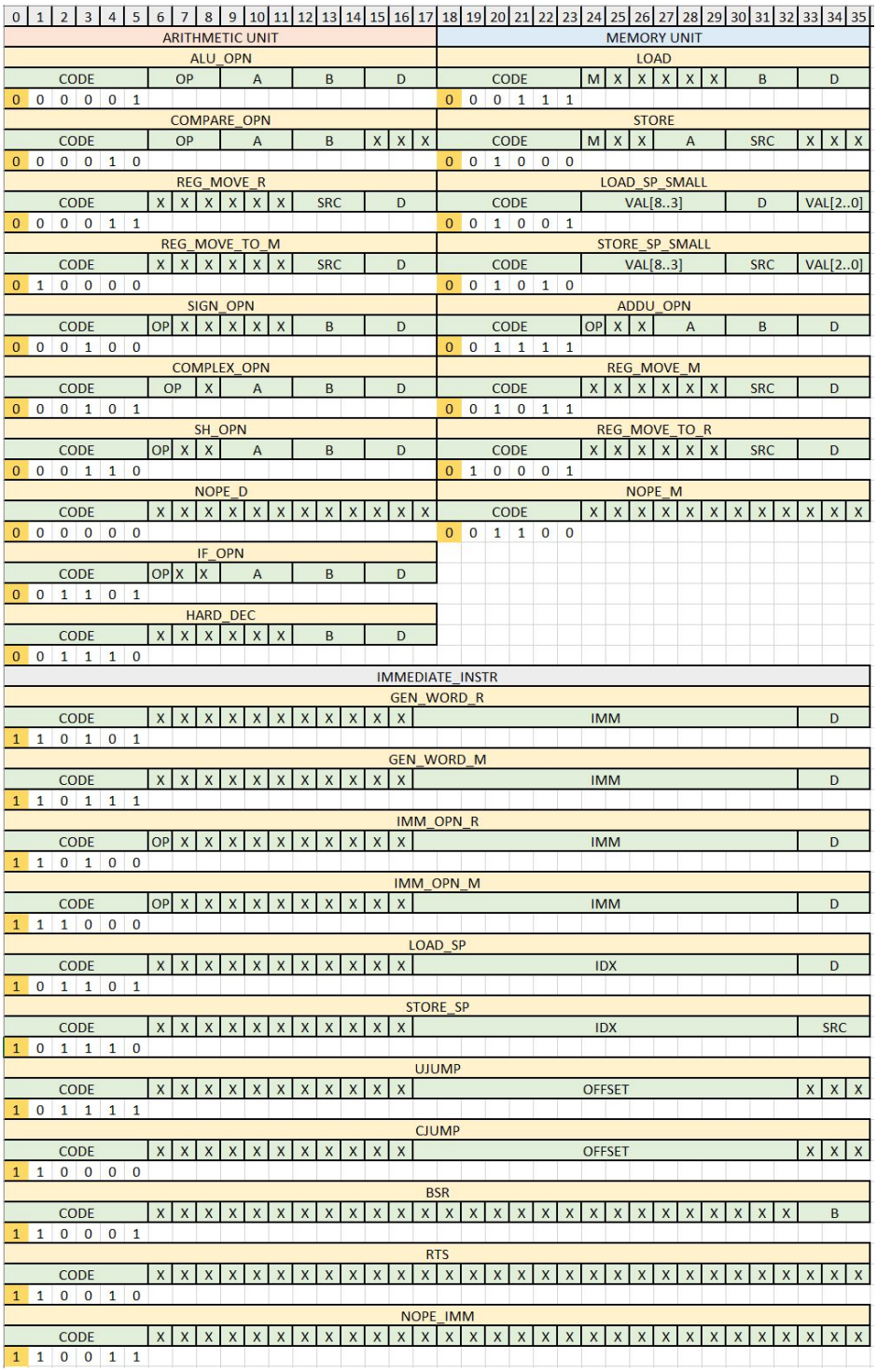

Figure 13: Design 3 instruction-set

The Design 3 simulation results are shown in Table 5 where as ADDU is indicated the arithmetic unit which performs the address computations. Unfortunately, the obtained results pointed out a problem: the number of cycles needed from Design 3 to complete the decoding process decreased by only 20.96% compared to Design 2. This is an unexpected result since the predicted speed-up computed with the Amdahl's law was higher than 50%. This problem can be solved applying the following points:

1. If the two arithmetic units are the same, the system flexibility increases and the software can allocate the instructions in a more efficient way.

2. The division of the register file brings a big handicap because of the additional move instructions needed in order to transfer partial values from one arithmetic unit to the other one. A performance increment should be appreciated designing a modified version of Design 3 where the two units are connected to a single register file.

3. The number of conditional jump instructions used to control the loop statements is too high compared to the instructions where the two units are actually exploited. This can be solved using the loop unfolding technique.

Design	2	3
Cycles	280994	222104
ALU0 usage	68.53%	34.66%
ADDU usage	/	26.11%
DM usage	36.17%	21.84%
PM size	235	207

Table 5: Design 2 vs Design 3 simulation results

6.3.1 Modified Design 3

The Design 3 architecture has been modified in this new implementation with the aim of solving all the problems enumerated in the previous section. The modified version of Design 3 is displayed in Fig. 14. The instruction bit-length grew to 42 bits for this ASIP implementation because of the extra bit needed to differentiate the additional instructions implemented and the register file addresses that are indicated with 4 bits in this case.

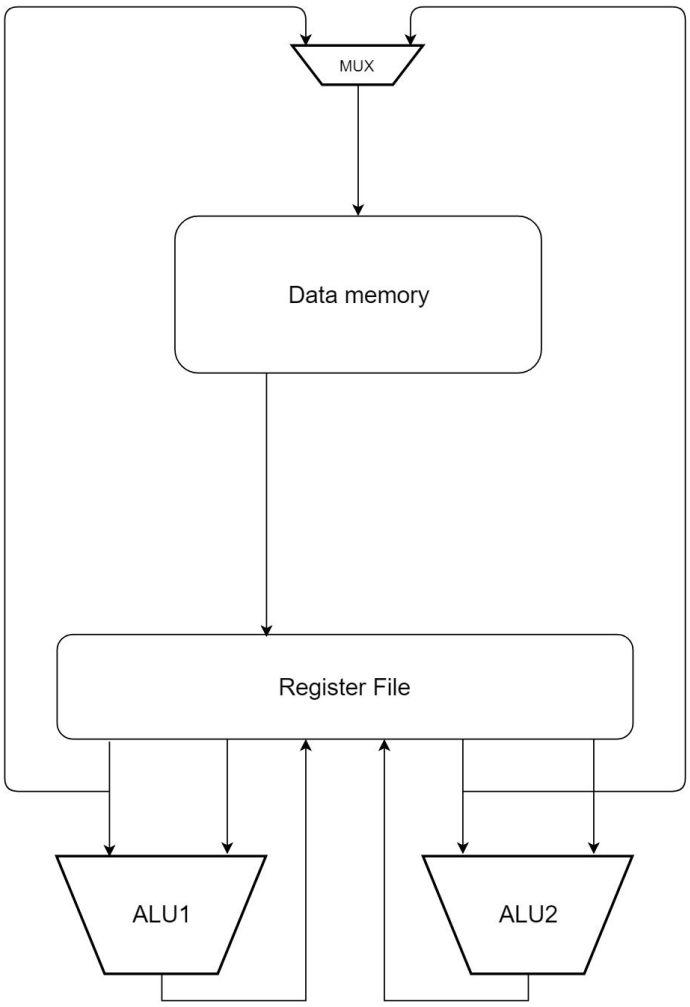

Figure 14: Modified Design 3 datapath

The new architecture presents a single register file which contains 16 registers. The register file is able to feed two identical arithmetic units and can load data from the Data Memory through a direct bus. The two arithmetic units are both able to use the Data memory for load/store operation but only one unit at the time can actually use it. The Data memory still presents a single port so it is not able to serve both arithmetic units simultaneously. A multiplexer selects the output from the correct unit each time a store instruction is executed.

