POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Tightly-coupled VS Loosely-coupled accelerators for data compression in space applications: a NOEL-V case study

Supervisor	Candidate
Prof. Guido MASERA	Enrico MANFREDI

April 2025

Abstract

The growing complexity of space systems and missions is causing a rapid increase of onboard data size, to the point where efficient data compression algorithms become essential. The resource constrained environment requires careful consideration of how to optimize this procedure. On the other hand, NOEL–V is emerging as a promising space-grade processor, offering both high performance and reliability for critical space applications. Being built on the RISC–V ecosystem, NOEL–V inherits its ability to enhance performance through the integration of hardware accelerators with the processor. Tightly-coupled and loosely-coupled accelerators (TCAs and LCAs respectively) offer different benefits: the former are more compact but require more effort to integrate them, while the latter should be carefully designed to minimize their area. This work will compare those two approaches regarding the optimization of the CCSDS 121.0 algorithm, a lossless data compression technique recommended as a standard by the Consultative Committee for Space Data Systems. Synopsys ASIP Designer has been used to design and integrate the accelerators into a NOEL–V model. Two variants of TCAs will be proposed, each targeting specific bottlenecks of the algorithm, reaching a total clock cycle reduction of 85%, with an area increment of only 9% (based on 65 nm low power ASIC technology). The proposed memory-mapped LCA performs the entire compression procedure, reaching a speed-up factor of 480× and a 48% area increment with respect to the reference implementation. With a throughput of 7.83 Gb/s, the LCA turned out to be the far superior design in terms of Throughput-Area Ratio, but the TCAs proved to be a valid choice for lossless data compression optimization in area-constrained environments.

Keywords: Hardware design, accelerators, space, RISC–V, NOEL–V, data compression, CCSDS 121.0, ASIP Designer.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Guido Masera, whose expertise and guidance have been fundamental in shaping this work.

I extend my sincere appreciation to my parents, Mauro and Elisabetta, my brothers, Pietro and Stefano, and all my family for their unwavering support, valuable advice, and patience throughout my academic career. Their encouragement has been a constant source of motivation, helping me face challenges with resilience and determination. I am deeply grateful for everything they've done to support me in achieving this goal.

A very special thank you goes to my girlfriend, Federica, for her constant support, understanding, and encouragement, which have meant so much to me throughout this path. Her presence has been a source of strength and comfort, helping me stay motivated even in the most challenging moments. I truly appreciate her patience, love, and belief in me, which have made this experience even more meaningful.

I would like to thank all my friends, near and far, and my university colleagues, for their support, collaboration, and the enriching discussions we've had throughout this journey. Their presence has made this experience not only more enjoyable but also intellectually stimulating, helping me grow both personally and academically. Together, we've shared challenges and achievements, making this time truly unforgettable.

Lastly, I would also like to express my heartfelt gratitude to Professor Rossana Salvetto, my middle school teacher, whose passionate and engaging teaching style sparked my enthusiasm for studying scientific subjects, and to all the teachers and professors who have guided me throughout my educational journey.

Thank you all.

Table of Contents

List of Tables

List of Figures

Acronyms

ISA

Instruction Set Architecture

ISE

Instruction Set Extension

ASIP

Application Specific Instruction Set Processor

ASIC

Application Specific Integrated Circuit

TCA

Tightly Coupled Accelerator

LCA

Loosely Coupled Accelerator

ALU

Arithmetic and Logic Unit

ISS

Instruction Set Simulator

AMBA

Advanced Microcontroller Bus Architecture

AHB

Advanced High-performance Bus

APB

Advanced Peripheral Bus

Chapter 1

Introduction

Modern satellite missions and space exploration platforms face a critical challenge when it comes to efficient data processing. The increasing complexity of payloads and Earth observation applications has led to a noticeable size increment of data generated onboard. For this reason, the implementation of advanced data compression techniques has become essential for both data storing and transmission. Although a software version of the compression algorithm is the most portable solution, it is also not efficient. A fully-dedicated hardware implementation, on the other hand, lacks in flexibility, requiring to redesign the hardware every time a change is needed. Hardware accelerators can be a good trade-off to greatly increase the performance without adversely affecting the occupied area and flexibility. This work will compare tightly-coupled and loosely-coupled accelerators, enhancing the performance of the recommended **CCSDS 121.0** standard for lossless data compression in space applications.

1.1 Thesis objectives and organization

The thesis will be organized as follows:

- **chapter 1** is a brief introduction about the purpose and motivations of this work;
- **chapter 2** presents the background information necessary to understand what will be discussed in the following chapters. Section 2.1 introduces data compression techniques, mainly focusing on the algorithm that is the subject

of this thesis. Section 2.2 is dedicated to RISC–V architectures: the NOEL–V processor is introduced alongside Synopsys ASIP Designer, the tool that allows the implementation of the accelerators. Section 2.3 will conclude this chapter with a general presentation of different variants of hardware accelerators;

- **chapter 3** gives a more comprehensive overview of the NOEL–V processor, describing its key components and the various configurations options. Section 3.1 illustrates the ASIP Designer model of the processor;
- **chapter 4** describes the development of the tightly coupled accelerators. Section 4.1 analyses the three main areas improved by the accelerators, while section 4.2 reports how the accelerators have been integrated into the official NOEL–V processor within GRLIB.
- **chapter 5** illustrates the developed loosely coupled accelerator. Firstly, the ASIP Designer implementation is reported in section 5.1, and then its integration with the official NOEL-V processor is described in section 5.2.
- **chapter 6** reports the area and performance results of the TCAs in section 6.1, of the LCA in section 6.2 and a comparison of the two approaches in section 6.3. The proposed accelerators are then compared to the designs found in the literature in section 6.4, and some future work proposals are laid out in section 6.5;
- **chapter 7** is the conclusion of the thesis, where the final remarks are discussed and all the achieved results are summarized.

Chapter 2

Theoretical background

This chapter provides an overview of the three key aspects of this thesis: data compression algorithms, RISC-V architectures and hardware accelerators. Data compression is a technique used to encode information using less bits than the original data, in order to save storage space and transmission time. The main focus will be on the CCSDS 121.0 standard for lossless data compression. RISC–V is an open-source instruction set architecture (ISA) processors that offer flexibility and efficiency for various applications. Accelerators are hardware blocks designed to perform a specific task efficiently. They can either be small and integrated inside the core of the processor, or perform harder computational task and being connected through other interfaces. Together, these technologies play a crucial role in optimizing resource utilization in constrained environments.

2.1 Data compression

The main objective of data compression is to encode information using fewer bits, reducing the size of data files or streams to save storage space and transmission time. Depending on the technique applied, the original data can be either preserved or some information can be discarded to achieve higher compression rates, these are the cases of **lossless** and **lossy compressions**. Lossless compression algorithms play a fundamental role in space data management, providing an efficient and reliable method to reduce data size without compromising integrity in environments where computational power, storage, and bandwidth are limited.

2.1.1 CCSDS 121.0

The chosen algorithm for this work is the CCSDS 121.0 standard, specifically the B-3 version [1], regarding lossless data compression. The standard consists of two separate functional parts: the preprocessor and the Adaptive Entropy Coder. The function of the preprocessor is to decorrelate input data and reformat them into non-negative integers, applying a reversible function to each block of input samples. This produces a 'preferred' source block, characterized by its low *entropy*: a measure of the smallest average number of bits that can be used to represent each sample. This component is not essential and, depending on the implementation, can be omitted. The preprocessor adopted for this work is the **unit delay predictor** with the **prediction error mapper** described in the selected standard. The Adaptive Entropy Coder uses the Rice's coding technique [2] that chooses, among a set of code options, the best representation for an incoming block of preprocessed data samples. This is achieved by applying concurrently several compression algorithms to a block of consecutive preprocessed samples. The configuration that yields the shortest encoded length for the current section of data is selected for transmission, and a unique identifier is attached to the block to indicate to the decoder which option to use. The possible block encoding techniques are:

- **No compression**: If no other scheme offers a better compression size, the data is not compressed, but rather written as is inside the memory. In this case the compression size is $block\ size \cdot samples\ bit\ depth$.
- **Fundamental Sequence**: The most basic encoding option is a variable-length codeword (called FS codeword), which consists of m zeros followed by a one when the preprocessed sample has a value equal to m.
- **Sample Splitting**: The k_{th} split-sample option is obtained by encoding the $n - k$ MSBs of each preprocessed sample with a FS codeword, and then appending the preprocessed sample's k LSBs uncoded.
- **Second Extension**: Each pair of preprocessed samples in a J-sample block is transformed into an H sample using the following equation:

$$H_{smp} = \frac{(J_{smp\ 2} + J_{smp\ 1}) \cdot (J_{smp\ 2} + J_{smp\ 1} + 1)}{2} + J_{smp\ 1} \tag{2.1}$$

  The H samples block is then encoded using a FS codeword.
- **Zero Block**: Each sequence of consecutive All-Zeros blocks is encoded by one FS codeword composed by a one preceded by $n - 1$ zeros, where n is the

number of consecutive All-Zeros blocks.

A graphical representation of the complete algorithm chosen for the optimization is reported in Figure 2.1.

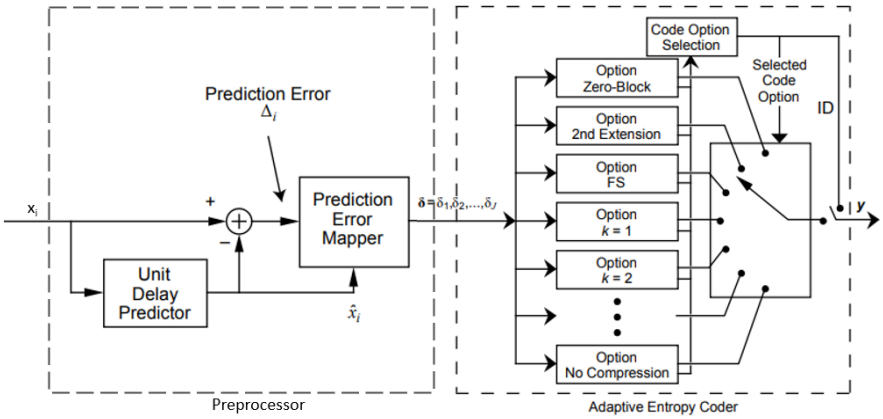


Figure 2.1: CCSDS 121.0 algorithm

2.1.2 OBPMark

The code implementation of the compression algorithm used in this work can be found in the OBPMark repository [3], an open source set of computational performance benchmarks developed specifically for spacecraft on-board data processing applications.
Aside from the employed data compression algorithm, the repostory also contain other interesting algorithm used in space applications like image and radar processing, image compressions, signal processing and machine learning.

2.2 RISC-V

RISC–V is a modern open-source instruction set architecture (ISA) originally developed as an educational and research instrument. However, given the widespread support it received, it has been quickly adopted in a large variety of applications. RISC-V architectures are characterized by 32 or 64-bit parallelism (with a 128-bit variant under development), and by a central register file with 32 fields, accessed using two read ports and one write port. Depending on the register usage, the base ISA divides the instructions into four formats:

- **R-Type**: two source registers and the destination register
- **I-Type**: one of the source registers and the destination register
- **S-Type**: only the two source registers
- **U-Type**: only the destination register

A further classification can be made according to the immediate encoding used by the instruction, explained in detail in [4].

The basic integer ISA support only basic arithmetic and boolean operations, but its capabilities can be enriched using **extensions**. They provide additional functionalities to the base processor, ranging from typical multiplication and floating point operations to more advanced operations like vector data and SIMD processing. Thanks to the open-source nature of this project, specialized instructions can be developed to enhance domain-specific applications. This procedure is called **Instruction Set Extension** (ISE) and allows the creation of a unique processor with specialized ISA, called Application Specific Instruction-set Processor (ASIP). The RISC-V based processor chosen for this thesis is the NOEL-V, that will be thoroughly described in Chapter 3.

2.2.1 Synopsys ASIP Designer

Synopsys' ASIP Designer [5] is a proprietary software that has been developed to ease the implementation of ASIP processors. The tool uses two proprietary languages to describe processor architectures:

- **nML**: this language is used to outline the instruction set architecture of the processor, using a hierarchical approach. Each instruction is described with its encoding and its behaviour individually, and operations of the same type

or that share the same functional unit are then bundled together. Every bundle is then assembled with the other ones until the complete ISA is formed. Memory resources are also described in this scope, and are fully customizable, for example in their bit-width, number of ports and access timing.

- **PDG**: this language describes the behaviour of functional units, using a C-like approach. This setup simplifies the process of creating new functions for the processor, as a C implementation of the new features can be easily transferred into PDG format.

The development flow of ASIP Designer consists on the compiler-in-the-loop approach reported in Figure 2.2. The processor model is compiled and an Instruction Set Simulator (ISS) is generated in order to emulate code running on the developed design. The ISS provides both instruction-accurate and cycle-accurate simulation, and the built-in debugger and disassembler allow for microcode stepping, which makes finding bugs and errors more convenient. Furthermore, ASIP Designer offers remarkable profiling capabilities, including the measurements of cycle count, register accesses, instructions count, and tracing. The processor can then be modified and re-compiled, updating the ISS, and this procedure can be iterated until the model performs as intended. Once a satisfactory design is achieved, the tool can generate a synthesizable HDL description of the processor either in Verilog or VHDL, which can be used for further development through other means.

2.3 Hardware accelerators

Data compression algorithms can be implemented with multiple methods. Although a software solution is the most portable, it is also not efficient, while a fully-dedicated hardware implementation lacks in flexibility, requiring to redesign the hardware every time a modification is needed. Hardware accelerators can be a good trade-off to greatly increase the performance without compromising too much area and flexibility. Hardware accelerators are usually divided in tightly-coupled and loosely-coupled, depending on their implementation.

2.3.1 Tightly-coupled accelerators

Tightly coupled accelerators (TCAs) reside within the CPU's microarchitecture, involving core and toolchain modifications. They usually need instruction set extension, meaning that the ISA of the processor is extended with new custom

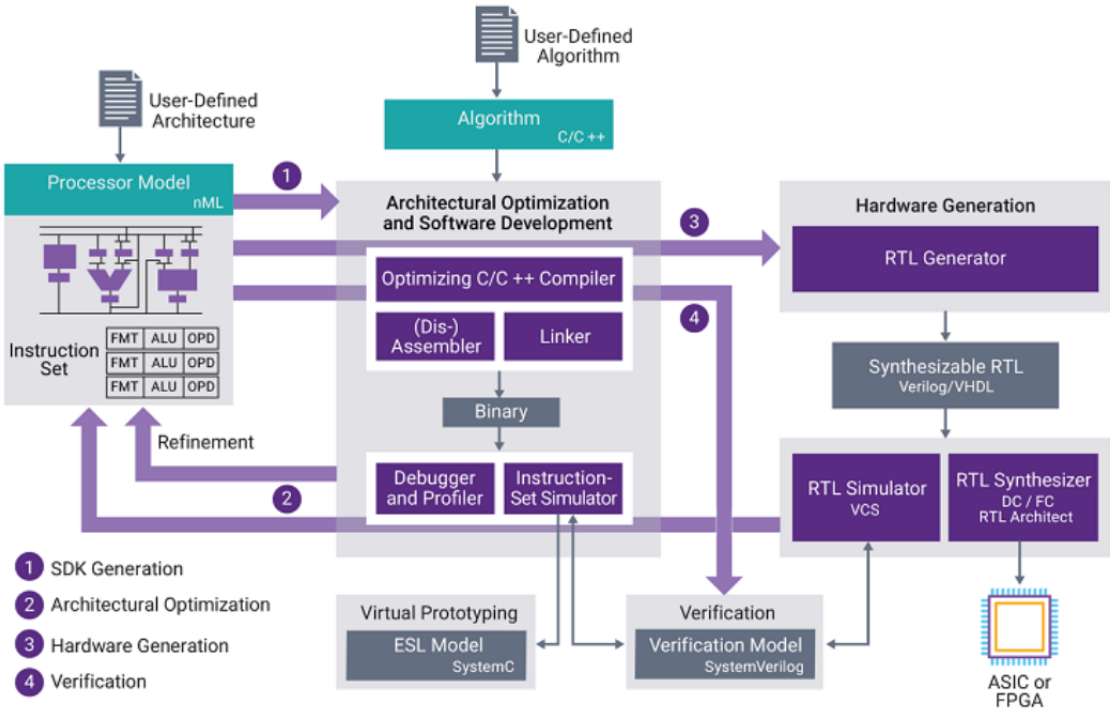


Figure 2.2: ASIP Designer's development flow.
Image source: Synopsys' ASIP Designer™ webpage [5].

instructions in order to use them. They are usually small, to keep the critical path short, and highly specialized in a single task. For this thesis, several TCAs have been developed, each targeting a specific bottleneck of the chosen algorithm.