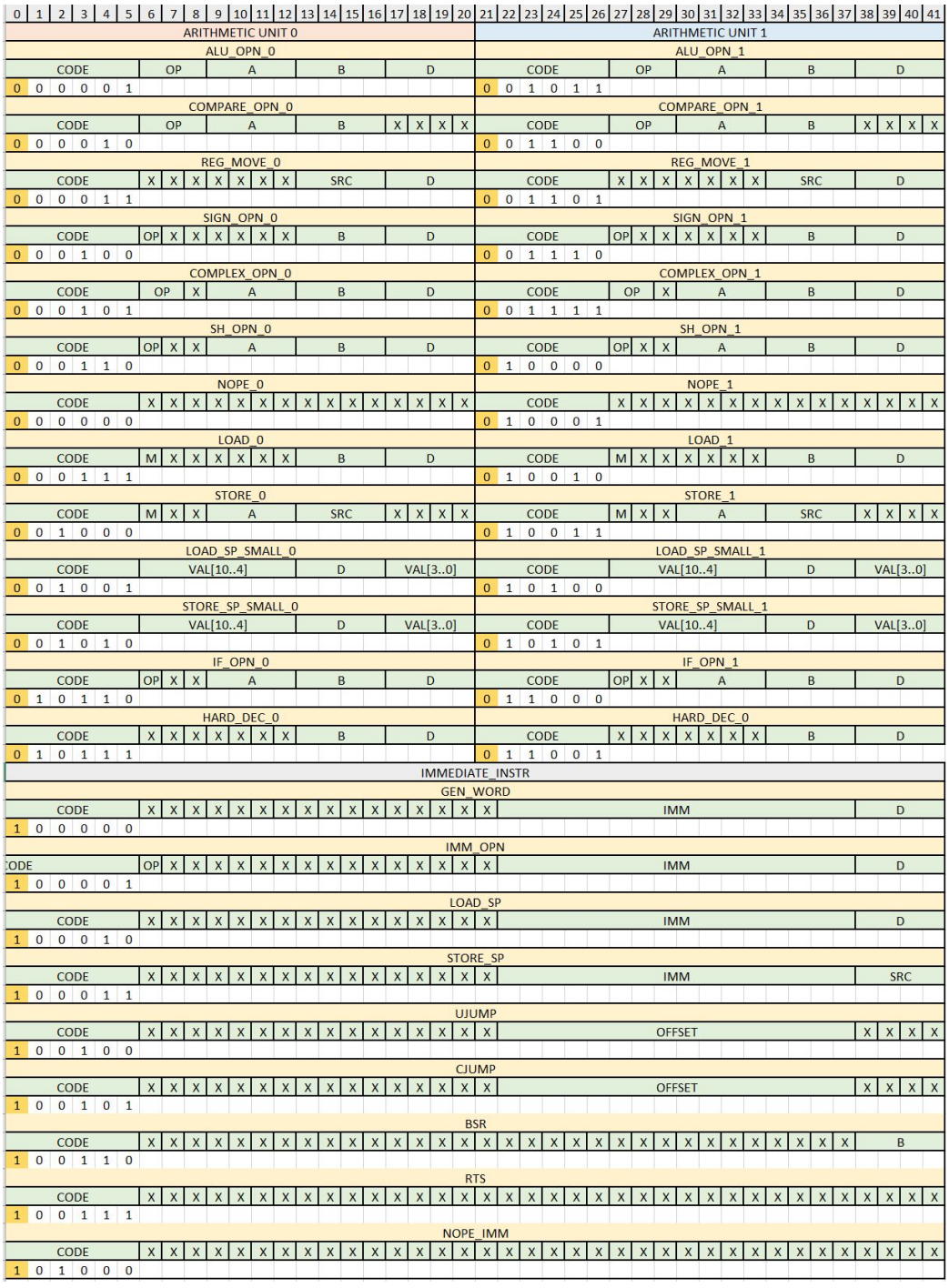

Figure 15: Design 3 instruction-set

Fig. 15 shows the new instruction-set of the modified version of Design 3.

This time, the parallel mode is able to execute both arithmetic and memory instructions on both the arithmetic units. Regarding the immediate mode, its functioning remains unchanged compared to the initial Design 3.

The modified Design 3 is tested with the 1024-codelength decoding simulation. Table 6 shows a comparison between the Design 3 simulation and the modified one using two algorithms: the normal one used for Design 3 and the loop unfolded version (indicated into the Table as "Modified Design 3 LU").

Design 3		Modified Design 3		Modified Design 3 LU	
Cycles	222104	Cycles	209184	Cycles	137437
ALU usage	34.66%	ALU1 usage	33.36%	ALU1 usage	42.79%
ADDU usage	26.11%	ALU2 usage	31.18%	ALU2 usage	40.12%
DM usage	21.84%	DM usage	18.58%	DM usage	28.14%
PM size	207	PM size	179	PM size	228

Table 6: Design 3 vs modified version with loop unfolding and normal algorithm

The comparison between the Design 3 simulation and the modified one without loop unfolding points out the benefits coming from the implementation of two identical arithmetic units connected to a single register files: the ALU1 and ALU2 have a perfectly balanced load in the modified version and the overall cycles needed to the decoding steps decreased by 5.8% without changing the decoding algorithm.
Regarding the loop unfolding version of the decoding algorithm, it further decreases the number of cycles by 38.12%. This is due to the increment of the arithmetic instructions weight over the control instructions coming from the conditional jumps that have to be verified at the end of each loop. Listing 2 illustrates a simple loop unfolding example: it is useful to understand how

the ASIP benefits from it. In the loop 1 version the compiler implements the instructions needed to the sum and then a conditional jump in order to verify the loop condition. This is repeated every loop cycle. In the loop 1 unfolded version, the compiler is able to execute the instructions needed to implement the two different sums before executing the conditional jump. Therefore, half conditional jumps are executed, decreasing the number of nop instructions and increasing the cycles where the datapath parallelism can be exploited.

Listing 2: Loop unfolding example

```
//loop 1
for(i = 0; i < N; i++){
        a[i] = b[i] + c[i];
}

//loop 1 unfolded
for(i = 0; i < N; i+=2){
        a[i] = b[i] + c[i];
        a[i+1] = b[i+1] + c[i+1];
}
```

6.3.2 Synthesis

In this section the synthesis results of both versions of Design 3 are presented. The synthesis is performed using the same technology and features exploited in the previous designs.

Design	2	3	Modified 3
Combinational Area (μm^2)	17226	17979	37041
Noncombinational Area (μm^2)	1970	3487	3620
Total Area (μm^2)	19195	21466	40660
Critical Path (ns)	4.8	4.8	4.8
Throughput (Mbps)	0.38	0.48	0.78

Table 7: Design 2 vs Design 3 synthesis results

Table 7 displays the synthesis results: the total area of Design 3 grows by only 11.83% respect Design 2 even if the hardware was doubled. This can be explained by the fact that the second arithmetic unit was not really a copy of

the first one but it was implemented with a smaller unit (ADDU) able to only perform address computations. Indeed, the computational area of Design 3 grows only by $753\mu m^2$ compared to Design 2 while the noncombinational area is much more increased because of the register file which was really doubled.

The modified Design 3 total area grows by 111.8% compared to Design 2 because the modified version of Design 3 was implemented with two identical arithmetic units. This is why the combinational area of the modified version is roughly twice the combinational area of Design 2.

Moreover, no changes are appreciated for the critical path delays of the three mentioned designs. This is due to the fact that the critical path corresponds to the path which goes from the output ports of the register file to the input port of the same unit passing across the slower part of the arithmetic unit. This path was not changed in the three different versions of the ASIP because the first arithmetic unit structure remains the same. Regarding the second implemented arithmetic unit, it is a simpler ALU for Design 3 which do not introduce any higher critical path for obvious reasons. In the case of the modified version of Design 3, the second arithmetic unit is an exact copy of the first one, so it does not introduce a different critical path.

The throughput of Design 3 and its modified version respectively increases by 26% and by 105% compared to Design 2. The throughput of the modified version of Design 3 was computed using the loop unfolding version of the decoding code.

6.4 Design 4

The Design 4 principal aim is to further enhance the ASIP performances implementing four different arithmetic units. The new datapath is displayed in Fig. 16.

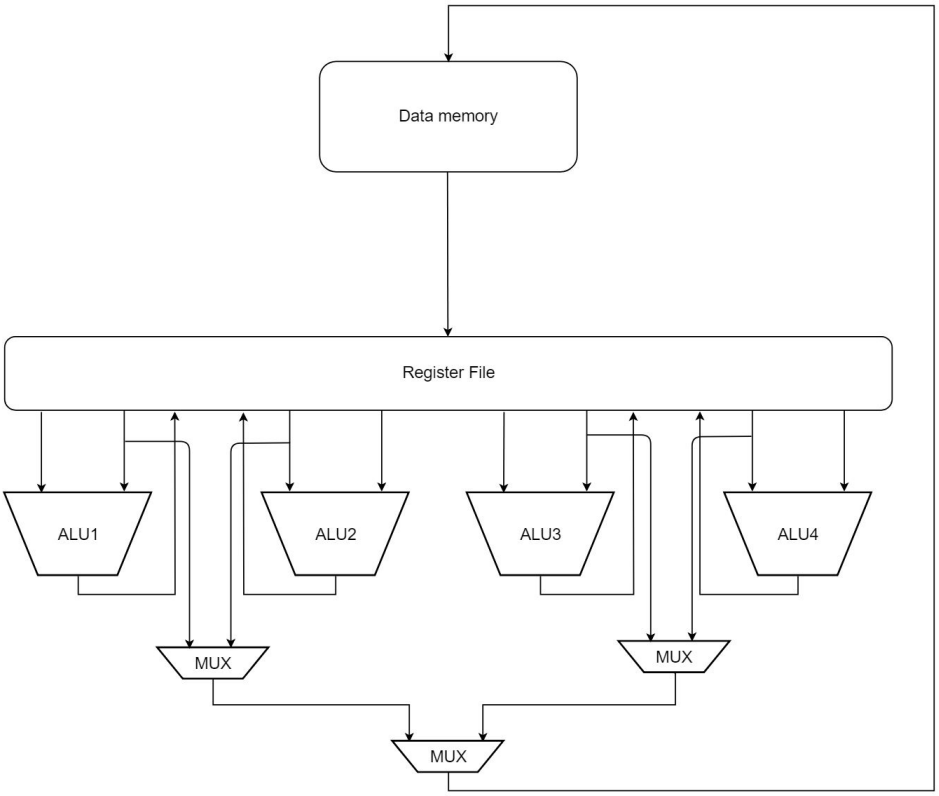

Figure 16: Design 4 datapath

The four arithmetic units are identical in order to avoid the problems faced in the previous sections. The ALUs interface the Data memory with a single port for load operations and a single port for store operations. This is the reason why three multiplexers were placed in order to select the data to be stored to the Data memory from the correct arithmetic unit.
Regarding the register file, it was composed by 16 registers as in the previous

design. The register file can simultaneously feed the four arithmetic units or store four computed values coming from them.

The Design 4 instruction-set is reported in Fig. 17. The division between immediate mode and parallel mode is still present. The parallel mode is able to execute four instructions in parallel exploiting the four arithmetic units. The immediate mode is also able to execute immediate instructions in parallel but only exploiting ALU1, ALU2 and ALU3 because only three instructions can be executed in this mode. A fourth instruction was not implemente in order to avoid an additional grow of the instruction word. Only ALU1 is able to execute the instruction control operations as jumps or return from subroutine instructions because they do not allow to run other instructions in parallel when they are executed. The grade-4 level parallelism brought the instruction bit-length to 88 bits, twice the length of the previous design.

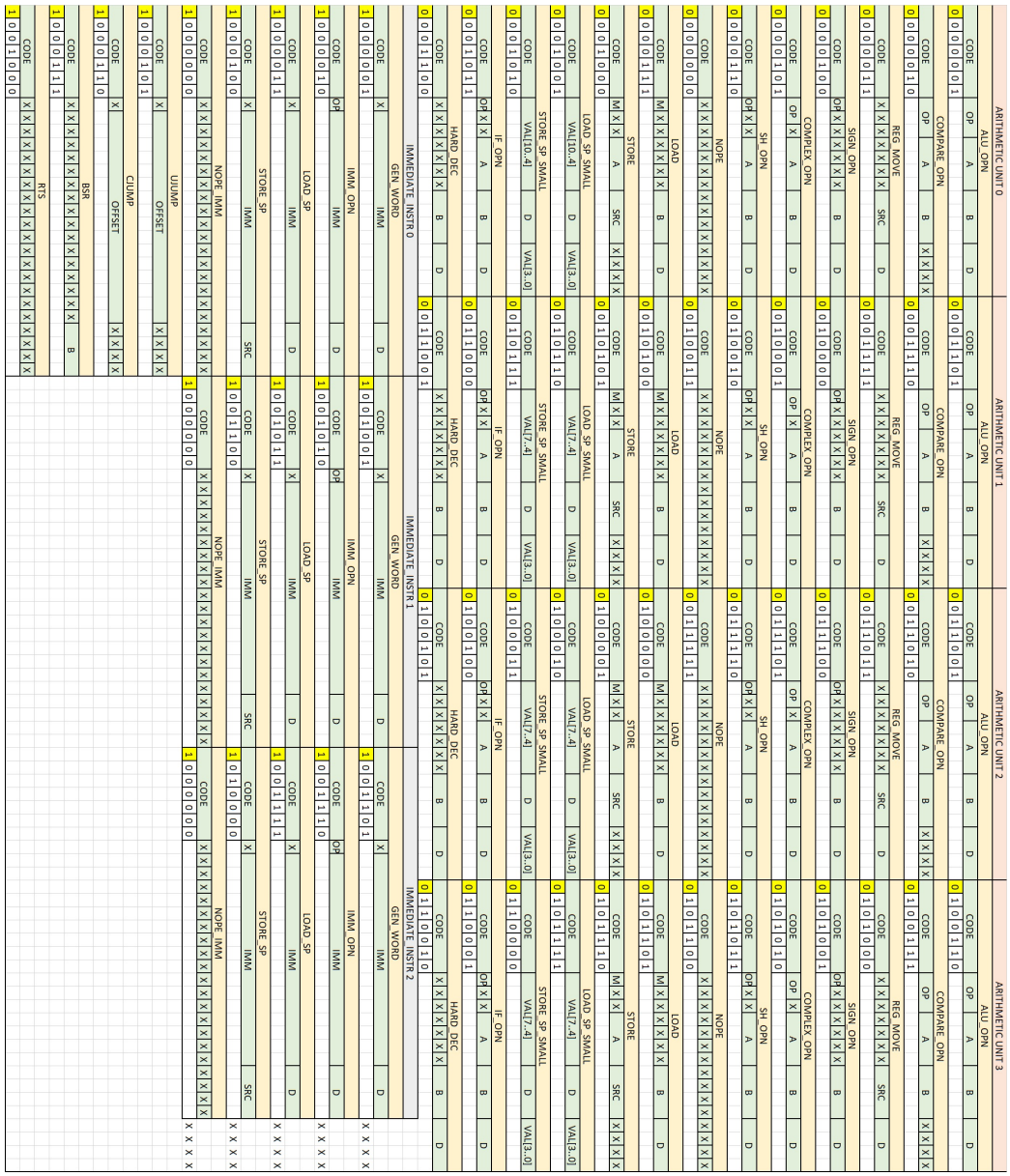

Figure 17: Design 4 instruction-set

The new Design is tested with the same code used in Section 6.3.1 but using a loop unfolding grade equal to four with the aim of adapting the decoding algorithm to the Design 4 parallelism grade. Table 8 shows a comparison between the modified version of Design 3 and the Design 4. Naturally, De-

sign 3 architecture was tested with the loop unfolded version of the decoded algorithm too.

Modified Design 3		Design 4	
Cycles	137437	Cycles	97502
ALU1 usage	42.79%	ALU1 usage	8.5%
ALU2 usage	40.12%	ALU2 usage	16.81%
/	/	ALU3 usage	34.78%
/	/	ALU4 usage	49.16%
DM usage	28.14%	DM usage	51.09%
PM size	228	PM size	282

Table 8: Modified Design 3 vs Design 4 simulation results

The Design 4 architecture was able to decode a frame with 29.06% less cycles than the modified version of Design 3. However, these results are not acceptable because of the unbalanced load recorded by the four arithmetic units of the Design 4. The new architecture was not able to correctly exploit the applied parallelism because of a blocking element: the Data memory. This can be demonstrated looking at the simulation results. The data memory recorded the highest usage among the other units. This means that double ports are needed for both load and store operations in order to properly serve the requests coming from the arithmetic units.

6.4.1 Design 4 with double ports for Data memory

Fig. 18 displays the Design 4 datapath using a Data memory with two ports for store operations and two ports for load operations. The four arithmetic units were divided in two groups and each group is able to use one different port of the Data memory. The ALU output is selected by a multiplexer. The register file is able to load data from the memory using the two load ports at the same time.