2.3.2 Loosely-coupled accelerators

Unlike TCAs, loosely-coupled accelerators (LCAs) are typically separated from the CPU and handle compute-heavy tasks with large data sets. LCAs are usually bigger than their counterparts, as they commonly work concurrently with the processor to complete complex tasks in few clock cycles. These accelerators are usually connected via system buses and can be driven in several ways. The LCA developed for this thesis uses the memory-mapping approach, meaning that some memory locations are reserved to exchange data and give commands to the accelerator. The LCA is connected to the system bus of the NOEL-V, whose controller handles bus ownership and selects the correct peripherals depending on the given address. A more detailed explanation will be given in Chapter 3.

Chapter 3

NOEL–V

The RISC–V processor that has been chosen for this thesis is the NOEL–V [6], a new candidate in the space-graded processor industry. This field is currently led in Europe by the LEON SPARCv8 [7] due to its open source nature that offers multiple sources of intellectual properties together with radiation hardened standard products from several vendors. Nonetheless, the NOEL–V proved to be a suitable alternative for novel space-graded processor architectures, attracting a growing interest in its advancement. The open-source version of this processor is distributed by Gaisler as part of GRLIB IP library [8], under GNU GPL license and comes in many configurations, as shown in Table 3.1. Each of them is available with either 32 or 64 bit parallelism, making the NOEL–V suitable for various types of applications. The configuration adopted for this thesis is the General Purpose (GP) one, with 64-bit parallelism.

Configuration	Target	Pipeline	RISC-V extensions	MMU	PMP	Privilege modes	Example SW
HP	High-performance processing	Dual issue	IMAFDB*CH	Yes	Yes	Supervisor, User and Machine + Virtualization	Hypervisor, Linux, VxWorks
GP	General purpose processing	Dual or single issue	IMAFDB*CH	Yes	Yes	Supervisor, User and Machine + Virtualization	Hypervisor, Linux, VxWorks
GP-lite	General purpose processing, area optimized	Dual or single issue	IMAFDB*C	No	No	Supervisor, User and Machine	Linux, VxWorks
MC	Controller applications	Single issue	IMAFDB*C	No	Yes	User and Machine	RTEMS
MC-lite	Controller applications, Area Optimized	Single issue	IMA	No	No	User and Machine	RTEMS

Table 3.1: NOEL-V configurations

The NOEL–V core of the selected configuration employs a 7-stage pipeline with dual issue and separate data and instruction L1 caches. Other relevant features, also reported in Figure 3.1, are the branch prediction unit, the FPU and the late-ALU and late-branch support.

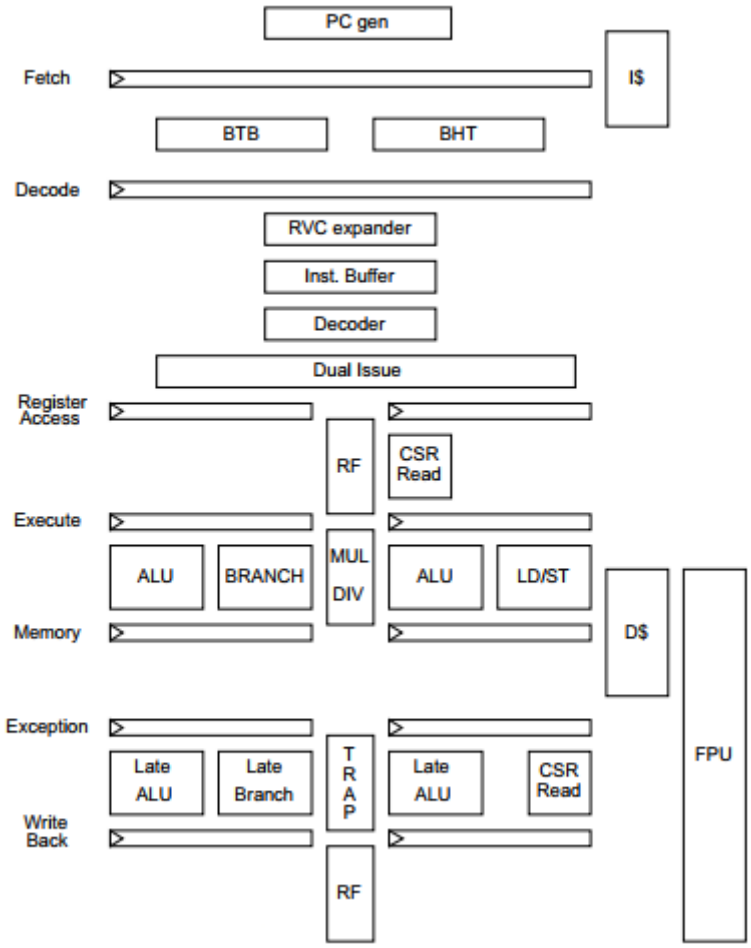


Figure 3.1: NOEL–V pipeline schematic

The core is then instantiated into a subsystem featuring a debug module, two system buses and several other peripherals, as shown in Figure 3.2. These buses use the Advanced Microcontroller Bus Architecture (AMBA) specification, which defines an on-chip communications standard for designing high-performance embedded microcontrollers [9]. The first system bus is the Advanced High-performance Bus (AHB), which is designed for high clock frequency system modules and supports the efficient connection of processors, on-chip memories and off-chip external memory

interfaces. The other bus is an Advanced Peripheral Bus (APB) that is optimized for low power consumption and reduced interface complexity to support peripheral functions. Several units are connected to this bus, like a UART unit, an Ethernet core and a GPIO module. Both these system buses and their peripherals use the AMBA Plug'n'play function described in GRLIB User's manual [10]. This feature allows the bus controller to automatically recognize the connected units and their memory space using a memory-mapped approach. The controller receives the address from the master that is currently using the bus, and selects the slave peripheral assigned to the corresponding address space. This feature allows to easily integrate other IPs into the system by simply providing their own address space. More information about this feature are reported in the GRLIB User's manual [10], while for further details about the AMBA interface specifications and timing please refer to its specification manual [9].

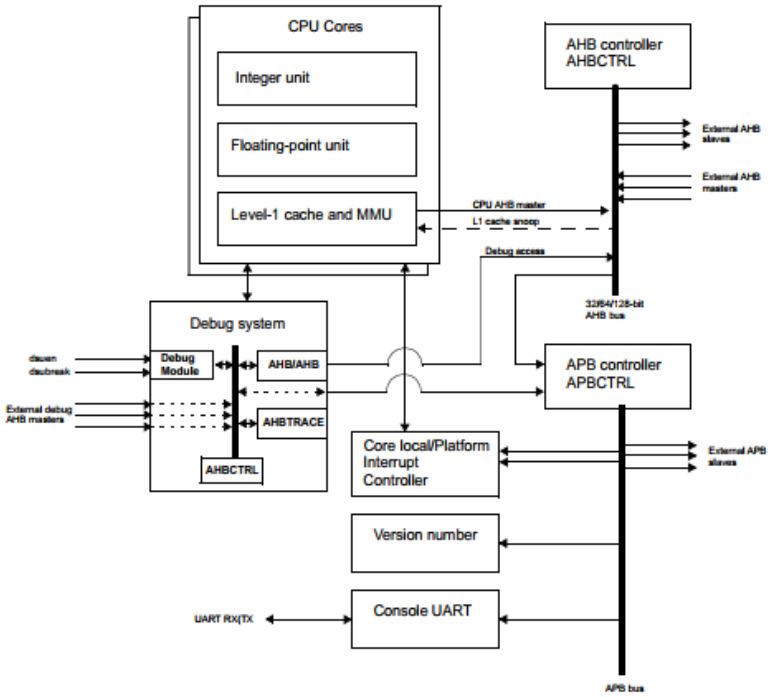


Figure 3.2: NOEL–V subsystem schematic

3.1 ASIP Designer processor model

Before starting the development of the accelerators, an ASIP Designer model of the NOEL–V has been realized. Even though it is not an exact replica, it comprises all the key features of the real processor, and therefore is a suitable environment to develop the accelerators. The tool comes with a plethora of design examples, including many RISC–V models. The one chosen as starting point for this work is the *trv64p5*, a 64-bit 5-stage pipeline RISC–V processor. This example model already contains the vast majority of the features and instructions needed to obtain an architecture close to the one of the NOEL–V. The first additions are the floating point unit and the CSR register, which have been retrieved from another example processor of ASIP Designer. Since this FPU will not be used by the chosen algorithm, no more precise details regarding its implementation will be provided. It is important to clarify that, even though the FPU is not actually used, it has been added anyway to produce a model closer to the NOEL–V and to give more significant results on area increments. The main changes that are worth looking into are the **pipeline extension** and the **dual issue** modification.

3.1.1 Extending the pipeline

The next modification made to the model was to extend the pipeline to reach the 7-stages structure of the NOEL–V. The register access (RA) and exception (XC) stages have been added to the corresponding enumerate in the parent nML description of the processor. The other stage names have also been changed to resemble the ones used by the actual NOEL–V, and all the instructions have been updated accordingly. The modified enumerate containing the new stage names is shown below:

```
...
enum stage_names {
  PF=-1, // Prefetch    -> just for information purposes
  FE,    // Inst. Fetch -> 64-bit word (2 instructions) is fetched
  DE,    // Decode      -> 2 instructions are decoded at a time
  RA,    // Reg. Access -> operands are read from RF or bypasses
  EX,    // Execute     -> ALU op. are executed, address is generated
  ME,    // Memory      -> Read data is received and store is executed
  XC,    // Exception   -> Exceptions, interrupts and late op. resolved
  WR     // Write-back  -> Results are written back to the RF
};
...
```

Next, processor resources are added for the new stages. The following modifications refer to the integer pipeline of the processor, but also the floating point pipeline must be extended in the same manner. To properly connect the new stages to the existing ones, new pipe registers have been added in the corresponding nML file:

```
// RA to EX
pipe praX1 <w64>;
pipe praX2 <w64>;
...
// XC to WR
pipe pxcX1 <w64>;
trn  txcX1 <w64>;
...
```

The starting model used to fetch the operands from the central register file inside the decode stage and pass them to the execute stage through the `pidX1` and `pidX2` pipe registers. However, in the new model, the register file is accessed in the register access stage, meaning that these two registers are no longer used. For the same reason, and also to correctly pipe data through the new exception stage, the mode rules used to read and write data to the central register file have been modified with the new pipe registers:

```
// in the read modes:
  ...
  read_value  : stage EX: praX1;
  read_action {
    stage RA: praX1 = r;
  }
  ...

//in the write modes
  ...
  write_action {
    stage EX: pexX1 = texX1;
    stage ME: pmeX1 = tmeX1 = pexX1;
    stage XC: pxcX1 = txcX1 = pmeX1;
    stage WR: r = twrX1 = pxcX1;
  }
  ...
```

All the read modes now take data from the register file in the RA stage and transfer

it to the execution stage. The write modes on the other hand use the new pipe register and transitory to correctly route results from the execution or memory stages to the write-back stage, where they are written in the central register file.

The new exception stage could cause data dependency problems if not handled carefully. If the result of an operation in the XC stage is needed as a source operand in the RA stage, a bypass is required to solve the dependency. To add it, the *hzrd.n* has been modified as shown below:

```
...
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Bypass XC to RA if offset = 3 cycles
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// cycle            0  1  2  3  4  5  6  7  8
// add x4,x5,x6     FE DE RA EX ME XC WR
// ...                 FE DE RA EX ME XC WR
// ...                    FE DE RA EX ME XC WR
// addi x3,x4,1              FE DE RA EX ME XC WR
//                                ^^ bypass
instantiate bypass_X (bypass_X_from_XC, 3, txcX1, XC);

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Bypass WR to RA if offset = 4 cycles
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// cycle            0  1  2  3  4  5  6  7  8
// add x4,x5,x6     FE DE RA EX ME XC WR
// ...                 FE DE RA EX ME XC WR
// ...                    FE DE RA EX ME XC WR
// ...                       FE DE RA EX ME XC WR
// addi x3,x4,1                 FE DE RA EX ME XC WR
//                                   ^^ bypass
instantiate bypass_X (bypass_X_from_WR, 4, twrX1, WR);
...
```

The newly instantiated bypass solves the aforementioned dependency, taking the result of the instruction from the XC stage (using the `txcX1` transitory created previously) and routing it to the EX stage. It's important to notice that also the WR to RA bypass has been modified, increasing the number in the parenthesis from 3 to 4, to take into account the new XC stage.

All the these modifications must also be performed on the floating point pipeline, which is affected by the same problems.

Since some instructions use the program counter (PC) as an operand in the EX stage, its reserved pipeline must also be extended to cover the new RA stage. The `PC_RA` register is then added to the *reg.n* file, and a register transfer is added to the instructions that need it:

```
// in the reg.n file PC_RA is added
...
reg PC_DE <addr> read(PC_DE_r) write(PC_DE_w); // PM addr. of instr @ DE
reg PC_RA <addr> read(PC_RA_r) write(PC_RA_w); // PM addr. of instr @ RA
reg PC_EX <addr> read(PC_EX_r) write(PC_EX_w); // PM addr. of instr @ EX
...

// the new register is used for example by the auipc instr. in alu.n:
...
action {
    stage DE:           PC_RA = PC_RA_w = PC_DE_r = PC_DE;
    stage RA:           PC_EX = PC_EX_w = PC_RA_r = PC_RA;
    stage EX:           aluA = PC_EX_r = PC_EX;
                        aluB = i;
    // ---
    stage EX:           aluR = add (aluA,aluB) @alu;
    // ---
    stage EX..WR:       rd = aluR`EX`;
}
...
```

The last modification needed to complete this pipeline extension regards the control flow instructions. When an indirect jump is decoded, bubbles must be inserted until the jump instruction reaches the execution stage, where the target address is calculated. With the addition of the RA stage, the number of bubbles to be inserted must be incremented by one. This can be done by modifying the `cycles(3)` property of the JALR function in the *ctrl.n*, changing it from 3 to 4.

3.1.2 Enabling dual-issue

After testing the pipeline modifications to ensure their correctness, the processor has been modified to enable two instructions to be issued at each clock cycle. To do so, the byte-addressed program memory has been modified to output two 32-bit instructions as a single 64-bit word, which is then issued and decoded. To fetch the next two instructions, the program counter is then incremented by 8 instead of 4. Next, the pipeline must be redesigned, nearly doubling the existing

resources. To accomplish that, ASIP Designer's components have been used, which are the tool's counterpart of VHDL components. It is in fact possible to use the `component` keyword to define a set of instructions and resources that can then be easily instantiated multiple times using the `instantiate` keyword. Each new instance is characterized by a unique name, and it's possible to access the internal data using the dot operator: `<comp_name>.<resource_name>`. For example, to duplicate the pipe registers and transitory to fit the new dual lane structure, the *pipe.n* file has been modified as follows:

```
component pipe_lane() {
    // RA to EX
    pipe praX1 <w64>;
    ...
    trn  twrX1 <w64>;
}

instantiate pipe_lane pl0();
instantiate pipe_lane pl1();
```

The two instances are therefore named *pl0* and *pl1*, and their elements can be accessed using the dot operator like `pl0.praX1`. The same procedure has been applied to the central register file, where the read and write ports have been duplicated. Exploiting these features, the ISA of the processor has been modified to resemble the one of the NOEL–V:

- since each lane has its own ALU unit, both the functional unit and its instructions have been included into a component and have been entirely doubled. The correct pipe registers, transitory, and central register's ports are then assigned to each instance;
- load, store, FPU and CSR operations are executed exclusively on lane 0;
- branch and jump instructions belong to the lane 1;
- the NOEL–V features a single multiplier and divider unit, but the instructions that use it can go in both lanes.

The first difference with the NOEL–V processor is found here. In a real dual issue processor, instruction in the decode stage can be halted to prevent conflicts and can be re-ordered when some instructions have special needs. This is done because the compiler that generates the code usually does not know the exact structure of

the processor on which the code will run, and instead generates only the instruction flow. The generated code is then loaded into the program memory, and it might happen that, for example, a load instruction ends up in the wrong lane. In the NOEL–V this problem is solved by swapping the instruction lanes. Also, some combinations of instructions are prohibited, for example two multiplications cannot be issued together because there is a single multiply unit. The real processor would then halt for one clock cycle the instruction that must be executed later. These two functions are hard to implement inside ASIP Designer, due to instructions being decoded and issued by different stages. The tool issues them in the stage before the decode one, i.e. the fetch stage, assigning their execution to one of the two lanes. When they reach the decode stage, their lane cannot be changed anymore, making it impossible to implement line swapping. Even though two lanes are instantiated, the tool does not allow stalling only one of them by selectively inserting bubbles, meaning that also the second aforementioned functionality cannot be implemented. There is technically one way to solve these problems, that is manipulating the tool assigning the *decode_stage* annotation to the RA stage. This will postpone the issue to the decode stage, where it should actually belong, allowing instructions to be handled freely before they are issued. The downside of this approach is that checking the constraints of the instructions would then require a manual decoding inside the processor's controller. This makes some of the tool's functionalities useless and complicates the development.