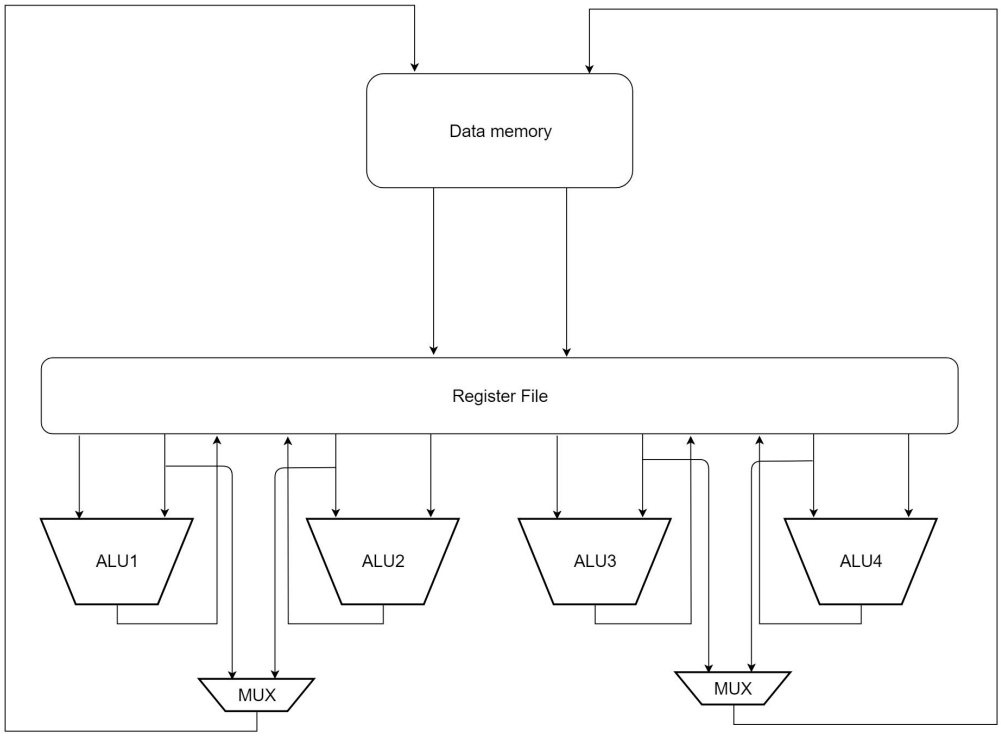

Figure 18: Design 4 datapath with double port for Data memory

Table 9 shows a comparison between the simulation results of Design 4 with single port and with double port for the Data memory. The Data memory usage decrement demonstrates that the previous guess was correct: the Data memory was a bottleneck for the ASIP. In the new Design 4 implementation the arithmetic units are more efficiently exploited since the overall work load is better distributed among them and the number of cycles requested by the decoding steps decreased by 10.68% compared to the single port version.

Design	4 (single port)	4 (double port)
Cycles	97502	87093
ALU1 usage	8.50%	14.70%
ALU2 usage	16.81%	25.39%
ALU3 usage	34.78%	36.72%
ALU4 usage	49.16%	45.58%
DM usage	51.09%	38.49%
PM size	282	261

Table 9: Design 4 single port vs double port for Data memory simulation results

The analysis of the assembly code generated by the software for the decoding process pointed out a problem: the parallelism of the Design 4 architecture is not completely exploited because it is not always possible to saturate the parallel mode or the immediate mode with respectively for arithmetic instructions or three immediate instructions. This handicap can be easily overcome if only one mode is implemented where both arithmetic and immediate instructions can be executed together.

6.4.2 Modified Design 4

The modified version of Design 4 presents some changes only for the instruction-set which is shown in Fig. 19. The instruction bit-length grew to 112 bits but the modified version of Design 4 is now able to execute four parallel instructions indifferently from their type. Each arithmetic unit can execute the same sub-set of instructions with the exception of the first one: it can execute four additional instructions that control the instruction fetch and, as it was explained before, they do not allow the execution of other instructions in parallel. The LOAD/STORE_SMALL_SP instructions were eliminated from this instruction-set because the normal SP instructions are now always available without switching ASIP mode.

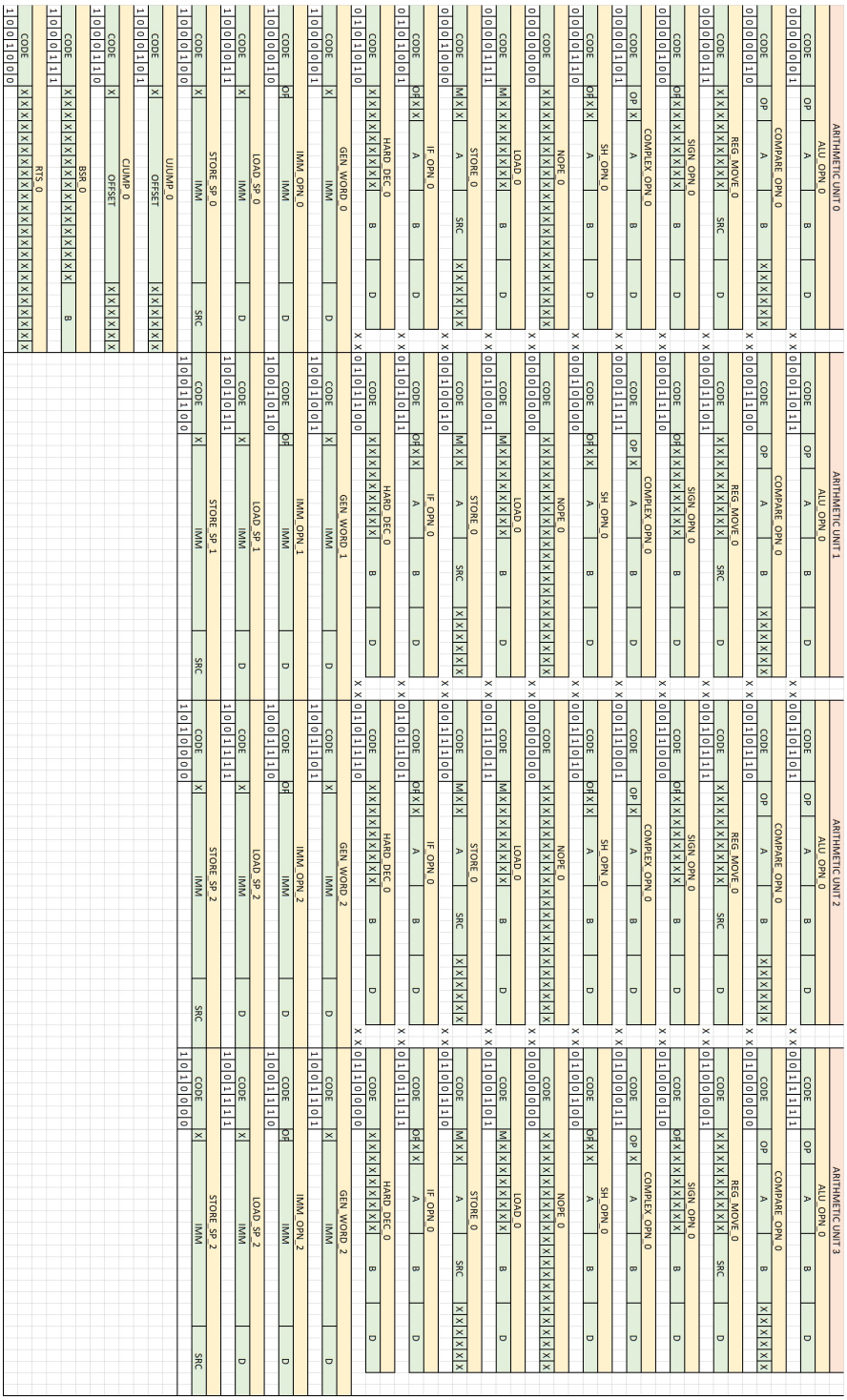

Figure 19: Modified Design 4 instruction-set

A comparison between the simulation results of Design 4 multiple port and the modified version of Design 4 is displayed in Table 10. The different instruction-set implementation brought a 2.6% decrement for the number of cycles. This improvement is lower than expected.
The modified Design 4 implementation was simulated step-by-step with a 32 bit code-length in order to investigate the blocking element which curbed the performance improvement. After several simulation cycles a problem was found for the register file: it was always saturated after a few cycles and so it was not able to proper feed all the arithmetic units. This type of problem emerged for the modified version of Design 4 because of the new instruction-set which allows a larger number of instruction types execution in parallel. Thus, a larger number of temporary values have to be loaded and stored into the register file increasing the number of locations required during each cycle.

Design	4 (double port)	Modified 4
Cycles	87093	84830
ALU1 usage	14.70%	14.87%
ALU2 usage	25.39%	21.79%
ALU3 usage	36.72%	32%
ALU4 usage	45.58%	56.99%
DM usage	38.49%	40.80%
PM size	261	257

Table 10: Design 4 double port vs modified Design 4 simulation results

6.4.3 Second Modified Design 4

The second modified version of Design 4 is implemented using the same instruction-set displayed by Fig. 19 and the same datapath shown in Fig. 18 with the only exception of the register file size: this ASIP version uses a larger register file composed by 32 registers with the aim of overcoming the

problem highlighted in the previous section.