ASIP Designer offers instead the possibility to specify which instruction pairing are allowed and which instructions can go in which lane. The built-in compiler then generates the code aware of these constraints, making sure each instruction pair can be correctly executed. Exploiting this functionality, the instruction pairing have been defined in the *noelv.n* file following the constraints reported inside the NOEL–V documentation [8]:

```
// 64-bit instruction format
opn noelv (
      alu_pair
    | jump_pair
    | branch_pair
    | mul_pair
    | div_pair
    | ldst_pair
    | swbrk_pair
    | csr_pair
    | fpu_pair
```

```
  );

// ------------- jump pairs -------------
opn jump_pair(l0 : alu0.alu_nop, l1 : jump_instrs) {
     dummy_syntax : l0;
     syntax : "nop" PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}

// ------------- branch pairs -------------
opn branch_pair(l0 : branch_paired, l1 : br_instr) {
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}
opn branch_paired (
            alu0.alu_rrr_ar_instr_or_nop
          | alu0.alu_rri_ar_instr
          | alu0.alu_rri_sh_instr
          | alu0.alu_rrr_arw_instr
          | alu0.alu_rri_arw_instr
          | alu0.alu_rri_shw_instr
          | alu0.lui_instr
          | mpy0.mpy_instrs
        );

// ------------- mul pairs -------------
opn mul_pair(mul_l0_pair | mul_l1_pair);
opn mul_l0_pair(l0 : mpy0.mpy_instrs, l1 : alu1.alu_instrs) {
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}
opn mul_l1_pair(l0 : alu0.alu_instrs, l1 : mpy1.mpy_instrs) {
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}

// ------------- div pairs -------------
opn div_pair(div_l0_pair | div_l1_pair);
opn div_l0_pair(l0 : div0.div_instrs, l1 : alu1.alu_nop) {
     dummy_syntax : l1;
     syntax : l0 PADINST " : nop";
     image  : l0::"11"::l1::"11";
}
opn div_l1_pair(l0 : alu0.alu_nop, l1 : div0.div_instrs) {
```

```
    dummy_syntax : l0;
    syntax : "nop" PADINST " : " l1;
    image  : l0::"11"::l1::"11";
}

// ------------ alu pairs ------------
opn alu_pair(l0 : alu0.alu_instrs, l1 : alu1.alu_instrs) {
    syntax : l0 PADINST " : " l1;
    image  : l0::"11"::l1::"11";
}

// ------------ ldst pairs ------------
opn ldst_pair(l0 : ldst_instrs, l1 : ldst_paired) {
    syntax : l0 PADINST " : " l1;
    image  : l0::"11"::l1::"11";
}
opn ldst_paired (
    alu1.alu_instrs
  | mpy1.mpy_instrs
);

// ------------ swbrk pairs ------------
opn swbrk_pair(l0 : swbrk_instr, l1 : swbrk_instr) {
    syntax : l0 PADINST " : " l1;
    image  : l0::"11"::l1::"11";
}
// ------------ csr pairs ------------
opn csr_pair(l0 : csr_instrs, l1 : csr_paired) {
    syntax : l0 PADINST " : " l1;
    image  : l0::"11"::l1::"11";
}
opn csr_paired (
    alu1.alu_instrs
  | mpy1.mpy_instrs
);

// ------------ fpu pairs ------------
opn fpu_pair(l0 : fpu_instrs, l1 : alu1.alu_nop) {
    dummy_syntax : l1;
    syntax : l0 PADINST " : nop";
    image  : l0::"11"::l1::"11";
}
```

The last modification needed to allow for correct dual issue execution is to update the bypasses. With two lanes it might happen that results from lane 0 are needed by lane 1 and vice versa. All the possible combinations are therefore instantiated exploiting components:

```
// From lane 0 to lane 0
instantiate bypass_X_lane bypass_X_l00(regX_l0, pl0, regX_trn_l0);
// From lane 1 to lane 0
instantiate bypass_X_lane bypass_X_l01(regX_l0, pl1, regX_trn_l0);
// From lane 0 to lane 1
instantiate bypass_X_lane bypass_X_l10(regX_l1, pl0, regX_trn_l1);
// From lane 1 to lane 1
instantiate bypass_X_lane bypass_X_l11(regX_l1, pl1, regX_trn_l1);
```

The modifications have then been tested to ensure the correct behaviour of the processor.

3.1.3 Model limitations

Aside from the aforementioned constraint on lane swapping and selective bubble insertions, the obtained processor model has also some other difference with the real NOEL–V. The most noticeable one is the absence of the late ALU and late branch units, which are used by the original processor to handle conflicts. These features are not included in the tool, and implementing them by hand would require the same modifications mentioned for the lane swapping, encountering the same problems. Other more sophisticated features, like branch prediction and caches, were intentionally omitted. This decision led to a simpler model, which is faster to simulate and easier to debug, but still remains coherent with the original processor. The model keeps in fact the same architectural key elements of the NOEL–V, and ensures a fairly accurate execution flow, leading to a suitable environment to develop the desired accelerators.

Chapter 4

Tightly-coupled accelerators

With the processor model ready, the development of the tightly-coupled accelerators can start. The first thing to do is to understand what are the bottlenecks of the algorithm through a profiling of the application. The code implementation taken from the OBPMark repository needs a small modification to compile it on the new processor model. More specifically, the part of the code that measures the execution time must be commented out, since the *time.h* library is not supported. However, the absence of this feature will not matter because the speed-up will be calculated in terms of clock cycles, which are measured by ASIP Designer profiling. The application is then run for the first time, using a reference input file, with the following parameters:

- Sample's bit depth: 16
- J-block size: 64
- Reference sample interval: 4096
- Preprocessor active: true

These parameters and the input file will remain unchanged when profiling different designs to guarantee a fair and accurate comparison. Tests, on the other hand, will also be performed changing parameters and input files to cover all the possible scenarios. The first profiling of the application highlights some complications with the code:

```
   Calls  Cycles tot  Cycles tot       Function
              (func)     (%func)
-------- ----------- -----------       ----------------------
  164415   253080042      24.58%  ...  memcpy
  148307   217029535      21.08%       memset
 1029312   195455024      18.99%       writeWord
  209079    94712787       9.20%       GetSizeSampleSplitting
    ...         ...         ...          ...
```

From this excerpt of the function report, it is possible to observe that the most computationally intensive functions are the standard library functions `memcpy` and `memset`. Those functions are used to allocate and zero-initialize blocks of data in the main memory, and relaying heavily on them leads to many wasted cycles and reduces the benefits brought by the TCAs. The reason behind all these function calls can be found by looking at how the best compression scheme is found. All the different techniques are tried one after the other, saving the better one after every comparison. The information about the compressed data is stored in a struct called `FCompressedData`, which is freed when the technique is not convenient. A new struct is then instantiated for the new scheme, and this procedure is iterated until all the possible techniques have been analysed. On top of that, each structure is passed as a function argument or used as a return value several times, leading to even more clock cycles wasted by continuously storing and loading data from the stack. The solution is to define only two of these structures in the `AdaptativeEntropyEncoder` function, and to use pointers to identify the best compression method, as shown in the code excerpt reported below:

```
// file processing.c
...
inline void MIN(struct FCompressedData** best, struct FCompressedData**
↪  new_candidate){
    if((*new_candidate)->size < (*best)->size){
        struct FCompressedData* tmp1 = *best;
        *best = *new_candidate;
        *new_candidate = tmp1;
    }
}
...
    // inside AdaptativeEntropyEncoder function
    struct FCompressedData CompressedData1;
    struct FCompressedData CompressedData2;

    CompressedData1.data = (unsigned int*) malloc(sizeof(unsigned int) *
    ↪  compression_data->j_blocksize);
```

```
    CompressedData2.data = (unsigned int*) malloc(sizeof(unsigned int) *
    ↪  compression_data->j_blocksize);

    struct FCompressedData* BestCompression = &CompressedData1;
    struct FCompressedData* CompressedData_tmp = &CompressedData2;

    if(NumberOfZeros == -1){
        NoCompression(compression_data, Samples, BestCompression);
        SecondExtension(compression_data, Samples, block, step,
        ↪  CompressedData_tmp);
        MIN(&BestCompression, &CompressedData_tmp);
        FundamentalSequence(compression_data, Samples, CompressedData_tmp);
        MIN(&BestCompression, &CompressedData_tmp);
        unsigned int max_i = compression_data->n_bits - 2;
        for(int i = 1; i < max_i; ++i){
          SampleSplitting(compression_data, Samples, i, CompressedData_tmp);
          MIN(&BestCompression, &CompressedData_tmp);
        }
    }
    else{
       ZeroBlock(compression_data, NumberOfZeros, BestCompression);
    }
...
```

Reporting every modified source file in its entirety would make this document too long. For this reason, only small sections of these files will be used to highlight important modifications done to the code and to showcase how TCAs are used. At the start of the algorithm, the no-compression technique is selected as the best compression, and therefore the *BestCompression* pointer will point to its struct. The other struct is then filled with data compressed using another scheme, and the two are compared. If the one pointed by the *BestCompression* pointer is still the better alternative, nothing happens, and the other struct is overwritten with another technique. If, instead, the new scheme is more advantageous than the current best one, the pointers are swapped. The struct previously pointed by the *BestCompression* pointer is then overwritten with a new technique. This allows using pointers as function arguments instead of the whole structs, saving even more clock cycles and removing the need of allocating data every time a new scheme is analysed.

4.1 Features improved by the accelerators

The function profiling of the rewritten code yields the following bottleneck functions:

```
   Calls  Cycles tot   Cycles tot       Function
              (func)      (%func)
-------- ------------ ------------       ----------------------
 1029312    195455024       33.93%       writeWord
  209079     94712787       16.44%       GetSizeSampleSplitting
 2059226     72874215       12.65%  ...  writeValue
  209079     61188198       10.62%       SampleSplitting
       4     27546382        4.78%       preprocess_data
 1048576     25411636        4.41%       PredictorErrorMapper
    ...          ...          ...            ...
```

These functions highlight three processing steps to be optimized: storing the compressed data in the main memory, calculating the size occupied by a J-block compressed with a certain technique and the preprocessing.

4.1.1 Storing compressed data

The *writeWord*, *writeValue* and *writeChar* functions are used to write compression data into the main memory. Together, they occupy almost half the total number of clock cycles spent to run the entire application. Looking for example at the *writeWord* function, it is possible to understand why they are so demanding:

```
void writeWord(struct OutputBitStream *status, unsigned int word, int
↪  number_bits){
    for (int i = number_bits - 1; i >=0; --i){
        status->OutputBitStream[status->num_total_bytes] |= ((word >>i)
        ↪  & 0x1) << (7 - status->num_bits);
        status->num_bits++;
        // Check if the byte is complete
        if (status->num_bits >= 8){
            status->num_bits = 0;
            status->num_total_bytes++;
        }
    }
}
```

This function stores an arbitrary number of bits, shifting them by a certain amount to make sure to append them exactly at the end of the previously stored data. For this reason, data is stored **one bit at a time**, leading to long loops and many cycles wasted. The writeWordChar does exactly the same thing but with shorter

data. The writeValue function works slightly differently than the writeWord one, because it writes a series of zeros followed by a one and allows compressed data longer than 32 bits. Aside from that, the writing procedure of the data itself is done in the exact same manner as the other two functions, meaning that an accelerator could potentially speed-up all of them.

The first TCA, hereafter called *writeWord accelerator*, allows to directly write entire 32-bit words into the memory with a single instruction while ensuring correct data alignment. Even though the model allows for 64-bit memory operations, 32 was chosen to allow the use of the accelerator even on 32-bit processors, like some of the NOEL–V's configurations. The TCA features four internal registers:

1. **incW** holds the incomplete 32-bit word that still needs to be written in the memory;
2. **lenD** contain the length of the new data to append to the one stored by incW;
3. **nbit** indicates how many valid bits are present inside incW;
4. **totB** keeps track of how many bytes of compressed data have been written into the memory to ensure that new data is stored in the right location.

The accelerator is operated by five different custom primitives, which are identified by the **CUSTOM1** RISC–V opcode:

- **rst_wW**: this instruction resets the accelerator, writing zero in all its internal registers. This function must be called before using the accelerator to ensure its registers do not hold unwanted data.
- **ld_lD**: this function loads the lenD register, indicating how many bits will be attached to the incW content. Normally, an entire J–block of samples is compressed maintaining this length constant, while distinct blocks may have it different.
- **wWord**: this is the core instruction of the accelerator that uses the hardware represented in Figure 4.1 to obtain a 32-bit word of compressed data that is then written to the memory. The PDG description of the unit is reported below:

```
w32 getWordToWrite(w64 data_i, w32 data_length, w32 incW_i, w32& incW_o,
↪  w08 nbit_i, w08& nbit_o, w32 totB_i, w32& totB_o){
  uint64_t buffer = incW_i::(uint32_t)0;
  uint32_t data = ((uint32_t) data_i) & lenD_mask[data_length];
```

```
  uint8_t bits_in_buffer = nbit_i + (uint8_t) data_length;

  buffer |= (uint64_t) (((uint64_t) data) << (64 - bits_in_buffer));


  if (bits_in_buffer < 32){
    incW_o = buffer[63:32];
    nbit_o = (uint8_t) bits_in_buffer;
    totB_o = totB_i;
  }
  else{
    incW_o = buffer[31:0];
    nbit_o = bits_in_buffer - 32;
    totB_o = totB_i + 4;
  }
  return change_endian(buffer[63:32]);
}
```

The input sample is first masked, retaining only the number of bits indicated by the lenD register. The masks are stored inside a small ROM, which is read using lenD. After that, the remaining bits are left-shifted in order to append them to the end of the valid data already present in the incW register. The shifted data is then OR-ed with the incW data, obtaining a temporal 64-bit buffer, whose 32 MSBs are written in the memory. The total number of valid bits in the buffer is given by the sum of lenD and nbit. If this number is less than 32, the word is still incomplete, and the 32 MSBs of the buffer are written in the incW register. If, instead, the total number of valid bits is higher than or equal to 32, a complete word is present. The incW register is loaded with the 32 LSBs of the buffer, i.e. the new incomplete word, and the totB and nbit registers are updated. A word written in the memory corresponds to 4 bytes, meaning that totB is incremented by 4, and the new number of valid bits in the incW register is obtained by subtracting 32 from the total number of valid bits in the buffer. The big-endian compressed word obtained with this unit is then converted to little-endian format in order to store it correctly inside the little-endian data memory.
In the EX stage, the memory address is also generated, adding totB to the address passed as an argument, and the compressed data is finally stored in the data memory.

- **st_tBiW**: when all the samples have been compressed, the total number of bytes written in the memory can be retrieved with this function. The last sample written in the memory may have led to some residual bits in the incW

register that still need to be written in the memory. This function also takes care of this aspect by doing one last memory store of the content of the incW register as it is. For the same reason, the total number of bytes is calculated by adding the number of valid bytes of this last word written in the memory (obtained from the nbit register) to the content of the totB register.

- **st_nbit**: similarly to the previous instruction, this one provides how many valid bits are present in the last byte of compressed data written in memory.

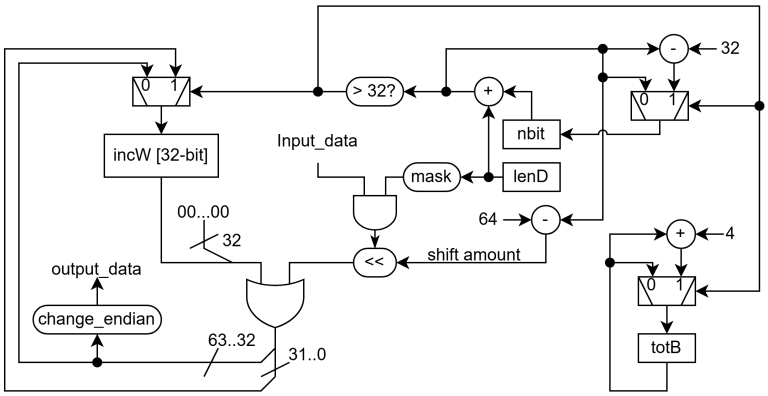


Figure 4.1: Schematic of the core function of the writeWord TCA

To use the new accelerator inside the source code, the compiler must be instructed on how to use the new primitive instructions. To do so, ASIP Designer allows to easily associate C-like functions to the primitive operations. With these custom functions, the source code can now be modified. Before the accelerator is used, a reset is performed at the beginning of the `process_benchmark` function, inside *device.c*, to clear the registers from unwanted values:

```
// inside process_benchmark function, in device.c
...
    //prepare writeWord registers
    reset_writeWord();
...
    compression_data->OutputDataBlock->num_total_bytes =
    ↪  storeLastWordNewTotBytes(
    ↪  compression_data->OutputDataBlock->OutputBitStream);
    compression_data->OutputDataBlock->num_bits = storeNewNumBits();
...
```

Next, the *output_format_utils.c* has been modified to use the accelerator during the storing procedure:

```
// beginning of output_format_utils.c
...
inline void writeWord_inline(struct OutputBitStream *status, unsigned
↪  int* data, unsigned int n_bits, unsigned int  size)
{
    unsigned int* outStreamStart = (unsigned int*)
    ↪  status->OutputBitStream;
    loadLenData(n_bits);
    for(int i = 0; i < size; ++i){
      writeWord(outStreamStart, data[i]);
    }
}

inline void writeValue_inline(struct OutputBitStream *status, unsigned
↪  int number_bits){
    unsigned int* outStreamStart = (unsigned int*)
    ↪  status->OutputBitStream;
    // number of 0 words
    unsigned int n0words = (number_bits-1)/32;
    if(n0words > 0){
      loadLenData(32);
      for(int i = 0; i < n0words; i++){
        writeWord(outStreamStart, 0);
      }
    }
    loadLenData(number_bits%32);
    writeWord(outStreamStart, 1);
}
...
```

Additionally, also the *processing.c* file has been modified, adding the writeWordChar function that uses the accelerators:

```
// beginning of output_format_utils.c
...
inline void writeChar_inline(struct OutputBitStream *status, unsigned char
↪  word, int number_bits)
{
    chess_protect_access unsigned int* outStreamStart = (unsigned int*)
    ↪  status->OutputBitStream;
    loadLenData(number_bits);
```

```
    chess_separator();
    writeWord(outStreamStart, word);
}
...
```

All these functions have been inlined to avoid calls overhead and save additional cycles. To reuse the developed accelerator also for the writeValue function, a preemptive control is needed, because, if the length of the compressed data is higher than 32 bits, more than 1 cycle is needed to store it correctly. First, the number of 0-words is calculated with the operation: $floor(\frac{number\_bits-1}{32})$. Next, the writeWord accelerator is called that many times writing 32-bit words of all zeros. The number of remaining bits to write in the memory is then calculated as $number\_bits\%32$ and is loaded into the lenD register. The accelerator is then called one more time writing the last 1 in the memory. The `chess_protect_access` attribute and the `chess_separator()` directive were used in the new writeWordChar function to ensure that the compiler does not optimize or reorder the operations. The application can now be executed both to ensure the correctness of output data and to check the benefits brought by the accelerator. A synthesis of the model to evaluate the area increment was then performed using Synopsys' Design Compiler. The results are reported in section 6.1.

4.1.2 J–block compression size

The Adaptive Entropy Coder used by the CCSDS 121.0 algorithm compares different compression techniques on each J–block, and chooses the one that gives the smallest compressed data size. The fundamental sequence, sample splitting and second extention schemes require iterating over the entire J–block to calculate the size of the data compressed using each of these techniques. The sample splitting option is particularly expensive in terms of clock cycles because the size calculation is repeated 29 times, changing the value of the k parameter from 1 to 29. The fundamental sequence is a particular instance of the sample splitting option where k is zero, while the second extension requires calculating H-samples from the input samples using Equation 2.1. Two TCAs have been realized to improve these procedures:

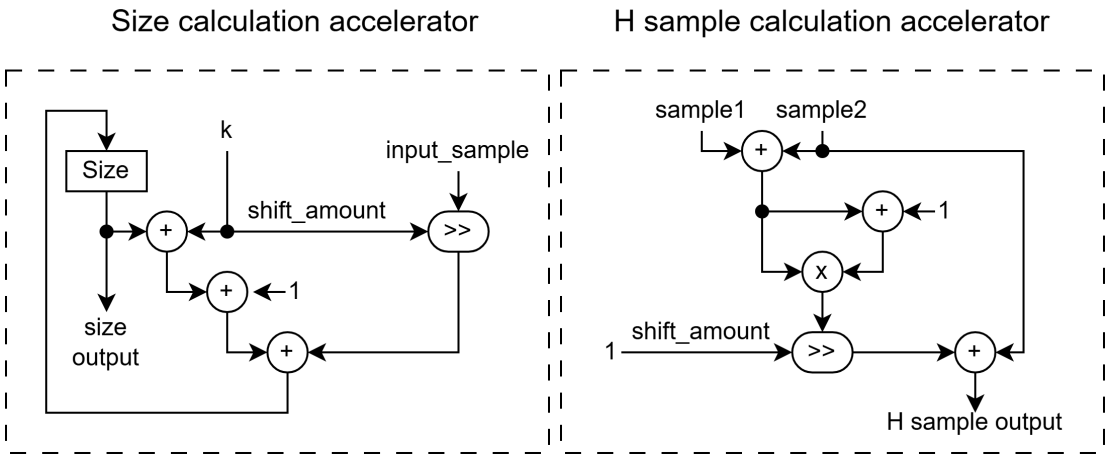
- the **size calculation** accelerator, shown in Figure 4.2a, uses an internal register and some additional hardware to speed-up every size calculation done iteratively over a block of samples. The internal *size register* is used to store

the intermediate value of the compression data size, which is calculated as $size \mathrel{+}= (input\_sample >> k) + k + 1$. The k value depends on the intruction that uses the accelerator, as will be described later. The PDG implementation of the accelerator is reported below:

```
w32 addSize(w32 old_size, w32 sample, w08 k_i){
  return (uint32_t) ((uint32_t)old_size + (uint8_t)k_i +
  ↪  (((uint32_t)sample) >> k_i[4:0]) + 1);
}
```

- the **H-sample calculation** accelerator, reported in Figure 4.2b, uses two input samples to calculate the corresponding H-sample. The PDG implementation of this unit is reported below:

```
w32 get_hSample(w32 sample1, w32 sample2){
  return (uint32_t) ( ((sample1 + sample2) * (sample1 + sample2 +
  ↪  1)) >> 1) + sample2;
}
```



(a) size calculation **(b)** H-sample calculation

Figure 4.2: J–block compression size accelerators

Five different primitives have been added to the processor to use the accelerators:

- **getHsmp**: this function is used to operate the H-sample calculation accelerator. It uses two source registers for the input samples and returns the

calculated H-sample in the destination register.

- **ld_k**: this primitive is used to load another internal register, the *k register*. It holds the value of k that can be used by the size calculation accelerator.
- **sz_wsh**: similarly to the previous wWord primitive, this instruction both operates the size calculation accelerator and performs a memory store operation. The TCA is used with the value of k read from its register, and the internal size register is updated with the newly calculated size. The input sample is then written in the memory at the address present in the other source register.
- **sz_nosh**: this function behaves exactly as the previous one, with the exception of the k value used by the accelerator being zero instead of the value read from the k register. Loading zero with ld_k and then using sz_wsh would have the same effect, but this function allows the accelerator to be used in contexts where the k is not mentioned, resulting in a clearer C code. On top of that the same hardware of the the previous primitive is used, meaning that adding this function causes a very small area overhead.
- **getSize**: this instruction is used to retrieve the size value stored in the internal register at the end of the block computation. The newly read size is stored in the central register file and the internal register is then reset to zero, ready for a new size calculation. This function can also be used to just reset the size register content, discarding the returned value, avoiding unwanted data in the register before its first usage.

Next, the custom C functions used to call the new primitives have been created. Two different functions, *addSizeHalvedSamples* and *addSizeFundamentalSequence*, are used to call the sz_nosh primitive, while the sz_wsh function can be called only from *addSizeSampleSplitting*. The source code in *processing.c* is then modified to use the newly added functions:

```
// in processing.c
...
// inside SecondExtension function
    for(unsigned int i = 0; i < HalfBlockSize; ++i){
        unsigned int HalvedSample = calculateHalvedSample(Samples[2*i],
        ↪  Samples[2*i + 1]);
        addSizeHalvedSamples(&cData_destination[i], HalvedSample);
    }
    CompressedData->size = getCalculatedSize();
...
// inside FundamentalSequence function
```

```
    for(int i = 0; i < j_blocksize; i++){
        addSizeFundamentalSequence(&cData_destination[i], Samples[i]);
    }
    CompressedData->size = getCalculatedSize();
...
// inside SampleSplitting function
    for(int i = 0; i < j_blocksize; i++){
        addSizeSampleSplitting(&cData_destination[i], Samples[i]);
    }
    CompressedData->size = getCalculatedSize();
...
```

The updated code was then executed to ensure the correctness of the output data, profiled to measure the obained speed-up, and finally synthtised to evauate the area increment.

4.1.3 Preprocessing samples

The last processing step that has been optimized is the preprocessing step. As already said in section 2.1, the preprocessor used by this algorithm is made by a simple **Unit Delay Predictor** and a **Prediction Error Mapper**. Both this units have been integrated inside a single accelerator and the entire computation is executed in a single clock cycle. The implementation of this accelerator was straightforward and the developed PDG code mirrors the C implementation almost perfectly:

```
// in ccsds121.p
...
w32 UDP_PEM(w32 new_sample, w32 PredictedValue, t1u preprocessor_active, w32
↪  x_max){
  if(preprocessor_active){
      int32_t PredictionError = new_sample - PredictedValue;
      int32_t difference = x_max - PredictedValue;
      int32_t theta = (PredictedValue < difference) ? PredictedValue :
      ↪  difference;
      int32_t PredictionErrorAbs = (PredictionError > 0) ? PredictionError :
      ↪  -PredictionError;
      int32_t PreprocessedSample;
      if(0 <= PredictionError && PredictionError <= theta) {
          PreprocessedSample = 2*PredictionError;
      }
      else if(-theta <= PredictionError && PredictionError < 0){
          PreprocessedSample = 2*PredictionErrorAbs -1;
      }
      else{
          PreprocessedSample = theta + PredictionErrorAbs;
```

```
        }
        return PreprocessedSample;
    }
    return new_sample;
}
...
```

Three additional registers have also been added. The first one stores the *x_max* value, which is calculated from the samples' bit depth. Another register holds a flag that indicates whether the preprocessor is active or not, allowing the internal hardware to be bypassed. Both these informations are obtained from the algorithm's parameters and need to be written inside the internal registers just a single time, at the start of the computation. The third register is used to store the delayed sample needed to calculate the prediction error. To load the internal registers and operate the accelerator, three custom primitives have been developed:

- **UDPclS**: this function clears the register that holds the delay sample by filling it with zeros. This is done before using the accelerator for the first time and at the end of every *reference sample interval*. This ensures that the first sample of each of those intervals passes the preprocessing stage unaltered, allowing to correctly reconstruct the initial data from the compressed data.
- **ld_UDPr**: this instruction is used to load the aforementioned registers with the preprocessor active flag and the x_max value.
- **UDPproc**: this primitive is the core of the accelerator as it actually performs the preprocessing. The input sample is provided by one source register, and it is conveyed towards the preprocessing logic and the *delayed-sample register*. If the active flag is set, the preprocessed sample is computed using the input sample and the delayed sample. If, instead, the flag is not set, no preprocessing is done and the input sample remains untouched. The result of the preprocessing is then both returned in the destination register and written directly in the main memory, at the address held by the other source register.

After the corresponding custom C functions have been added, the source code in the *device.c* and *processing.c* file has been added:

```
// in device.c
...
    // inside process_benchmark function
    UDPloadRegs(compression_data->preprocessor_active, (uint32_t)
    ↪  (((uint64_t)0x1 << compression_data->n_bits) - 1));
```

```
...

// in processing.c
...
    // inside preprocess_data function
    unsigned int r_samplesInterval = compression_data->r_samplesInterval;
    unsigned int j_blocksize = compression_data->j_blocksize;
    unsigned int step_offset = step * j_blocksize * r_samplesInterval;
    unsigned int* InputDataStart = compression_data->InputDataBlock;
    unsigned int* OutputPreprocessedStart =
    ↪  compression_data->OutputPreprocessedValue;
    // Pre-processing the input data values, precalculating ZeroBlock offsets
    for(unsigned int block = 0; block < r_samplesInterval; ++block){
        AllZerosInBlock = true;
        unsigned int* OutputPreprocessedBlock = &OutputPreprocessedStart[block
        ↪  * j_blocksize];
        unsigned int* InputDataBlock = &InputDataStart[block * j_blocksize +
        ↪  step_offset];
        // Preprocessing the samples
        for(unsigned int i = 0; i < j_blocksize; ++i){
            // print InputDataBlock with accelerator
            unsigned int preprocessed_data =
            ↪  UDPprocess(&OutputPreprocessedBlock[i], InputDataBlock[i]);
            if (preprocessed_data != 0) AllZerosInBlock = false;
        }
...
    // inside process_blocks function
    UDPclearDelayedStack();
...
```

After testing the new functionalities of the processor, a profiling of the application is performed. The function report now gives the following results:

```
   Calls  Cycles tot   Cycles tot       Function
              (func)      (%func)
--------- ----------- ------------       ----------------------
  209079    83004363       46.56%       SampleSplitting
   16083    24979943       14.01%  ...  SampleSplittingWriter
     304    21604080       12.12%       memset
       4    18109190       10.16%       preprocess_data
   16083     7430346        4.17%       SecondExtension
     ...         ...          ...            ...
```

From this it's possible to observe that now the most intensive function is the *SampleSplitting*, a function that has been already optimized. The other functions use significantly less clock cycles and they have been already optimized too. This means

that further enhancements of unoptimized functions would improve performance by less then 4%. Increasing the processor area for such a small performance increment is usually not worth it, especially in area-constrained environments. More details about the achieved performance and the area increment are reported in section 6.1.

4.1.4 Version B of the J–block TCAs

The algorithm stores the preprocessed samples inside a big vector of 32-bit elements. In a processor with a 32-bit parallelism only two elements can be computed with a single instruction, one for each source register. On a 64-bit processor however, a single source register can potentially hold two vector elements. Following this idea, a second version of the J–block accelerators has also been developed, doubling the amount of data computed with each instruction:

```
w32 addSize(w32 old_size, w64 sample_couple, w08 k_i){
  uint32_t shifted_sample1 = (uint32_t) (((uint32_t)sample_couple[31:0]) >>
  ↪  k_i[4:0]);
  uint32_t shifted_sample2 = (uint32_t) (((uint32_t)sample_couple[63:32]) >>
  ↪  k_i[4:0]);
  uint8_t k = "000"::k_i[4:0]::"0";
  return (uint32_t) (old_size + k + shifted_sample1 + shifted_sample2 + 2);
}

w64 get_hSample(w64 sample_couple1, w64 sample_couple2){
  uint32_t sample1 = sample_couple1[31:0];
  uint32_t sample2 = sample_couple1[63:32];
  uint32_t sample3 = sample_couple2[31:0];
  uint32_t sample4 = sample_couple2[63:32];
  uint32_t hsample1 = (uint32_t) ( ((sample1 + sample2) * (sample1 + sample2 +
  ↪  1)) >> 1) + sample2;
  uint32_t hsample2 = (uint32_t) ( ((sample3 + sample4) * (sample3 + sample4 +
  ↪  1)) >> 1) + sample4;
  return hsample2::hsample1;
}
```

The C code was also slightly modified, halving the loop iterations and adding casts to the vectors' pointer to ensure 64-bit memory operations. The modified code of the second extension function is reported as an example:

```
// inside SecondExtension, in processing.c
...
    const unsigned int HalfBlockSize = compression_data->j_blocksize / 2;
    unsigned long* cData_destination = (unsigned long*) CompressedData->data;
    unsigned long* Samples_align64 = (unsigned long*) Samples;
```

```
    // Halving the data using the SE Option algorithm. See:
    ↪  https://public.ccsds.org/Pubs/121x0b2ec1.pdf
    for(unsigned int i = 0; i < HalfBlockSize/2; ++i){
        unsigned long HalvedSamples =
        ↪  calculateHalvedSample(Samples_align64[2*i], Samples_align64[2*i +
        ↪  1]);
        addSizeHalvedSamples(&cData_destination[i], HalvedSamples);
    }
...
```

Even though this modification can be performed only on 64-bit processors, it allows speeding-up even more the execution of the size computation, which is still the most demanding step of the algorithm. The performance and area results obtained with this TCA are reported in section 6.1.