Design	4 (double port)	Modified 4	Second Modified 4
Cycles	87093	84830	74213
ALU1 usage	14.70%	14.87%	19.85%
ALU2 usage	25.39%	21.79%	25.81%
ALU3 usage	36.72%	32%	37.35%
ALU4 usage	45.58%	56.99%	56.22%
DM usage	38.49%	40.80%	45.02%
PM size	261	257	256

Table 11: Design 4 versions simulation results

Table 11 summarises all the simulation results obtained for the different versions of Design 4 where the double port for the Data memory was implemented. Concerning the second modified version, the number of cycles needed for the decoding decreases by 12.52% compared to the first modified version. The bigger register file enhances the parallel usage of the four arithmetic units as it can be seen from the more balanced work load with respect to the previous Design 4 versions. Although, the higher flexibility given to the system in the successive versions of Design 4 increases the Data memory usage. When the arithmetic units are more efficiently exploited an higher throughput is required from the Data memory.

6.4.4 Synthesis

Table 12 presents the synthesis results for all the different versions of Design 4. The SP term indicates the single port version while the MP term indicates the multiple port version.

Design	4 (SP)	4 (DP)	Mod. 4	Sec. Mod. 4
Combinational Area (μm^2)	69697	69458	70209	77433
Noncombinational Area (μm^2)	4734	4761	5232	15563
Total Area (μm^2)	74432	74219	75441	105119
Critical Path (ns)	4.9	4.9	4.9	5.1
Throughput (Mbps)	1.07	1.2	1.23	1.35

Table 12: Designs 4 Synthesis

Comparing the Design 4 SP version with the modified Design 3 version, the total area increased by 83.07%. This is mostly due to the four arithmetic units that doubled the combinational area and to the additional multiplexers that were needed for the interfacing of the Data memory.

The total area between the Design 4 SP and MP versions remains almost unchanged. There is only a slightly decrement for the noncombinational area because of the lower number of multiplexers required for the memory interface.

Between the DP version and the first modified version, only a slightly increment for the total area is appreciated too. This small increment is given by the additional complexity required by the system in order to manage all the types of instructions in one mode and by each arithmetic unit. This is the reason why an increment is recorded for both the combinational and the noncombinational area.

Between the different Design 4 versions, the higher area increment is recorded comparing the first modified version with the second one. In this case, the total area grows by 39.34% because of both the combinational and the noncombinational area increments. In particular, the latter triplicates compared to the previous version even though the number of registers of the register file were only doubled. The reason for that was found in the additional complexity required by the register file in order to simultaneously vehicle information

from twice registers to the four arithmetic units and vice versa. In addition, higher complexity is also required in order to interface more registers to the Data memory load ports. Regarding the combinational area, a slightly increment is also recorded for the same reason: an additional complexity is required to interface more registers when the result has to be written back.
The critical path remains unchanged for all Design 4 implementations with the exception of the last version. An increase of 4.08% is recorded for the critical path of the second modified version of Design 4. This handicap is again due to the additional multiplexers needed to correctly interface the arithmetic units with the 32 registers of the register file.
The Table also reports the constant throughput increment for each successive implementation of the ASIP.

6.5 Design 5

The main purpose of Design 5 implementation is to solve the problem of the Data memory high usage implementing a new type of unit: a vector arithmetic unit. It is a module able to perform the same types of operations executed by the ALUs but using a different type of operators: vectors of word. This means that in a single cycle the vector arithmetic unit is able to process several words at the same time. In addition, the vector arithmetic unit is connected to a different register file, called vector file, able to store an entire vector in each location. Finally, the main advantage of this new unit is the possibility to perform load and store operations moving simultaneously several words from/to the Data memory as vectors. This last feature can be very useful in order to solve the Data memory high usage problem previously highlighted.
Regarding the Data memory, it is now able to work in two different modes:

- Single word mode: the memory is accessed by normal addresses and processes one word at the time. This mode is needed to correctly feed the normal arithmetic units.

- Vector mode: the memory is accessed only by addresses multiple of the vector length computed by the vector unit. The memory is able to process an entire vector of words in parallel.

The two modes can not be used simultaneously. The memory can be accessed only with one mode in each cycle. Double ports are still present for the single word mode but only a single port is available for the multi-word mode.
The complete datapath of Design 5 is shown in Fig. 20.

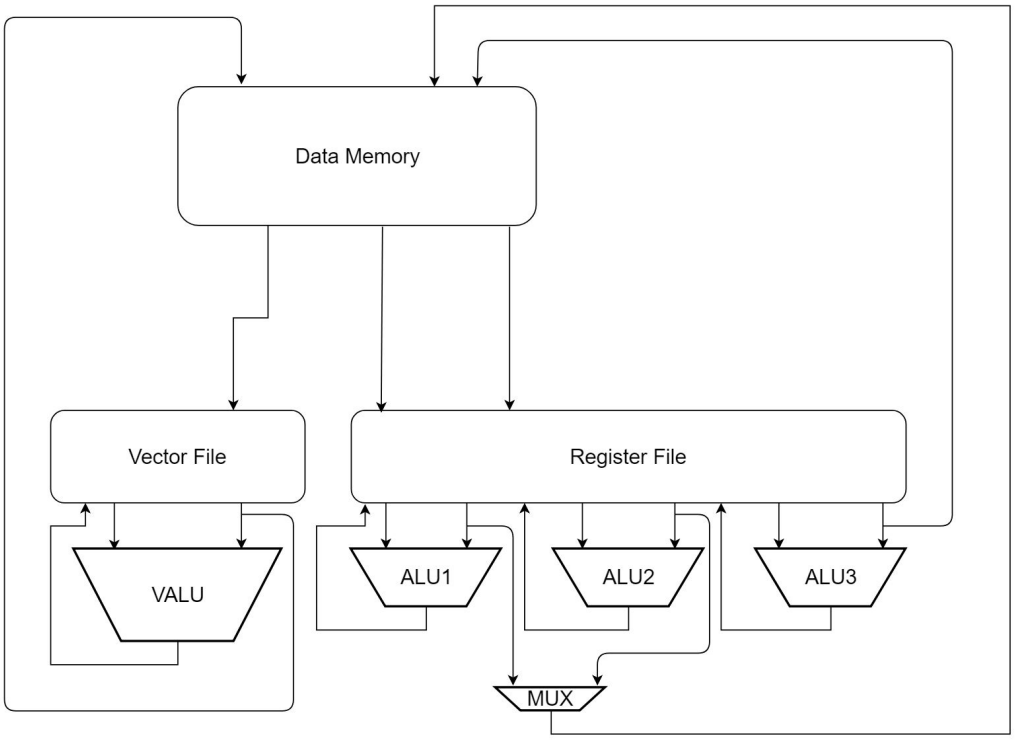

Figure 20: Design 5 datatpath

The register file is still implemented with 32 registers while the vector file contains 8 locations where 8 different vectors can be stored.
The instructions are coded in 116 bits and the instruction set is shown in Fig. 20. The instructions that control the instruction flow are executed only by ALU1 while ALU4 instructions are substituted by the instructions executed by the vector unit.