4.2 Integration into NOEL-V within GRLIB

Once all the TCAs have been profiled and tested extensively they were integrated inside the NOEL–V pipeline. In addition to modifying the core within the GRLIB library, this step required to re-build the NOEL–V compiler adding the new custom instructions. ASIP Designer comes with a built-in compiler, which is used also for simulation purposes, that allows code to be generated already including the new custom functions. However, since the devoleped processor model does not have all the features of the real processor, rebuilding the compiler ensures that the NOEL–V is used exploiting all its capabilities.

4.2.1 Adding instructions to the compiler

The compiler used to generate code for the NOEL–V processor, is NCC [11], a toolchain based on the GNU GCC compiler [12]. Being open-source, GCC can be freely modified, allowing custom instructions to be easily added. After downloading the NCC source files from [11] and extracting it in the *src* directory of the toolchain, the source files can be modified. To correctly add the custom instructions, only two files must be modified: **riscv-opc.h** and **riscv-opc.c**. The first one is located in the sources directory, at `<NCC_src_dir>/binutils/include/opcodes/riscv-opc.h`. Inside this file two macros are defined for each custom instruction:

- **MATCH**: is the basic instruction opcode provided initially. It consistes of the seven opcode bits, i.e. the seven LSBs, and the funct bits that are placed

in the correct position depending on the instruction encoding. Bits occupied by operands must be filled with zeros;

- **MASK**: is used to identify the position of operand bits in the instruction. The i_{th} bit in the mask is 1 if that bit is not used as an operand in the instruction, otherwise it is 0;

The instructions are then declared using the `DECLARE_INSN` function, as shown below:

```
...
#define MATCH_WWORD     0x4000102b
#define MATCH_RST_WW    0x200102b
#define MATCH_LD_LD     0x400102b
#define MATCH_ST_NBIT   0x600102b
#define MATCH_ST_TBIW   0x4800102b
#define MATCH_SZ_WSH    0x4000202b
#define MATCH_SZ_NOSH   0x4200202b
#define MATCH_GETSIZE   0x400202b
#define MATCH_GETHSMP   0x600202b
#define MATCH_LD_K      0x800202b
#define MATCH_UDPPROC   0x4000402b
#define MATCH_UDPCLS    0x200402b
#define MATCH_LD_UDPR   0x400402b

#define MASK_WWORD      0xfe007fff
#define MASK_RST_WW     0xffffffff
#define MASK_LD_LD      0xfe0fffff
#define MASK_ST_NBIT    0xfffff07f
#define MASK_ST_TBIW    0xfff0707f
#define MASK_SZ_WSH     0xfe007fff
#define MASK_SZ_NOSH    0xfe007fff
#define MASK_GETSIZE    0xfffff07f
#define MASK_GETHSMP    0xfe00707f
#define MASK_LD_K       0xfff07fff
#define MASK_UDPPROC    0xfe00707f
#define MASK_UDPCLS     0xffffffff
#define MASK_LD_UDPR    0xfe007fff
...
DECLARE_INSN(wWord  , MATCH_WWORD  , MASK_WWORD  )
DECLARE_INSN(rst_ww , MATCH_RST_WW , MASK_RST_WW )
DECLARE_INSN(ld_lD  , MATCH_LD_LD  , MASK_LD_LD  )
DECLARE_INSN(st_nbit, MATCH_ST_NBIT, MASK_ST_NBIT)
DECLARE_INSN(st_tBiW, MATCH_ST_TBIW, MASK_ST_TBIW)
```

```
DECLARE_INSN(sz_wsh , MATCH_SZ_WSH , MASK_SZ_WSH )
DECLARE_INSN(sz_nosh, MATCH_SZ_NOSH, MASK_SZ_NOSH)
DECLARE_INSN(getSize, MATCH_GETSIZE, MASK_GETSIZE)
DECLARE_INSN(getHsmp, MATCH_GETHSMP, MASK_GETHSMP)
DECLARE_INSN(ld_k   , MATCH_LD_K   , MASK_LD_K   )
DECLARE_INSN(UDPproc, MATCH_UDPPRO , MASK_UDPPRO )
DECLARE_INSN(UDPclS , MATCH_UDPCLS , MASK_UDPCLS )
DECLARE_INSN(ld_UDPr, MATCH_LD_UDPR, MASK_LD_UDPR)
...
```

The second file can be found again in the NCC sources directory, following the path `<NCC_src_dir>/binutils/opcodes/riscv-opc.c`. Inside this file, the opcodes array is defined, and every instruction is represented by a `riscv_opcode` struct whose elements represent different constraints and features of the instruction:

1. **name**: is the name that will be associated to the micro-instruction;
2. **xlen**: setting this parameter to either 32 or 64 specifies whether the instruction is targeted for only 32 or 64-bit RISC–V variants. If this value is set to 0 then the instruction will work on both variants;
3. **insn_class**: describes the class of instruction, whether it is an integer, atomic or compressed. `INSN_CLASS_I` is used for the integer class;
4. **args***: this string is used to specify the operands/register involved in the instruction. Here “d” indicates that the instruction uses a destination register. Similarly “s” is for the first source register and “t” is for the second one. Additionally, also other letters can be used to specify immediates, but the developed accelerators do not need them;
5. **match**: is the match used for the instruction
6. **mask**: is the mask used for this instruction
7. **match_func**: is a pointer to the function that will be used during compilation to detect if any instruction matches with given C operation. Here, the same function as the other opcodes was used;
8. **pinfo**: is used to describe the instruction by binary codes. Since it is not needed by the developed TCA, it is set to 0;

The new custom instructions have then been added as reported below:

```
...
{"wword"   , 0, INSN_CLASS_I, "s,t"  , MATCH_WWORD  , MASK_WWORD  , match_opcode, 0},
{"rst_ww"  , 0, INSN_CLASS_I, ""     , MATCH_RST_WW , MASK_RST_WW , match_opcode, 0},
{"ld_lend" , 0, INSN_CLASS_I, "t"    , MATCH_LD_LD  , MASK_LD_LD  , match_opcode, 0},
{"st_nbit" , 0, INSN_CLASS_I, "d"    , MATCH_ST_NBIT, MASK_ST_NBIT, match_opcode, 0},
{"st_tbiw" , 0, INSN_CLASS_I, "d,s"  , MATCH_ST_TBIW, MASK_ST_TBIW, match_opcode, 0},
{"sz_wsh"  , 0, INSN_CLASS_I, "s,t"  , MATCH_SZ_WSH , MASK_SZ_WSH , match_opcode, 0},
{"sz_nosh" , 0, INSN_CLASS_I, "s,t"  , MATCH_SZ_NOSH, MASK_SZ_NOSH, match_opcode, 0},
{"getsize" , 0, INSN_CLASS_I, "d"    , MATCH_GETSIZE, MASK_GETSIZE, match_opcode, 0},
{"gethsmp" , 0, INSN_CLASS_I, "d,s,t", MATCH_GETHSMP, MASK_GETHSMP, match_opcode, 0},
{"ld_k"    , 0, INSN_CLASS_I, "s"    , MATCH_LD_K   , MASK_LD_K   , match_opcode, 0},
{"udpproc" , 0, INSN_CLASS_I, "d,s,t", MATCH_UDPPROC, MASK_UDPPROC, match_opcode, 0},
{"udpcls"  , 0, INSN_CLASS_I, ""     , MATCH_UDPCLS , MASK_UDPCLS , match_opcode, 0},
{"ld_udpr" , 0, INSN_CLASS_I, "s,t"  , MATCH_LD_UDPR, MASK_LD_UDPR, match_opcode, 0},
...
```

After applying these changes, the compiler can be re-built. The script needed to do this is not provided directly with the NCC sources, but it can be easily obtained from BCC2 [13], the LEON compiler toolchain. After downloading its sources from [13], the required script can be found at `<BCC2_src_dir>/ubuild.sh`. The script was simply copied inside the NCC sources directory and only a single modification is necessary. The `TARGET` variable at the beginning of the script must be changed with one for the NOEL–V:

```
...
TARGET=riscv-gaisler-elf
...
```

Optionally, also the `OPT` variable can be modified, specifying the desired directory path of the new compiler. The modified script was then ran using the options `--toolchain --destination <path>`. The building process is then started and the rebuilt compiler is saved in the directory specified by `<path>`. If the `OPT` variable has been updated, the `--destination <path>` option is not needed.

4.2.2 Changes made to the NOEL–V core

Now that the compiler is ready, the NOEL–V core can be modified to integrate the TCAs. Only the first version of the accelerators has been integrated because the process is practically the same in both cases. The VHDL descriptions of all the functional units and multiplexers that are used by the accelerators have been automatically generated by ASIP Designer. All the components of each TCA are

then gathered together and wrapped by a standard interface. The three interfaces are then instantiated in a single component, which will be then used inside the NOEL–V core. The library files of the NOEL–V model that need to be modified are the integer pipeline of the processor, described in *iunv.vhd*, and the *nvsupport.vhd* file, which contains some support functions. The length of those files is considerable, therefore they are not reported in full form. However, the changes made to them will be briefly discussed hereafter, starting from the *iunv.vhd* file:

- First, the TCA library is defined and its corresponding package is included. This package contains the component instantiations needed to use the TCAs, as well as many other information needed by the ASIP Designer hdl files.

```
-- added lines 124 and 125
library ccsds121_TCA_vA;
use ccsds121_TCA_vA.TCA_vA_pkg.all;
```

- Next, the TCA component is instantiated and new signals are used to for its connections. The *comb* process sensitivity list is updated with the output signals of the accelerators:

```
-- added between lines 11725 and 11726
signal TCA_argA        : std_logic_vector(63 downto 0);
signal TCA_argB        : std_logic_vector(63 downto 0);
signal TCA_instruction : std_logic_vector(31 downto 0);
signal TCA_argR        : std_logic_vector(63 downto 0);
signal TCA_dm_addr     : std_logic_vector(31 downto 0);
signal TCA_dm_data     : std_logic_vector(63 downto 0);
signal TCA_reset       : std_logic;

-- added between lines 11727 and 11728
inst_tca: TCA_vA
port map (
argA         => TCA_argA        ,
argB         => TCA_argB        ,
instruction  => TCA_instruction ,
argR         => TCA_argR        ,
dm_addr      => TCA_dm_addr     ,
dm_data      => TCA_dm_data     ,
reset        => TCA_reset       ,
clock        => clk
);
```

```
TCA_reset       <= not arst;

-- modified line 11762
fpuo, fpuoa, tbo, hart, rstn, holdn, mmu_csr, perf, cap, imsico,
↪  TCA_argR, TCA_dm_addr, TCA_dm_data)
```

- The signals connected to the accelerator ports are driven from inside the *comb* process. The instruction word is briefly decoded inside the TCAs component to activate the correct accelerator. For this reason, the instruction word is passed to the component only if the instrucion is valid:

```
-- added between lines 13829 and 13830
TCA_argA <= ex_alu_op1(0);
TCA_argB <= ex_alu_op2(0);
-- check if accelerator instruction
if(opcode(r.e.ctrl(0).inst) = OP_CUSTOM1 and r.e.ctrl(0).valid =
↪  '1' and holdn = '1' and x_flush = '0' and not (x_branch_valid
↪  and x_branch_mispredict and r.x.lbranch) = '1') then
  -- assign instruction to "decode" which TCA to use
  TCA_instruction <= r.e.ctrl(0).inst;
  -- write TCA result
  ex_result(0) := TCA_argR;
  -- TCA result can be forwarded (r.e.ctrl(0).valid already
  ↪  checked)
  ex_result_fwd(0) := '1';
  -- TCA data for memory (address generated separately
  ex_stdata := TCA_dm_data;
else
  TCA_instruction <= (others => '0');
end if;
```

- This file must also be modified to handle the storage operations performed by the TCAs. The correct data size must be passed to the cache input and the address must be correctly generated, hence the `TCA_dm_addr` signal is passed to the address generation function (which is implemented in the *nvsupport.vhd* file). Since TCAs store 32-bit data, the *instruction_control* function must be modified to allow these back to back store operations. Additionally, line 16258 must be modified because some of the TCA instructions use both the store

functional unit and the destination register, which is normally not used for storing operations.

```
-- added between lines 4990 and 4991
if(opcode(inst_in) = OP_CUSTOM1) then
    -- for TCA store inst size = 32 bit
    dci.size := "10";
end if;
...
-- added between lines 13734 and 13735
TCA_dm_addr,
...
-- added between lines 9306 and 9307
variable sz_st_a_lt_32 : std_logic := r.a.ctrl(memory_lane).inst(13);
variable sz_st_e_lt_32 : std_logic := r.e.ctrl(memory_lane).inst(13);
...
-- added between lines 9836 and 9837
if (opcode(r.a.ctrl(memory_lane).inst) = OP_CUSTOM1 and v_fusel_eq(r, a,
↪  memory_lane, ST)) then
    sz_st_a_lt_32 := '1';
end if;
if (opcode(r.e.ctrl(memory_lane).inst) = OP_CUSTOM1) then
    sz_st_e_lt_32 := '1';
end if;
...
-- modified line 9845
sz_st_a_lt_32 = '0') or
...
-- modified line 9849
sz_st_e_lt_32 = '0') or
...
-- modified line 16258
if (v_fusel_eq(r, m, memory_lane, ST) and not (opcode(r.m.ctrl(0).inst) =
↪  OP_CUSTOM1))then
```

- To solve dependencies and allow data forwarding some additional changes are required. Normally, forwarding from store operations is not performed, but some TCA operations store data and also present results to be written in the destination register. The processor must then be able to forward this data in the same way as other results. For what concerns dependencies, TCA operations must wait if they need the result of a load instruction. They must also be considered when deciding if ALU operations, paired with TCA instructions, must be executed in the late ALUs to solve dependencies.

```
-- added between lines 4241 and 4242
variable TCA_inst : std_logic := '0';
...
```

```
-- added between lines 4268 and 4269
if(v_fusel_eq(r, same, lane, WWORD) or v_fusel_eq(r, same, lane,
↪  JBLOCK) or v_fusel_eq(r, same, lane, UDP_FU)) then
    TCA_inst := '1';
end if;
...
-- modified line 4430
v_fusel_eq(r, same, nlane, MUL or ALU or FPU or WWORD or JBLOCK or
↪  UDP_FU) and
...
-- modified line 9391
if v_fusel_eq(r, a, 0, ALU or LD or MUL or FPU or WWORD or JBLOCK
↪  or UDP_FU)  then
...
--modified line 9584
if not (i = 0 and v_fusel_eq(r, a, memory_lane, ST) and not
↪  opcode(r.a.ctrl(memory_lane).inst) = OP_CUSTOM1) then
```

- Finally, the data produced by the accelerators that must be written inside the register file is assigned to the correct variable:

```
-- added between lines 12814 and 12815
if(opcode(r.x.ctrl(0).inst) = OP_CUSTOM1) then
    x_wb_data(0) := r.x.result(0);
end if;
```

Next, the *nvsupport.vhd* file has been modified. It contains support functions called from other files, such as the *iunv.vhd* of before. The modifications are reported hereafter:

- First, the address generation function must be modified. The function call present in the *iunv.vhd* file has been modified adding the `TCA_dm_addr` signal, so the function declaration must be updated accordingly. If the instruction calling the function is then recognized as a TCA instruction, the address passed with the new signal is used as output value:

```
-- added between lines 6477 and 6478
dm_addr    : in std_logic_vector(31 downto 0);
...
-- added between lines 6524 and 6525
```

```
if(opcode(inst_in) = OP_CUSTOM1) then
    add(63 downto 32) := (others => '0');
    add(31 downto 0) := dm_addr;
    size := "10";
end if;
...
```

- Next, the TCA functional units have been added. The number of bits used to represent the functional units have been incremented by 3, since one functional unit for each type of accelerator is defined. The newly defined function unit bits are then assigned depending on the opcode of the instruction and its funct bits. For TCA instructions that perform also storing operations, the *ST* functional unit bit is set as well.

```
-- modified line 80
constant FUSELBITS   : integer := 17;
...
-- added between lines 845 and 846
constant WWORD  : fuseltype;  -- WWORD TCA
constant JBLOCK : fuseltype;  -- JBLOCK TCA
constant UDP_FU : fuseltype;  -- UDP TCA
...
-- added between lines 1902 and 1903
constant WWORD  : fuseltype := fusel(13);  -- WWORD TCA
constant JBLOCK : fuseltype := fusel(14);  -- JBLOCK TCA
constant UDP_FU : fuseltype := fusel(15);  -- UDP TCA
...
-- added between lines 6425 and 6426
when OP_CUSTOM1 =>
  case funct3 is
    -- writeWord
    when "001" =>
      if (funct7(6 downto 5) = "00") then
        fusel := WWORD;
      elsif (funct7(6 downto 5) = "01") then
        fusel := (WWORD or ST);
      end if;
    -- J-block
    when "010" =>
      if (funct7(6 downto 5) = "00") then
        fusel := JBLOCK;
      elsif (funct7(6 downto 5) = "01") then
```

```
        fusel := (JBLOCK or ST);
      end if;
    -- UDP
    when "100" =>
      if (funct7(6 downto 5) = "00") then
        fusel := UDP_FU;
      elsif (funct7(6 downto 5) = "01") then
        fusel := (UDP_FU or ST);
      end if;
    when others => null;
  end case;
```

- Some of the TCA operations use the second source register, so the function that validates this fields during instruction decoding must be updated:

```
-- added between lines 5030 and 5031
when OP_CUSTOM1 =>
```

- Next, the *dual_issue_check* function has been modified. It checks for data dependencies and conflicts, disabling instructions that do not meet the required constraints. More specifically, TCA operations should not be issued along side branches and instructions that must use lane 0. Additionally, a conflict must be raised if the TCA operation needs data from the destination register of the paired instruction. Normally in this case late ALUs will be used, but this is not an option for TCA operations.

```
-- added between lines 5556 and 5557
when OP_CUSTOM1 =>
  if (for_lane0(active, cfi_en, lane, inst_in(one)) or opcode_1 =
  ↪  OP_BRANCH) then
    conflict := '1';
  end if;
...
-- added between lines 5604 and 5605
when OP_CUSTOM1 =>
  -- TCA inst. can wrongly update internal registers otherwise
  conflict := '1';
...
-- added between lines 5842 and 5843
OP_CUSTOM1 |
```

- Finally some other minor changes are needed in the *for_lane0* function, since TCA must be executed exclusively on lane 0 and in the *exception_check* procedure, where it must be indicated that the TCA instructions do not generate exceptions.

```
-- added between lines 5109 and 5110
op = OP_CUSTOM1 or
...
-- added between lines 7201 and 7202
when OP_CUSTOM1 =>
  null;
```

After all these modifications, the entire processor with the accelerators can be tested. To use the newly added TCA operations inside the code, `asm` instructions were employed. The code is then compiled using GRLIB scripts, which also generate the memory files used to simulate the design. If the name and the correct number of arguments are used inside the `asm` function, the rebuilt compiler recognizes the instruction and correctly generates the code. For hardware simulation purposes, Mentor Graphics' Questasim was used. Due to the long simulation time typical of complex hardware systems like processors, the application code was not run in its entirety, but each accelerator underwent thorough testing individually.

Chapter 5

Loosely-coupled accelerator

This chapter will focus on the development of the loosely-couped accelerator for the CCSDS 121.0 data compression algorithm. The LCA, which is represented schematically in Figure 5.1, will feature two AHB interfaces, one master and one slave, as well as a standard memory port. The device was, in fact, designed to be connected to a dual port memory, allowing simultaneous reading and writing operations. The ASIP Designer model is described in depth in section 5.1, while the integration within GRLIB is discussed in section 5.2.

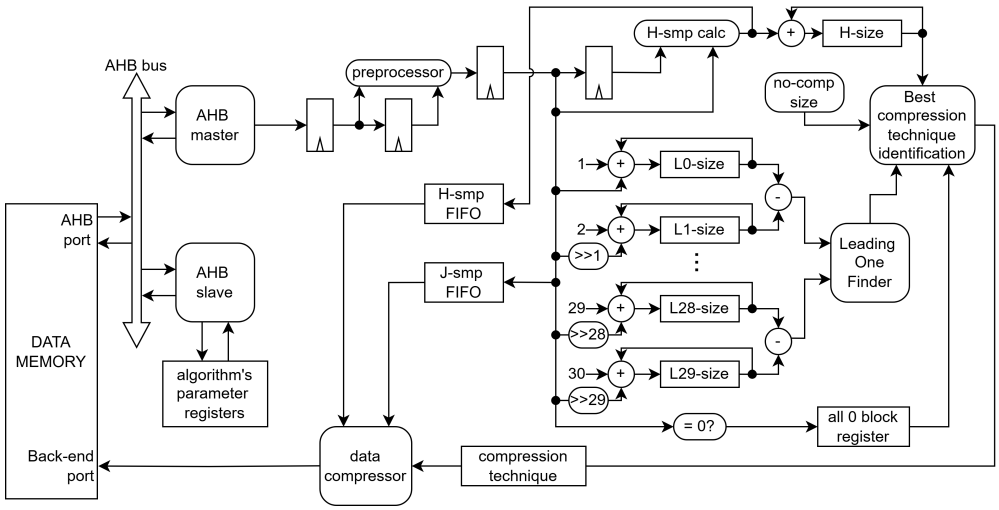


Figure 5.1: Schematic representation of the developed LCA

5.1 ASIP Designer implementation

The accelerator has been developed using ASIP Designer's **io_interface** feature. This allows creating PDG units that can work independently of the processor. Registers, memories and transitory can be declared like in nML files, but C like functions and data-types can be used as well. An io_interface is composed of:

- **External interfaces**: that are used as connections with the processor or with other io_interfaces.
- **Local storages**: are internal register and memories used inside the interface.
- **process_result process**: is a C-like function that drives the external interfaces. The output values are taken either from transitories or registers.
- **process_request process**: reads from the external interfaces and does the main data elaboration.
- **dbg_access function**: is used to define the behaviour of the interface during debug.

Once the unit is developed, it can be easily included in the main processor's PDG description.

5.1.1 Interface and memory model

The *dm.p* file of the processor is an io_interface that implements the actual data memory of the model. Since the LCA is a memory-mapped accelerator, it has been connected to this interface. The memory implementation was therefore modified to correctly route data to the actual memory or the accelerator, depending on the received address. First, the memory was modified, adding the second port, and the accelerator's input and output ports were added.

The slave interface of the LCA is used to receive commands and algorithm parameters from the processor. When a store operation is performed, if the provided address falls in the LCA's address range, data is forwarded to the accelerator's interface instead of the memory. Registers are used to capture the store operation performed by the processor and delay the data by one clock cycle to satisfy the AHB protocol timing. Load operations are treated similarly, with the exception that they already satisfy the AHB timing requirements. If a load instruction with an LCA address reaches the memory io_interface, it is redirected towards the LCA.

On the next clock cycle, the accelerator will provide the data to the interface, which will in turn forward it to the processor. If an LCA load is requested while the accelerator is busy, the entire processor, except the LCA, is stalled using the `insert_wait_this_cycle()` Chess function. Since the processor is frozen, for every successive clock cycle it will try to perform the load operation until the LCA stops being busy.

The master interface of the accelerator is used to read input samples directly from the memory during data compression. When a master wants to use the bus, it must assert its *bus_request* signal. The bus controller then grants the bus to that master, which can then start using it. To implement this feature, an additional register used to keep track of who controls the memory is added to the io_interface. Normally, the control is granted to the processor, but when the LCA computation starts, the memory control passes to the accelerator that starts reading samples through its AHB master interface. If the processor then tries to use the memory, it is again stalled using the same function discussed before. During sample processing, the LCA also stores the compressed data using the second memory port, allowing concurrent read/write operations. After the end of the computation, the bus is freed and the processor can take back the control over the memory. The full PDG implementation of the memory io_interface is reported in Appendix B.

5.1.2 Compression parameters and commands

After finishing the *dm.p* modification, the LCA was developed. After the definition of the required input and output interface ports, the registers needed to store the algorithm parameters have been added. These parameters are passed to the accelerator by performing AHB write operations, which are handled inside the `process_request` process. The input samples and the compressed data start addresses are also stored to allow direct read and write operations. Start and reset commands are also received in the same way:

```
ahb_slv_op_ncc_ff = 0;
ahb_pipe = 0;
if ((hsel_slv_i == "1") && (htrans_slv_i == "10") && (hsize_slv_i ==
↪  "010") && hready_slv_i && !acc_bsy){
  ahb_pipe = haddr_slv_i;
  ahb_slv_wr_ncc_ff = hwrite_slv_i;
  ahb_slv_op_ncc_ff = 1;
}
```

```
if(ahb_slv_op_ncc_ff && ahb_slv_wr_ncc_ff){
  switch (ahb_pipe){
    // load configuration registers
    case NBITS_ADDR:
      n_bits = hwdata_slv_i[5:0];
      break;
    case JSIZE_ADDR:
      j_blocksize = hwdata_slv_i[6:0];
      break;
    case RSINT_ADDR:
      r_samplesInterval = hwdata_slv_i[12:0];
      break;
    case PPACT_ADDR:
      preprocessor_active = hwdata_slv_i[0];
      break;
    case TOT_SMP_ADDR:
      tot_samples = hwdata_slv_i;
      break;
    case SAMPLE_ADDR:
      samples_address = hwdata_slv_i;
      break;
    case CDATA_ADDR:
      cdata_address = hwdata_slv_i;
      break;
    // accelerator operations
    case START_COMP_ADDR:
      if (hwdata_slv_i == 1){
        ahb_req_mem = 1;
        start_comp_ff = 1;
      }
      break;
    case RST_ADDR:
      if (hwdata_slv_i == 1){
        smp_in_del = 0;
        totB = 0;
        n_bits = 0;
        cnt_j_wr = 0;
        cnt_j_rd = 0;
        cnt_smp = 0;
        incW = 0;
        size_se = 0;
        v30_uint32_t size_k_tmp;
        for(int k = 0; k<30; k++){
          size_k_tmp[k] = 0;
```

```
        }
        size_k = size_k_tmp;
      }
      break;
    default:
      break;
  }
}
```

The reset command initializes all the internal registers, while the start command starts the computation by requesting the bus ownership. The processing is divided into several steps, exploiting a pipeline architecture where each step is enabled by a flip-flop. Each section can enable the following one, and thanks to the pipeline structure, many sections can work concurrently. For example, when the start command is received, the `start_comp_ff` is set, meaning that the next clock cycle the following if statement is executed:

```
if(start_comp_ff){
  // accelerator becomes busy
  acc_bsy = 1;

  if(hgrant_mst_i == "1"){
    // start a new read and increment the sample counter
    start_read = 1;
    cnt_smp += 1;

    // process the first read and start the second one
    start_rd_ff = 1;
    rd_smp_ff = 1;

    // reset register
    start_comp_ff = 0;
  }
  else{
    // if the bus is not granted keep asking for it
    ahb_req_mem = 1;
  }
}
```

Here, the accelerator checks if it has the bus ownership, and if it does, it proceeds to the next processing step and starts reading the input samples. If it doesn't have

the bus ownership, it keeps requesting it.

5.1.3 Preprocessing

Once the bus has been granted to the accelerator, the computation starts by reading the input samples from the memory. It is done through the AHB master interface of the LCA that sends read commands to the AHB slave memory interface. If the `start_read` is set, read commands are sent from the `process_result` function:

```
if(start_read){
  haddr_mst_o = samples_address + cnt_smp::"00";
  htrans_mst_o = "10";
  hsize_mst_o = "010";
  hlock_mst_o = "1";
}
```

During the entire processing, the bus is locked using the *hlock* signal, preventing the bus ownership from changing. Once a sample arrives on the bus, it is stored inside a pipe register, ready to be preprocessed. During this phase, the same preprocessing unit developed as TCA has been used:

```
if(pproc_smp_ff){
  // preprocessor section
  ppsmp = UDP_PEM(smp_in, smp_in_del, preprocessor_active,
  ↪  w32_mask[n_bits]);
  smp_in_del = smp_in;

  // proceed to size calculation and start new read
  size_calc_ff = 1;

  // keep the control of the bus with locked idle transmission when
  ↪  read is ended
  locked_idle = !start_rd_ff;

  // update registers
  pproc_smp_ff = rd_smp_ff;
}
```

The preprocessed sample is then stored inside a pipe register to discontinue the combinational path, and the size calculation step is started.
If the preprocessor is deactivated, the developed LCA implements only the block

entropy coding, and can therefore be used with other algorithms, like the **CCSDS 123** [14, 15]. This multispectral and hyperspectral image compression algorithm can use the same decoder as the CCSDS 121 standard, as showed by the SHyLoC implementation in [16].

5.1.4 Size calculation

During this step, the preprocessed sample is added to the fundamental sequence and all the sample splitting sizes concurrently, and the results are saved in the corresponding registers. On top of that, an H-sample is calculated every two samples using the current sample and the preceding one, which was saved in a register. The obtained H-sample is then added to the second extension size register. The preprocessed sample is then stored in the J register, the FIFO-like structure represented in Figure 5.2 which is built to hold the preprocessed samples until the writing operation is executed, adapting to every J-block size.

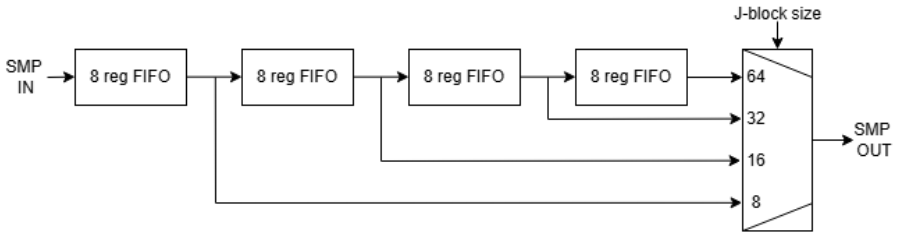


Figure 5.2: FIFO structure used for the J registers

To write and read samples from the FIFO structure, two auxillary PDG functions have been added:

```
void wr_Jreg(uint32_t smp_to_write){
  v64_uint32_t regJ_tmp;
  regJ_tmp[0] = smp_to_write;
  for(int i = 0; i < 63; i++){
    regJ_tmp[i+1] = regJ[i];
  }
  regJ = regJ_tmp;
}

uint32_t rd_Jreg(){
  uint32_t j_out;
  switch(j_blocksize){
    case 8:
```

```
      j_out = regJ[7];
      break;
    case 16:
      j_out = regJ[15];
      break;
    case 32:
      j_out = regJ[31];
      break;
    case 64:
      j_out = regJ[63];
      break;
  }
  return j_out;
}
```

A similar, but smaller, unit is also instantiated for the H register holding the H-samples.
During this step, which is iterated over all the samples of a J-block, a 1 bit register is used to check whether the block contains all zeros, allowing zero-block identification during the next section.

5.1.5 Compression technique identification

Once a J-block has been analysed entirely, the best compression technique can be found. This process happens in a single clock cycle, where all the compression technique sizes are compared and the best one is chosen. First, the best sample splitting candidate is found:

```
// find leading one
v15_uint1_t diff_signs_vector;
for(int k = 0; k < 15; k++){
  uint32_t diff = size_k[2*k] - size_k[2*k+1];
  diff_signs_vector[k] = diff[31];
}
uint4_t leading1 = find_leading_1(diff_signs_vector);

// find best sample-splitting (0 == fundamental sequence)
uint5_t candidate1 = (uint5_t) leading1::"0";
uint5_t candidate2 = ((uint5_t) leading1::"0") - 1;
uint5_t candidate3 = candidate2[4:1]::"0";
uint5_t index_smallest;
```

```
if(leading1 == 0){
  index_smallest = 0;
}
else if((size_k[candidate3] <= size_k[candidate2]) &&
↪  (size_k[candidate3] <= size_k[candidate1])){
  index_smallest = candidate3;
}
else if(size_k[candidate2] <= size_k[candidate1]){
  index_smallest = candidate2;
}
else{
  index_smallest = candidate1;
}
uint5_t index_size_lk = (index_smallest <= n_bits-3) ? index_smallest :
↪  n_bits-3;
size_lk = size_k[index_size_lk];
```

The registers that hold the sizes of all the different sample splitting options are ordered with k growing from 0 to 29. This means that register 0 holds the size for the sample splitting option with *k=0*, i.e. the fundamental sequence scheme, register 1 holds the size of the option with *k=1* and so on. With this arrangement, the resulting sizes will either present "V-shaped" or monotonous trends. This ensures that it is always possible to find the smallest value without actually comparing every size one by one.
To do so, from each size the next one is subtracted, and the MSBs of the 15 results are gathered together. The resulting vector will have zeros until a certain point, meaning that the corresponding couples present the k_{th} element bigger or equal than the k_{th+1} one. The remaining bits will be all ones, meaning that here the trend is monotonically increasing. With a leading one detector, it is therefore possible to identify the first "1" of the vector, which reduces the number of possible candidates to three, as shown in Figure 5.3. The three candidate sizes are then compared and the smallest one is chosen.