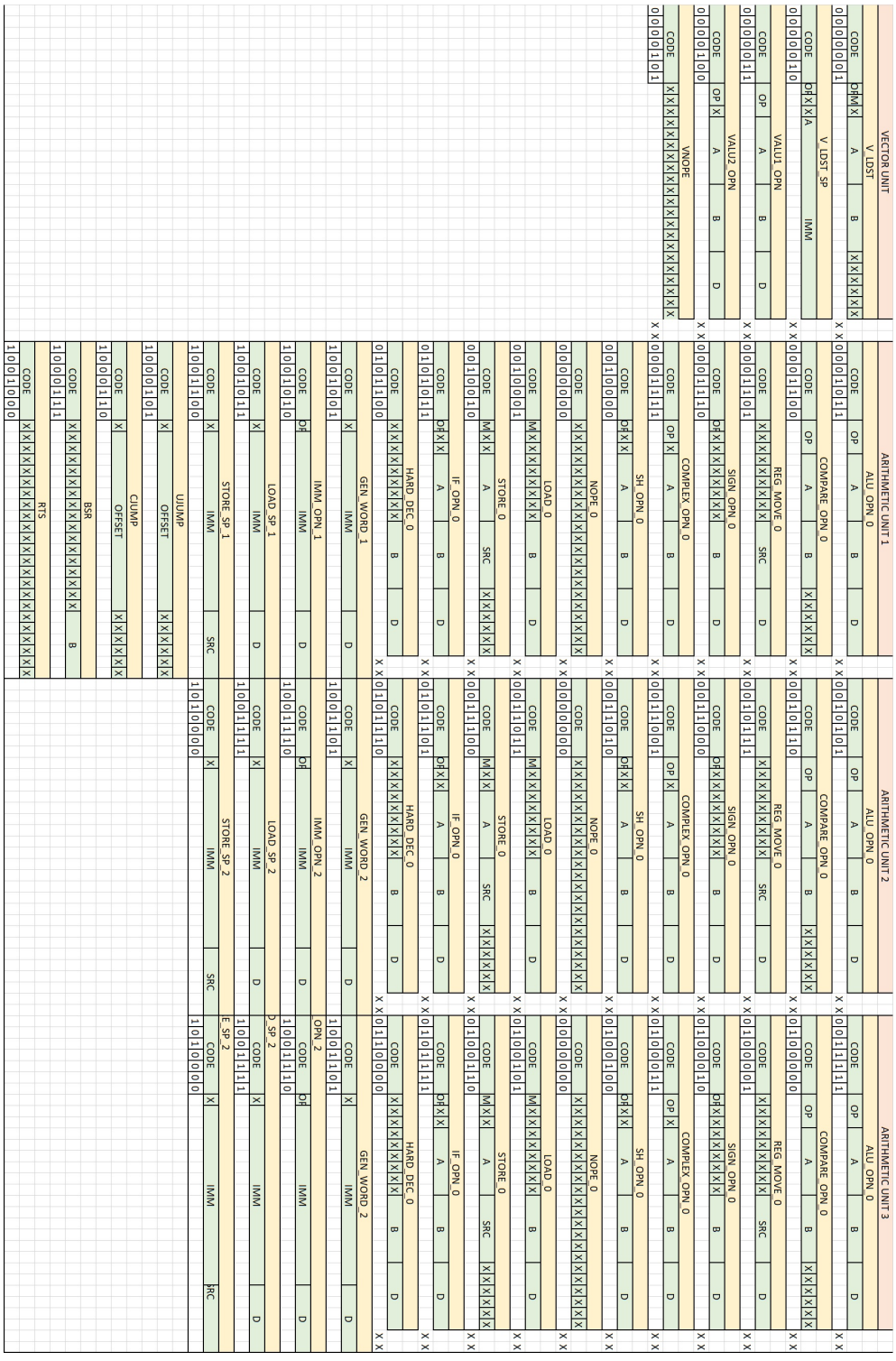

Figure 21: Design 5 instruction-set

The vector units executes new types of instructions: the V_LDST and V_LDST_SP are instructions able to use the vector mode of the Data memory and load/store an entire vector from/to the memory in one single cycle. The other two instructions run by the vector unit are: VALU1_OPN and VALU2_OPN.
The VALU1_OPN processes only one vector at the time and it can select between the following operations:

- Vector sign: returns a vector containing the signs of the elements stored in the processed vector.

$$Y[i] = sign(A[i]) \tag{16}$$

- Vector arf: returns a vector Y containing elements elaborated with the following formula:

$$Y[i] = 1 - (2 * A[i]) \tag{17}$$

  This operation is needed in order to prepare vectors for equation 5.

- Vector ABS: returns a vector containing the absolute module of the elements contained in the input vector

$$Y[i] = abs(A[i]) \tag{18}$$

- Vector hard decision: returns a vector which contains the hard decision performed on each element of the input vector, according to formula 7.

$$Y[i] = HARD\_DEC(A[i]) \tag{19}$$

- Vector froze: returns a vector containing only frozen values. This operation is required in order to store a segment which contains only frozen bits in the Data memory with a single store operation.

$$Y[i] = 0 \tag{20}$$

Whereas, the VALU2_OPN processes two vectors as operands at the time and it can select between the following operations:

- Vector minimum: returns a vector containing the minimum values between the elements of the two input vectors at the same position.

$$Y[i] = min(A[i], B[i]) \tag{21}$$

- Vector XOR: returns a vector containing the XOR operation results between the two elements of the input vectors at the same position.

$$Y[i] = A[i] \oplus B[i] \tag{22}$$

- Vector sum: returns a vector containing the sums between the elements of the two input vectors at the same position.

$$Y[i] = A[i] + B[i] \tag{23}$$

- Vector multiplication: returns a vector containing the multiplications between the elements of the two input vectors at the same position.

$$Y[i] = A[i] * B[i] \tag{24}$$

The Design 5 is simulated with the decoding code adapted to the vector unit which was detailed in section 5.3. It is important to underline that only the nodes that compute vectors of bits with length equal or multiple to the length of the vectors computed by the vector unit can benefit from the vector computations. The nodes that handle less bits (like the nodes at the end of the tree shown in Fig. 6, for example) can not benefit from the vector computation and they have to use the normal arithmetic units. In consequence of this fact, the choice of the vector length which is computed from the vector units becomes fundamental for the achievable performances increment.

According to the tree distribution of the nodes, lower is the length of the vectors higher is the number of nodes that benefits from the vector computation. On the other hand, lower is the vector length processed by the vector unit lower will be the speed-up. A trade-off has to be found simulating the architecture with different length sizes with the aim of choosing the best configuration. The number of simulations needed to do this analysis is limited because only the lower vector sizes have the possibility to exploit the vector computation for an higher number of nodes.

Vector length	4	8	16	32
Cycles	61094	47160	46064	53230

Table 13: Number of cycles with different vector lengths

Table 13 exposes the number of cycles needed to complete a single frame decode using different vector lengths. The best performances are obtained with a vector length of 16 then, with a vector length of 32 the needed cycles start to increase. However, between the vector lengths 8 and 16 there is a difference of only 2.3% in terms of performance but the integration of the vector unit with a vector length of 16 requests twice the area needed by a

vector length of 8. For this reason, the vector length for Design 5 was set to 8.

Inside the vector unit is embedded a second unit able to compute the addresses for all the vectors that have to be calculated and stored during the ASIP functioning. A specialised unit for the vector addresses was needed because the memory has to be accessed with addresses multiple of the vector length in order to correctly extract the vectors of word in vector mode. Furthermore, the embedded vector address unit has to compute addresses for the vectors to be stored that do not create conflict with the other data stored into the Data memory.

Table 14 shows the simulation results of Design 5 compared to the last version of Design 4. The number of cycles required by Design 5 to complete the single frame decode decreased by 39.1% compared to the second modified version of Design 4. It is also important to underline the Data memory usage decrement for Design 5 when, for Design 4, it was the main reason to slowdowns. However, the ALU1 and vector units are not exploited enough in Design 5. More frames can be decoded simultaneously in order to increase their usage.

Design	Second Modified 4	5
Cycles	77433	47160
ALU1 usage	19.85%	5.72
ALU2 usage	25.81%	18.19%
ALU3 usage	37.35%	39.59%
ALU4 usage	56.22%	/
VALU usage	/	8.33%
VADD usage	/	4.69%
DM usage	45.02%	28.11%
PM size	256	275

Table 14: Second modified Design 4 vs Design 5 simulation results

6.5.1 Modified Design 5

The modified version of Design 5 is introduced in order to allow the computation of multiple frames in parallel. This is needed because of the low usage recorded for the vector unit and the ALU1 in the simulation of Design 5. However, the simultaneously decoding of multiple frames requires a bigger Data memory to store all the partial data elaborated during the functioning. This is the reason why the modified version of Design 5 is designed with 18-bits data parallelism which is the same length of the new Data memory address which can now interface 2^{18} different word locations. This is the only modification performed on Design 5 architecture. The datapath and the instruction-set do not change compared to the ones shown in Figs. 20 and 21.

Design	5 (1 frame)	Modified 5 (2 frames)	Modified 5 (4 frames)
Cycles	47160	57248	83302
ALU2 usage	5.75%	9.55%	15.79%
ALU3 usage	18.19%	25.92%	30.41%
ALU4 usage	39.59%	41.16%	39.67%
VALU usage	8.33%	13.72%	18.85%
VADD usage	4.69%	7.74%	10.63%
DM usage	28.11%	37.50%	47.12%
PM size	275	328	460

Table 15: Design 5 vs modified Design 5 simulation results with multi frame decoding

Table 15 displays the simulation results of Design 5 and the modified version of Design 5 with the parallel decoding of two frames and four frames. Naturally, if more frames are computed in parallel the usage of the different arithmetic units will be higher. The decoding of multiple frames also affects the Program memory size which has to be bigger in order to contain the instructions of a longer code where all the the operations needed to decode

the different frames are listed. Regarding the ASIP performance, this time the number of cycles are not the absolute value of confrontation but the real throughput has to be used. In this way, the modified Design 5 version with 2 frames parallel decoding improves by 39.3% compared to Design 5 while the same modified Design 5 version, but with 4 frames parallel decoding, improves by 55.84% compared to Design 5.
However, with an higher number of frames the usage of the Data memory increases. This is due to the higher data request coming from the arithmetic units that now have to compute several frames at the same time. This is the reason why in the following ASIP implementation a double port for the vector mode of the Data memory will be implemented.