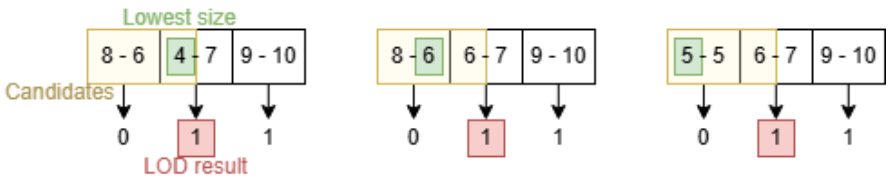


Figure 5.3: Smallest sample splitting size cases

After that, the best compression method is then found by comparing the remaining sizes and the zero-block flag:

```
// no copression size
uint32_t size_nc = j_blocksize * n_bits;

// find best compression method
if(all_0_block_ff){
  // for ZB J and H mem are not needed
  // if not ended samples start a new read
  start_rd_ff = (cnt_smp != tot_samples);

  cid_int_tmp = ZB_CID_INT;
  compression_identifier = 0;
}
else if (size_nc <= size_se && size_nc <= size_lk){
  // writing has same timing than reading, since writing starts earlier
  // there is no risk of loosing data not yet compressed
  start_rd_ff = (cnt_smp != tot_samples);

  nZeros = 0;
  cid_int_tmp = NC_CID_INT;
  compression_identifier = cid_nc_tmp;
}
else if(size_se < size_nc && size_se <= size_lk){
  // writing has same timing than reading, since writing starts earlier
  // there is no risk of loosing data not yet compressed
  start_rd_ff = (cnt_smp != tot_samples);

  nZeros = 0;
  cid_int_tmp = SE_CID_INT;
  compression_identifier = 1;
}
else{
  // if FS start reading, if SS wait to finish phase 1
  start_rd_ff = (index_size_lk == FS_CID_INT) && (cnt_smp !=
  ↪  tot_samples);

  nZeros = 0;
  cid_int_tmp = index_size_lk;
  compression_identifier = index_size_lk+1;
}
```

If the chosen technique is not the Sample Splitting option, the reading procedure

of a new J-block can start. The new samples are stored in the internal register with the same timing as they are read for the compression, so no data is lost. The Sample Splitting option, on the other hand, requires more cycles to write the data, because the J-block must be read two times. This means that the samples acquisition is stopped until this writing operation ends.
With the best compression technique now chosen, the last operation performed during this step is to store the compression technique identifier in the memory:

```
uint6_t cid_int_tmp;
uint6_t cid_nc_tmp;
uint6_t compression_identifier;
uint3_t compression_identifier_size;

if (n_bits < 3){
  compression_identifier_size = 1;
  cid_nc_tmp = 0x1;
}
else if (n_bits < 5){
  compression_identifier_size = 2;
  cid_nc_tmp = 0x3;
}
else if (n_bits <= 8){
  compression_identifier_size = 3;
  cid_nc_tmp = 0x7;
}
else if (n_bits <= 16){
  compression_identifier_size = 4;
  cid_nc_tmp = 0xF;
}
else{
  compression_identifier_size = 5;
  cid_nc_tmp = 0x1F;
}
...
// best compression technique identification
...
cid_int = cid_int_tmp;
// If the selected technique is ZB or the SE, the
↪  compression_identifier_size +1
if(cid_int_tmp == ZB_CID_INT || cid_int_tmp == SE_CID_INT){
  compression_identifier_size += 1;
}
```

```
// write compression identifier in memory
uint32_t tmp_cdata = getWordToWrite(compression_identifier,
↪  compression_identifier_size);
cdata_to_wr_tmp = change_endian(tmp_cdata);
//ahb_pipe = change_endian(cdata_to_wr);
start_write = 1;
```

5.1.6 Data compression

During this last step, data is compressed using the technique chosen previously. The samples stored inside the J register, or in the H register depending on the compression scheme, are read one by one and stored with the selected technique. The complete PDG implementation of each procedure is fairly straightforward and can be seen in the Appendix B. It is worth mentioning that the hardware unit developed for the writeWord accelerator has been used also here, as well as for the technique identifier storing operation, to correctly align the compressed data. The actual store commands are performed inside the `process_result` process, through the back-end port of the memory:

```
if(start_write){
  wrport_addr_acc_o = totB[31:2]::"00" + cdata_address;
  wrport_datain_acc_o = cdata_to_wr;
  wrport_en_acc_o = "1";
  wrport_write_acc_o = "1";
}
```

After all the J-blocks have been processed, one last cycle is needed to store the remaining bits that may not have been written yet. The total number of bytes written in the memory is therefore updated and the accelerator becomes not busy. This last clock cycle is also used to guarantee the recommended idle cycle at the end of the locked cycle.

```
if(end_wr_cdata_ff){
  // this also works as the recommended
  // idle cycle after end of locked transmission

  // write the remaining incW content in memory and update totB
  cdata_to_wr_tmp = change_endian(incW);
  start_write = 1;
```

```
  totB += nbit[4:3];

  // signals the end of the processing
  acc_bsy = 0;

  // reset register
  end_wr_cdata_ff = 0;
}
```

A known limitation of the proposed design is encountered when the chosen compression technique is the Sample Splitting scheme. This scheme requires the same J-block to be read two times, forcing to stop the sample acquisition until the block has been analysed for the first time. If this is not done, the new preprocessed samples of the new block would overwrite the previous ones, resulting in a wrong encoding. This issue obviously limits the throughput when this technique is applied, as discussed in the following Chapter.
Now that the model is ready, the C source files must be modified accordingly. To communicate with the accelerator, memory operations with the correct address must be performed. The code is therefore modified by adding the necessary store operations before the start of the computation.

```
// process_benchmark inside device.c is modified as follows
unsigned int* input_data = compression_data->InputDataBlock;
unsigned int* output_data = (unsigned int*)
↪  compression_data->OutputDataBlock->OutputBitStream;
unsigned int j_blocksize = compression_data->j_blocksize;
unsigned int r_samplesInterval = compression_data->r_samplesInterval;
unsigned int total_samples = compression_data->TotalSamplesStep;
unsigned int tot_byte = 0;

// algorithm parameters
* (unsigned int*) NBITS_ADDR = compression_data->n_bits;
* (unsigned int*) JSIZE_ADDR = j_blocksize;
* (unsigned int*) RSINT_ADDR = r_samplesInterval;
* (unsigned int*) PPACT_ADDR = compression_data->preprocessor_active;
* (unsigned int*) TOT_SMP_ADDR = total_samples;

// input samples and output compressed data addresses
* (unsigned int*) CDATA_ADDR = (unsigned int) output_data;
* (unsigned int*) SAMPLE_ADDR = (unsigned int) input_data;
```

```
// start computation
* (unsigned int*) START_COMP_ADDR = 1;

// store resulting number of bits in memory
compression_data->OutputDataBlock->num_total_bytes = *(unsigned int*)
↪  TOTB_ADDR;
compression_data->OutputDataBlock->num_bits = *(unsigned int*)
↪  SPARE_BITS_ADDR;
```

First, the algorithm parameters are loaded into the corresponding accelerator's registers, and the computation is then started. The processor will wait for the accelerator to finish the computation, and the resulting number of bytes written is read from the LCA.
Finally, the accelerator was thoroughly tested and profiled, obtaining the results reported in Section 6.2.

5.2 Integration into NOEL–V within GRLIB

The finished accelerator was then integrated with the NOEL–V processor of the GRLIB IP library. This process was much easier than the TCAs integration, thanks to the AHB plug&play feature of the library. However, before discussing the connection of the actual accelerator, the employed memory model is examined.

5.2.1 Memory model

The memory model used as a reference is the **AHBDPRAM**, a dual port synchronous RAM with an AHB interface on one port. The memory comes only as a synthesizable component with a fixed data width of 32 bits, and therefore a more fitting simulation model was derived.
The majority of the simulation model was ported from the *AHBRAM_SIM* model, which already handles AHB commands and single-port memory operations with variable bit lengths. On top of that, another process for the back-end memory port was added:

```
-- BACK-END port process
RamBEProc: process(clk) is
begin
  if rising_edge(clk) then
    if enable = '1' then
```

```
      if conv_integer(write) > 0 then
        for i in 0 to dw/8-1 loop
          if (write(i) = '1') then
            ram(to_integer(unsigned(be_ramaddr)))(i*8+7 downto i*8) :=
            ↪  be_wrdata(i*8+7 downto i*8);
          end if;
        end loop;
      end if;
    end if;
    be_read_address <= be_ramaddr;
  end if;
  be_rddata <= ram(to_integer(unsigned(be_read_address)));
end process RamBEProc;
```

The new memory can now simulate correctly operations performed on both ports, and can be adapted to different data widths. Therefore, it has been instantiated inside the `noelvmp.vhd` file, substituting the previously used memory.

```
-- new dual-port memory model instantiation
mig_ahbram: ahbdpram_sim
generic map (
  hindex    => 0,
  haddr     => L2C_HADDR,
  hmask     => L2C_HMASK,
  tech      => CFG_MEMTECH,
  abits     => log2(1024) + 8 - log2(AHBDW/32),
  bytewrite => 1,
  cacheable => 0,
  maccsz    => AHBDW,
  fname     => ramfile
)
port map (
  rst     => rstn,
  clk     => clkm,
  ahbsi   => mem_ahbsi0,
  ahbso   => mem_ahbso0,
  clkdp   => clkm,
  address => bemp_address,
  datain  => bemp_datain ,
  dataout => bemp_dataout,
  enable  => bemp_enable ,
  write   => bemp_wr
);
```

5.2.2 Connecting the accelerator

With the new memory model in place, the LCA can be finally connected. A wrapper was used to adapt the interface of the accelerator component obtained with ASIP Designer to the other interfaces:

```
entity LCA is
  generic (
    mst_hindex : integer := 0;
    slv_hindex : integer := 0;
    slv_haddr  : integer := 16#ffa#;
    slv_hmask  : integer := 16#fff#;
    port_width : integer := 32);
  port (
    rst : in std_ulogic;
    clk : in std_ulogic;
    -- AHB slave interface
    ahbsi : in ahb_slv_in_type;
    ahbso : out ahb_slv_out_type;
    -- AHB master interface
    ahbmi : in ahb_mst_in_type;
    ahbmo : out ahb_mst_out_type;
    -- Beck-end memory port interface
    bemp_din: out std_logic_vector(port_width-1 downto 0);
    bemp_addr: out std_logic_vector(31 downto 0);
    bemp_en:  out std_logic;
    bemp_wr:  out std_logic_vector(port_width/8-1 downto 0));
end;
```

The AHB configuration of the accelerator must be declared inside the `cfgmap.vhd` file in the design directory. Indexes are assigned to the new master and slave modules, and the slave's address range is chosen among the available ones.
The accelerator was then instantiated inside the `noelvmp.vhd` file, connecting it to the AHB bus and the AHBDPRAM memory back-end port, as shown in Figure 5.4. After updating the total number of the AHB modules using the bus, no other modifications are needed, and the design can be simulated. This time the compiler does not need any changes, because the accelerator is operated with standard memory operations. The design was finally simulated using GRLIB automatically generated scripts and QuestaSim. Due to the long simulation time, the entire application code was not entirely simulated, but the accelerator was thoroughly tested, varying all the algorithm parameters and test samples.

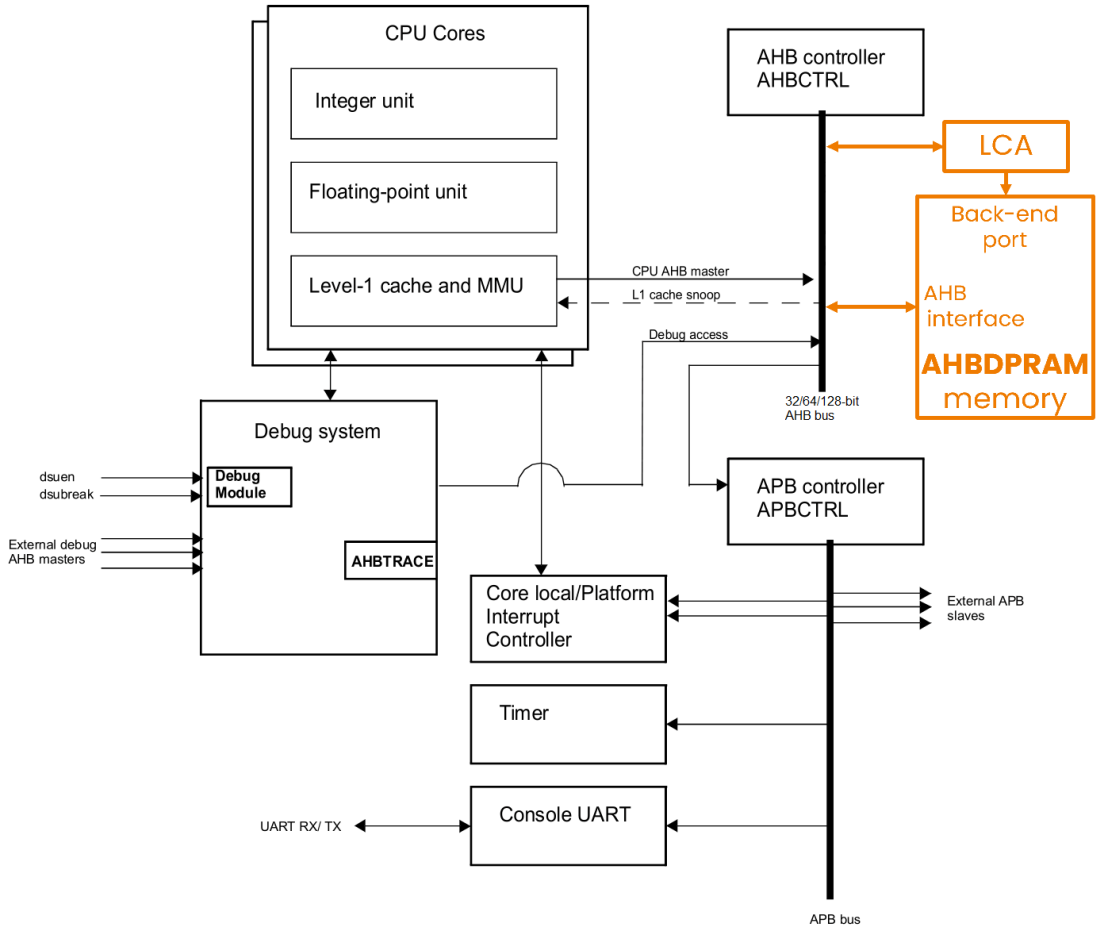


Figure 5.4: Representation of the LCA integrated inside the NOEL–V subsystem

Chapter 6

Results and comparisons

In this chapter, results obtained profiling and synthesizing every design are analysed and compared. TCA results are reported in Section 6.1, while the LCA is reviewed in Section 6.2. The two acceleration approaches are compared in Section 6.3, and the developed designs are also compared with the ones found in the literature in Section 6.4. Finally, in Section 6.5, some future work possibilities are briefly discussed.

6.1 Tightly-coupled accelerators

After implementing each optimization, the algorithm has been profiled to check the performance increment. The HDL description of the processor automatically provided by ASIP Designer was then synthesised using Synopsys' Design. The ASIC implementation has been chosen because TCAs need to be as close to the processor's pipeline as possible to reduce delays. ASIC technologies also ensure high energy efficiency and reliability, which are critical factors in space applications. The library used for the synthesis is the TSMC 65 nm low power library.
The maximum frequency of the design is then obtained from the synthesis results, and the throughput can be calculated as:

$$TH = \#samples \cdot samples\ bit\ depth \cdot \frac{f_{max}}{CC} \tag{6.1}$$

To finally compare the effectiveness of each optimization, the throughput to area

ratio (TAR) has been calculated.
The results obtained for the developed accelerators are reported in Table 6.1.