6.5.2 Synthesis

Table 16 reports the synthesis results for Design 5 and its modified version. Between the two versions there is an overall area increment of around 20% which is directly proportional to the data parallelism increment. Regarding the critical path delay, it increases by 7.8% for the modified version of Design 5 because of the higher complexity that the computational units have to manage with an higher data parallelism. However, the performance increment recorded for the four frames parallel decoding was much higher than the critical path delay handicap.

Design	5	Modified 5
Computational Area (μm^2)	89556	107906
Noncomputational Area (μm^2)	15562	17301
Total Area (μm^2)	105119	125207
Critical Path (ns)	5.1	5.5
Throughput (Mbps)	2.13	4.47

Table 16: Design 5 and modified version of Design 5 synthesis

As mentioned before, the throughput of the modified version of Design 5 improves even if its critical path is higher than the original Design 5. In detail, the throughput increases by 110% compared to the Design 5 implementation.

6.6 Design 6

The Design 6 is the last architecture that will be presented in this thesis and it was designed in order to further improve the ASIP performance doubling both the arithmetic units and the vector one. As consequence of that, some problems already faced in the previous sections will recur, as the Data memory slowdowns or the register file saturation.

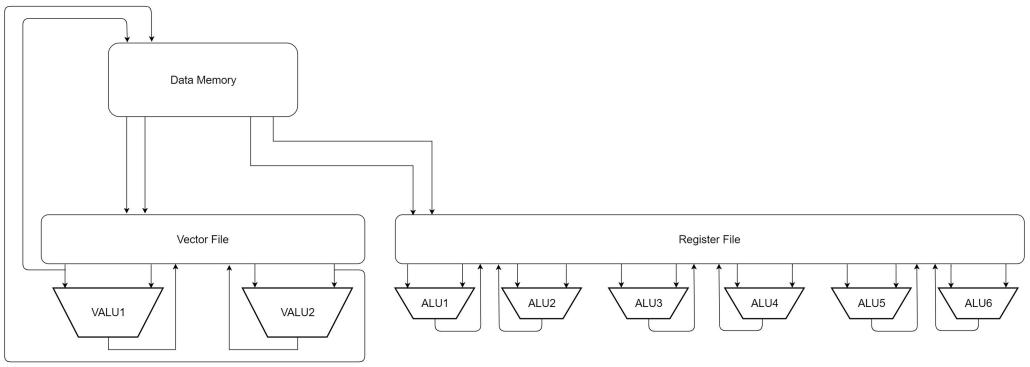

Figure 22: Design 6 architecture

Fig. 22 displays the architecture of Design 6. This time, the Data memory has double ports for vector mode in order to overcome the problems faced in section 6.5.1. The normal arithmetic units are also able to interface the Data memory with double ports but the store buses were not reported in the architecture in order to have a cleaner representation of the whole datapath. Although, the Data memory is still able to work only in one mode at the time. The vector file now contains 16 locations for vectors stores. It was doubled in order to avoid saturation problems since the Vector file was always saturated when it was tested in the previous section. The register file dimension remains unchanged because despite the case of the Vector file, the Register file was not completely used in the previous ASIP implementation. Finally, Design 6 implements instruction words of 250 bits. The instruction-set is the same

as the one shown in Fig. 21 but it was doubled in order to manage all the additional units of Design 6.

Design	Modified 5 (4 frames)	6 (4 frames)	6 (6 frames)
Cycles	83302	64098	86640
ALU1 usage	15.79%	5.17%	3.01%
ALU2 usage	30.41%	5.36%	5.21%
ALU3 usage	39.67%	10.37%	7.38%
ALU4 usage	/	13.59%	16.54%
ALU5 usage	/	31.08%	35.24%
ALU6 usage	/	45.60%	51.18%
VALU1 usage	18.85%	11.47%	10.91%
VALU2 usage	15.79%	13.03%	16.28%
VADD1 usage	/	4.55%	5.88%
VADD2 usage	/	9.27%	9.46%
DM usage	47.12%	47.31	55.91%
PM size	460	385	508

Table 17: Design 5 vs Design 6 simulation results with multiple frame decoding

Table 17 reports the simulation results of the modified version of Design 5 and Design 6 with 4 frames and 6 frames parallel decoding. The Design 6 architecture improves the throughput by 23.05% compared to the modified version of Design 5 in the case of 4 frames parallel decoding for both implementations. The usage of the vector units and the first arithmetic units is still too low so a 6 frames parallel decoding was simulated. However, the results are disappointing: it is true that the performances of the 6 frames parallel decoding improves by 9.89% but the usage of the previous mentioned units remains too low. This is due to the slowdowns introduced by the Data memory whose usage increased to 55.91%, far higher than the other units. This was anticipated at the start of this section and can be solved implementing more ports for the Data memory.

However, it is very uncommon to use memories with more than two ports

because of the additional complexity which would decrease too much the memory speed. Usually, two different approaches can be adopted:

- Divide the memory in more banks where the data is organised in different groups so that the computational units can access different banks at the same time. This is a good approach for power consumption because it allows to turn-off the banks that are not used but, on the other hand, the data has to be well organised inside the different banks with the aim of avoiding that two different units need to access the same bank at the same time.

- Divide the memory in more banks where the data is interleaved between them. This is a good approach in terms of performance because it is very common that two different units need to access to two sequential data at the same time but, on the other hand, the banks stay on for the majority of the time.

In addition, the microcode provided by ASIP Design for Design 6 pointed out a particular problem: with the higher level parallelism the actual instructions where the parallelism is efficiently exploited begin to be comparable with the total number of nops and conditional jump checks run at the end of each code loop. This problem can be easily overcome using an hardware loop.
The Design 6 was updated in order to integrate an hardware loop unit embedded into the ALU1 so that the handicap coming from the instructions executed after each loop instantiated into the C code was eliminated. The ALU1 was chosen to implement the hardware loop because it was the least used ALU of Design 6. The structure of the hardware loop unit is displayed by Fig. 23.

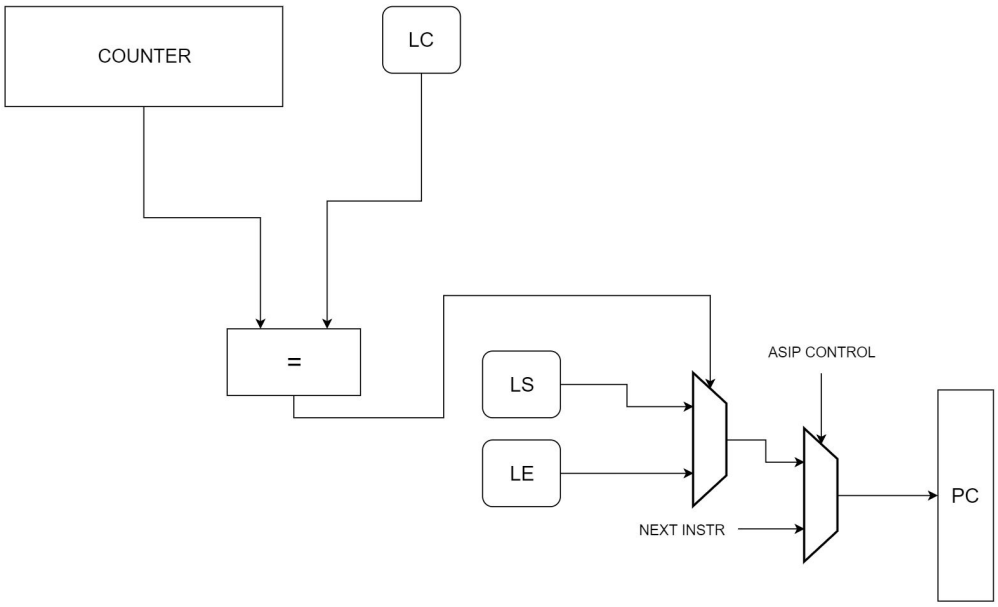

Figure 23: Hardware loop unit

The hardware loop unit is composed by a counter and three registers. When a hardware loop instruction is executed (instruction syntax reported in Fig. 24) the number of cycles needed for a certain loop is transferred from the A register of the RF to the LC register. The start address of the loop and the end address of the same loop (provided by the instruction code) are respectively stored in the LS and LE registers. Every time the last instruction of the loop routine is executed, the counter is incremented and its output is compared to the value stored into the LC register. When the two numbers match, the LE address is fetched into the program counter. Otherwise, the program counter fetches the address provided by the LS register so that the loop routine is restarted.

HARDWARE LOOP																																								
CODE								END ADDRESS																								A											
1	1	1	1	1	1	1	1																																			

Figure 24: Hardware loop unit instruction syntax

The simulation results of the old Design 6 and the one upgraded with the hardware loop are presented in Table 18. Both designs are simulated using the six frame parallel version of the decoding code. The introduction of the hardware loop decreased the cycles needed for the decoding steps by 19.17% compared to the previous design. The upgraded Design 6 also recorded a usage increment for all the reported units and a reduction of 35 instructions for the program memory size. Both the changes are due to the elimination of the nop and the conditional jump instructions related to the loop condition check.

Design	6 (old)	6
Cycles	86640	70033
ALU1 usage	3.01%	6.47%
ALU2 usage	5.21%	6.71%
ALU3 usage	7.38%	10.16%
ALU4 usage	16.54%	19.82%
ALU5 usage	35.24%	43.74%
ALU6 usage	51.18%	57.98%
VALU1 usage	10.91%	15.49%
VALU2 usage	16.28%	18.14%
VADD1 usage	5.88%	8.82%
VADD2 usage	9.46%	10.16%
DM usage	55.91%	66.59%
PM size	508	473

Table 18: Old Design 6 vs upgraded Design 6 simulation results using six frame parallel decoding

6.6.1 Synthesis

Table 19 displays the synthesis results for the modified version of Design 5 and the upgraded version of Design 6. The total area of Design 6 increases by 109.21% compared to the previous version. This is the results of the doubled arithmetic units, the doubled vector file and additional complexity needed

to interface the Data memory with double ports for both vector mode and single mode plus the additional complexity needed to vehicle the information between the doubled vector file and the other units. The critical path increases too but this result is negligible because the throughput of the Design 6 implementation increases by 72.3% compared to the modified version of Design 5.

Design	Modified 5	6
Computational Area (μm^2)	107906	225750
Noncomputational Area (μm^2)	17301	30138
Total Area (μm^2)	125207	255888
Critical Path (ns)	5.5	5.7
Throughput (Mbps)	4.47	7.7

Table 19: Modified version of Design 5 vs Design 6 Synthesis

7 Conclusions

All the designs detailed in the past sections are reported in Table 20. Each line contains the total area and the throughput (Th) with whom each design was characterised in its last version. There is also a column where the area efficiency is reported. The area efficiency is computed with the ratio between the throughput and the total area. It provides a good point of confrontation for the different architectures because it is a measure of the performance provided by each area unit. Thus, higher is the area efficiency higher is the effectiveness of the different architectural choices for each design.

Design	Area (mm^2)	Th ($Mbps$)	Area Eff. ($Mbps/mm^2$)
1	0.018	0.142	7.89
2	0.02	0.38	19
3	0.041	0.78	19.02
4	0.11	1.35	12.27
5	0.13	4.47	34.39
6	0.26	7.7	29.62

Table 20: Results summary

The best design in terms of required area is the first one but it is also the worst design in terms of performance. The best throughput is provided by the Design 6 implementation thanks to its two vector units, six arithmetic units and six parallel frame decoding code run but, on the other hand, Design 6 is also the architecture with the higher implementation area.
Regarding the area efficiency, Design 6 is not the best choice even with the addition of the hardware loop. The reason of that can be found in the high usage of the Data memory which is the bottleneck of the system. As it was stated in section 6.6, a Data memory with an higher number of port for the load and store operations would increase the performance of Design 6 re-wording this last architecture as the best one even in terms of area efficiency.

However, if only the results presented in this thesis are considered, the best area efficiency is provided by Design 5. The introduction of the vector unit and the parallel frame decoding code was able to nearly triple the area efficiency of Design 4 which suffered from the bottleneck provided by the Data memory. Design 2 also deserves a special attention because it was capable of providing an area efficiency 141% higher than the previous implementation with an area handicap of only 11%. This was possible thanks to the introduction of "specialised" instructions that were able to compute functions of the decoding algorithm using only one cycle. It is important to stress this concept because it is the distinctiveness which allows to the ASIP implementations to be way more efficient than the general purpose implementations.

7.1 State-Of-The-Art Comparison

Finally, a comparison between the ASIC decoders reported in section 3.1 and the fastest ASIP designed in this work (Design 6) will be presented. Table 21 displays throughput and implementation area for each design.

	This Work	[13]	[14]	[15]
Th (Mbps)	7.7	49	132	187
Area (mm^2)	0.26	1.72	0.31	0.3
Area Norm. for $65nm$ (mm^2)	0.26	0.22	0.31	1.62

Table 21: State-of-the-art comparison between throughput and implementation area

The ASIP implementation of this thesis is the slowest one between the reported architectures, as it was anticipated in the first sections. The ASIC designs are able to provide a throughput one order of magnitude higher than the presented ASIP. Regarding the implementation area, the different architectures were implemented with different technologies so a proper comparison

can be done using the last row of Table 21 where a normalized value of the area is computed using the 65-nm technology as reference. In this case, the implementation area of the ASIP presented in this work is comparable with the examples reported in [13] and [14]. An exception is made for the ASIC presented in [15] where the normalized implementation area is seven times higher than the developed ASIP. This is due to the possibility offered by the ASIC of setting the code-rate and the frozen-bit positions and to the implementation of three different decoding algorithms with whom the decoder can be set. Another important point of comparison is the number of processing elements implemented in each architecture. The higher performance of this work is reached with a parallelism grade of 22 counting each vector arithmetic unit as eight parallel processing elements. Whereas, the state-of-the-art ASICs exploited a parallelism grade that was much higher and allowed to reach the displayed performances without an excessive area handicap.

References

[1] E. Arikan, "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels", 2009.

[2] Alexandre J. Raymond, "Design and Hardware Implementation of Decoder Architectures for Polar Codes", 2013.

[3] Gabi Sarkis and Warren J. Gross, "Increasing the Throughput of Polar Decoders", 2013.

[4] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive cancellation decoder for polar codes", 2011.

[5] Anthony Barre, Emmanuel Boutillon, Neysser Blas and Daniel Diaz, "A polar-based demapper of 8PSK demodulation for DVB-S2 systems", 2013

[6] www.umc.com, "65 Nanomiter", 2005

[7] Synopsys, "ASIP Designer: Design Tool for ApplicationSpecific Instruction-Set Processors", 2018

[8] Synopsys, "Guidelines for Hands-On Training", 2017

[9] Synopsys, "The nML Processor Description Language", 2017

[10] Synopsys, "Chess Compiler Processor Modeling Manual", 2017

[11] Synopsys, "Checkers Simulator Manual", 2017

[12] Synopsys, "Go User Manual", 2017

[13] A. Mishra, A. J. Raymond, L. G. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg and W. J. Gross, "A successive Cancellation Decoder ASIC for a 1024-bit Polar Code in 180nm CMOS", 2012

[14] Camille Leroux, Alexandre Raymond, GabiSarkis and Warren Gross, "A Semi-Parallel Successive-Cancellation Decoder for Polar Codes", 2012

[15] Pascal Giard, Alexios Balatsoukas-Stimming, Thomas Christoph Müller, Andrea Bonetti, Claude Thibeault, Warren J. Gross, Philippe Flatresse, Andreas Burg, "POLARBEAR: A 28-nm FD-SOI ASIC for Decoding for Decoding of Polar Codes", 2017

[16] O. Afisiadis, A. Balatsoukas-Stimming, and A. Burg, “A low-complexity improved successive cancellation decoder for polar codes”, 2014

[17] I. Tal and A. Vardy, “List decoding of polar codes”, 2015

[18] Rachid Al-Khayat, Purushotham Murugappa, Amer Baghdadi, Michel Jez equel, "Area and Throughput Optimized ASIP for Multi-Standard Turbo decoding", 2011

[19] Purushotham Murugappa, Rachid Al-Khayat, Amer Baghdadi and Michel Jezequel, "A Flexible High Throughput Multi-ASIP Architecture for LDPC and Turbo Decoding", 2011

[20] Vianney Lapotre, Purushotham Murugappay, Guy Gogniat, Amer Baghdadiy, Jean-Philippe Diguet, Jean-Noel Bazin, and Michael Hubner, "Optimizations for an Efficient Reconfiguration of an ASIP-Based Turbo Decoder", 2013

[21] Meng Li, Youngjoo Lee, Yanxiang Huang and Liesbet Van der Perre, "Area and energy efficient 802.11ad LDPC decoding processor", 2015