Metrics	TCA variant A: 1 sample/inst (step-by-step)					TCA variant B: 2 samples/inst
	Reference	rewriting C	writeWord	J-block	Preprocessor	
CC	1,029,520,644	576,095,801	283,249,960	203,407,562	153,878,674	107,093,829
CC reduction VS Reference	0.0%	44.0%	72.5%	80.2%	85.1%	89.6%
Speed-up VS Reference	1.00	1.79	3.63	5.06	6.69	9.61
CC reduction VS rewriting C	-	0	50.8%	64.7%	73.3%	81.4%
Speed-up VS rewriting C	-	1	2.03	2.83	3.74	5.38
ASIC synthesis technology	l.p. 65 nm	l.p. 65 nm	l.p. 65 nm	l.p. 65 nm	l.p. 65 nm	l.p. 65 nm
Max Frequency [MHz]	500	500	500	500	500	500
Total Core Area [μm^2]	168,478	168,478	171,657	181,103	184,244	192,104
Area Increment	0%	0%	1.9%	7.5%	9.4%	14.0%
Throughput at f_{max} [Gb/s]	0.02	0.03	0.06	0.08	0.11	0.16
TAR	96.73	172.86	345.06	455.44	591.76	815.49

Table 6.1: TCA performance and area results

From the table, it is possible to observe that rewriting the C code already saves 44% of the required clock cycles, highlighting the importance of this step. To understand the performance increment brought by each accelerator, the clock cycle reduction and the speed-up factor have also been calculated with respect to the rewritten code. The writeWord accelerator is the one that brings more benefits, speeding up the computation by more than 50% with an area increment of less than 2%. With the other two TCAs, the performance are enhanced up to 73.3% for the A version and to 81.4% for the B version. The former increases the area by less than 10%, while the latter increases it by 14%. The developed TCAs do not introduce any new critical paths, therefore the maximum operating frequency stayed the same for each design.
The TAR highlights that if the system supports 64-bit operations, the B version of the TCAs is more advantageous, giving the best performance while maintaining a small area increment.

6.2 Loosely-coupled accelerator

After testing it, also the loosely-coupled accelerator has been profiled and synthesised, obtaining the results reported in Table 6.2. Due to the aforementioned Sample Splitting limitation, the profiling was performed in the worst-case scenario, where all the sample blocks have been compressed using that technique.

The LCA increases the performance substantially, achieving a speed-up factor of $\times 480$ in the worst-case scenario. When the Sample Splitting technique is not

Metrics	Ref. LCA (added AHB)	LCA
CC	1,029,520,644	2,143,543
CC reduction	0	99.8%
Speed-up	1.0	480.3
ASIC synthesis technology	l.p. 65 nm	l.p. 65 nm
Max Frequency [MHz]	500	500
Total Core Area [μm^2]	175138	258310
Area Increment	0%	47.5%
Throughput at f_{max}[Gb/s]	0.016	7.827
TAR	93.05	30300.27

Table 6.2: LCA performance and area results

used, one sample can be computed every clock cycle, achieving a speed-up of $\times 924$. However, this increment comes at the cost of a larger occupied area, which has grown by 47.5% compared to the reference implementation.

6.3 TCA vs LCA

The most meaningful results obtained for the developed accelerators are compared in Table 6.3. From that, it is clear that for what concerns performance, the LCA is clearly the best design. Executing all the operations and controls in hardware inside the accelerator allows concurrent executions of many processing steps. The compression of one sample per clock cycle for all the compression techniques, except for the Sample Splitting, increases the throughput up to 7.8 Gb/s, which is way higher than any of the other designs.
The TCAs, on the other hand, are more compact but enhance the compression procedure substantially, which makes them a suitable choice to have good performance with contained area overhead.

Metrics	TCA variant A: 1 sample/inst	TCA variant B: 2 samples/inst	LCA
CC reduction	85.1%	89.6%	99.8%
Speed-up	6.7	9.6	480.3
Max Frequency [MHz]	500	500	500
Area Increment	9.4%	14.0%	47.5%
Throughput at f_{max} [Gb/s]	0.109	0.157	7.827
TAR	591.76	815.49	30300.27

Table 6.3: Results comparison between the developed TCAs and LCA

As with the occupied area, other aspects must also be taken into consideration

when choosing which type of accelerator to use. TCAs pros and cons are:

+ They offer good performance increment with low area investment.

+ Since they are usually single-function units operated by software, algorithm changes are easily handled with source code changes.

+ They can be used selectively, allowing one or more TCAs to be used also for other algorithms beyond the one they were created for.

- They require high effort for their integration into existing designs. The processor core must be modified, adding the new hardware and updating all the necessary control logic. On top of that, the compiler toolchain must also be modified, adding the new custom instructions that operate the accelerators.

- To achieve the best possible performance with TCAs, a proper coding style is required. Accelerators enhances certain parts of the computation, but if the code is not properly written, those improvements may be overshadowed by the remaining part of the code. The algorithm should be written exploiting data reuse and temporal locality to reduce memory operations and speed up the execution.

• To maintain small delays, low energy consumption and high reliability, TCAs are best implemented with the ASIC technology.

Loosely-coupled accelerators, on the other hand, have different characteristics:

+ If the accelerator is properly designed, and features direct memory access (DMA), it can often outperform other types of accelerators, obtaining the best performance increment.

+ The use of a standard interface makes the LCA allowing effortless integration in many processor's subsystems. Additionally, memory-mapped accelerators do not require custom instructions to operate, making them even more portable.

+ With a proper design, after the starting command, LCAs can work concurrently with the processor.

- Since LCAs perform complex tasks, they usually require higher area investment with respect to TCAs.

• Even though an ASIC implementation guarantees the best possible performance, it is also not flexible to algorithm changes. For this reason, FPGA

implementations may be preferable.

6.4 Comparison with other works

At the time of writing, no works similar to the proposed TCAs have been reported. Project [17] obtained comparable performance results, but the approach used is completely different, featuring a multicore and GPU implementation of the algorithm. On the other hand, many implementations similar to the proposed LCA can be found in the literature, even though the only reference to an ASIC implementation is found for the SHyLoC in [18]. The SHyLoC comes in many configurations, but the one more similar to the proposed design is represented by the *set 4* that employs the runtime configuration feature. It is worth noting that, in both the proposed design and [18], small memories like the internal FIFOs have been mapped to FFs, causing an area overhead. These results should then be considered as upper bounds, because synthesis with appropriate memory macros should yield a lower area.

Metric	Proposed LCA only	SHyLoC of [18] (set 4 w mem)
Samples bit depth	32	16
ASIC synthesis technology	l.p. 65 nm	DARE 180 nm
Max Frequency [MHz]	709	153.85
Total Core Area [μm^2]	94532	3280000
Throughput at f_{max} [Gb/s]	11.10	4.92

Table 6.4: ASIC design results and comparisons

The proposed LCA has also been implemented on the XCVU3P FPGA for an easier comparison with other works. Due to the aforementioned Sample Splitting limitation, the worst-case performance of this work are slightly lower than the one reported for work [16], despite the higher frequency of the proposed design. Solving this issue would lead to better results, with additional improvements possible by reducing the critical path. Paper [19] parallelizes part of the computation of SHyLoC, achieving a throughput higher than the proposed solution, at the cost of higher resource utilization. The work in [20] presents an efficient implementation of the algorithm, but detailed material is hard to find, and implementation specifics are not provided in the available sources. The solution adopted for work [21] is constrained to image data compression targeting remote sensing applications.

This means that the maximum bit depth for the input samples is 16 bits, whereas the proposed is 32 bits. The architecture is then tailored around this constraint, achieving a smaller area footprint and better performance than this work.

Metric	Proposed LCA only	SHyLoC of [16]	Parallel121 of [19]	USES-32 of [20]	Work of [21] (D = 16)
FPGA	XCVU3P	XQR5VFX130	XCKU040	EP3SE50F484C2	XC6VLX75T
Max Frequency [MHz]	142.9	79.9	121.5	-	313
LUTs / DSPs	8929 / 3	7670 / 5	28329 / 4	6255 / -	9% slices
FFs / BRAM or LUTRAM	4275 / 96	2291 / 0	8774 / 0	3383 / 16 Kb	5% BRAMs
Throughput at f_{max} [Gb/s]	2.24	2.56	7.78	6.4	4.67

Table 6.5: FPGA design results and comparisons

6.5 Future work

The reference implementation of the algorithm taken from OBPMark is not optimally written. For example, all the input samples are read many times from the memory, wasting clock cycles in loop overheads, function calls and memory operations. Rewriting the code exploiting proper data reuse will certainly increase the performance of both the reference implementation and the TCA design.
Future work will also be done to remove the aforementioned limitation of the Sample Splitting encoding on the LCA, matching the rate of one sample per clock cycle already achieved by the other compression techniques. The resulting higher throughput would make the accelerator more suitable for high-performance designs and more competitive with respect to the other similar works.
Finally, this comparison can also be extended to the CCSDS 123.0 standard, where the complex preprocessing stage presents significant challenges for its optimization. Given the increasingly widespread use of this standard, several studies have been conducted to speed up its execution [22]. The removal of data dependencies [23] and the use of multicore solutions [24] offer interesting insights for future work possibilities.

Chapter 7

Conclusion

This work greatly enhanced the performance of the CCSDS 121.0 algorithm, exploring various strategies in the solution space.
Given the widespread support for the RISC–V project from numerous vendors and corporates, the NOEL–V proved to be an optimal candidate processor for this optimization, and ASIP Designer allowed for rapid and intuitive development of the accelerators.
The LCA proved to be the best design for what concerns TAR, reaching a throughput of 7.83 Gb/s with an area increment of 47.5%. On the other hand, TCAs still increased the performance noticeably while remaining compact enough for area constrained devices.
Comparing the developed accelerators from different perspectives provided interesting insights about their development, integration, and potential performance. The analysis highlighted key trade-offs between flexibility, efficiency, and resource utilization, and discussed specific design constraints that can guide the selection of the most suitable accelerator type depending on the application.
The developed accelerators have therefore demonstrated a promising potential, providing interesting insights and opportunities for improvement. Their design and implementation ultimately provide a solid foundation for extending this work to other, more complex, compression standards.

Appendix A

TCA model and source code

```
//------------ CCSDS121 top level opn ------------

// 0b00xxxxxxxx -> no ST or LD operation in dM
// 0b01xxxxxxxx -> require ST operation in dM
// 0b10xxxxxxxx -> require LD operation in dM

// 0bxxxxAAAxxx -> AAA used to identify instruction

enum majCUSTOM1_fn10 {
  // writeWord instructions -> 0bxxxxxxx001
  wWord    = 0b0100000001,
  rst_wW   = 0b0000001001,
  ld_lD    = 0b0000010001,
  st_nbit  = 0b0000011001,
  st_tBiW  = 0b0100100001,
  // Jblock size instructions -> 0bxxxxxxx010
  sz_wsh   = 0b0100000010,
  sz_nosh  = 0b0100001010,
  getSize  = 0b0000010010,
  getHsmp  = 0b0000011010,
  ld_k     = 0b0000100010,
  // preprocessor instructions -> 0bxxxxxxx100
  UDPproc  = 0b0100000100,
  UDPclS   = 0b0000001100,
  ld_UDPr  = 0b0000010100,
  // just needed by ASIP Designer for correct enum
  last     = 0b1111111111
};


trn argA <w64>;
trn argB <w64>;
trn argR <w64>;


opn ccsds121_pair(CDwrite_pair | Jsize_pair | UDPproc_pair);
```

```
// ------------------------------------------------------------------------
// ------------------------------ writeWord -------------------------------
// ------------------------------------------------------------------------


//------------ writeWord top level opn ------------


opn CDwrite_pair(l0 : CDwrite_instrs, l1 : alu1.alu_instrs) {
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}

opn CDwrite_instrs (CDwrite_write_instr |
                    CDwrite_reset_instr |
                    CDwrite_ld_lD       |
                    CDwrite_st_tBiW     |
                    CDwrite_st_nbit);



//------------ writeWord resources ------------

fu writeCompData;

reg nbit <w08> read (nbit_r) write (nbit_w);
reg totB <w32> read (totB_r) write (totB_w);
reg incW <w32> read (incW_r) write (incW_w);
reg lenD <w32> read (lenD_r) write (lenD_w);

trn uncompressed_data <w64>;
trn compressed_data <w32>;
trn nbits <w64>;
trn address <addr>;
trn totBytes <w64>;
trn dataLength <w64>;
trn new_nbit <w64>;
trn new_totB <w64>;

hw_init incW = 0;

//------------ writeWord instruction ------------

opn CDwrite_write_instr (op: majCUSTOM1_fn10, rs1: mX_l0.mX1r_EX, rs2: mX_l0.mX2r_EX) {
  action {
  stage RA..EX:         argA`EX` = rs1;
                        argB`EX` = rs2;
  // ---
  stage EX:
    switch (op) {
      case wWord  :     address = argA;
                        uncompressed_data = argB;
                        dm_addr = getWriteAddr(address, totB_r = totB) @writeCompData;
                        compressed_data = getWordToWrite( uncompressed_data, lenD_r=lenD,
                        ↪  incW_r=incW, incW=incW_w,
                                                          nbit_r=nbit, nbit=nbit_w, totB_r = totB,
                                                          ↪  totB=totB_w) @writeCompData;
                                                          ↪
```

```
                        DMw[dm_addr`EX`] = dmw_wr`EX` = compressed_data`EX`;

    }
  }
  syntax : op PADMNM " " rs1 "," PADOP1 rs2;
  image  : op[9..3] :: rs2 :: rs1 :: op[2..0] :: "00000" :: opc32.CUSTOM1
       class(ccsds121), class(wW_acc);
}

opn CDwrite_reset_instr (op: majCUSTOM1_fn10) {
  action {
  stage EX:
    switch (op) {
      case rst_wW  :    incW = incW_w = 0;
                        lenD = lenD_w = 0;
                        nbit = nbit_w = 0;
                        totB = totB_w = 0;

    }
  }
  syntax : op PADMNM;
  image  : op[9..3] :: "00000" :: "00000" :: op[2..0] :: "00000" :: opc32.CUSTOM1
       class(ccsds121), class(wW_acc);
}


//------------ writeWord load instructions ------------

opn CDwrite_ld_lD (op: majCUSTOM1_fn10, rs2: mX_l0.mX1r_EX) {
  action {
  stage RA..EX:         argB `EX` = rs2;
  // ---
  stage EX:
    switch (op) {
      case ld_lD  :     lenD = lenD_w = argB;
    }
  }
  syntax : op PADMNM " " rs2;
  image  : op[9..3] :: rs2 :: "00000" :: op[2..0] :: "00000" :: opc32.CUSTOM1
       class(ccsds121);
}



//------------ writeWord store instructions ------------

opn CDwrite_st_tBiW (op: majCUSTOM1_fn10, rs1: mX_l0.mX1r_EX, rd: mX_l0.mX1w_EX) {
  action {
  stage RA..EX:         argA `EX` = rs1;
  // ---
  stage EX:
    switch (op) {
      case st_tBiW  :   address = argA;
                        dm_addr = getWriteAddr(address, totB_r = totB) @writeCompData;
                        DMw[dm_addr`EX`] = dmw_wr`EX` = change_endian(incW_r`EX` =
                        ↪  incW`EX`)@writeCompData;
                        argR = getNewTotB(totB_r = totB, nbit_r = nbit) @writeCompData;
    }
  // ---
  stage EX..WR:         rd = argR`EX`;
  }
```

```
  syntax : op PADMNM " " rd "," PADOP1 rs1;
  image  : op[9..3] :: "00000" :: rs1 :: op[2..0] :: rd :: opc32.CUSTOM1
       class(ccsds121);
}

opn CDwrite_st_nbit (op: majCUSTOM1_fn10, rd: mX_l0.mX1w_EX) {
  action {
  stage EX:
    switch (op) {
      case st_nbit  :   argR = getNewNbit(nbit_r = nbit) @writeCompData;
    }
  // ---
  stage EX..WR:         rd = argR`EX`;
  }
  syntax : op PADMNM " " rd;
  image  : op[9..3] :: "00000" :: "00000" :: op[2..0] :: rd :: opc32.CUSTOM1
       class(ccsds121);
}


//------------ writeWord chess_view and sw_stall ------------

// writeWord
chess_view () {
  dm_addr = getWriteAddr(address, totB_r=totB);
  compressed_data = getWordToWrite(uncompressed_data, lenD_r=lenD, incW_r=incW, incW=incW_w,
  ↪  nbit_r=nbit, nbit=nbit_w, totB_r=totB, totB=totB_w);
  DMw[dm_addr] = compressed_data;
} -> {
  writeWord_p(address, uncompressed_data);
}

// reset accelerator registers
chess_view () {
  incW = incW_w = 0;
  lenD = lenD_w = 0;
  nbit = nbit_w = 0;
  totB = totB_w = 0;
} -> {
  reset_wWord_p();
}

// load of lenD
chess_view () {
  lenD = lenD_w = argB;
} -> {
  ld_lD(argB);
}

// store of totB and incW at the end of execution
chess_view () {
  dm_addr = getWriteAddr(address, totB_r=totB);
  DMw[dm_addr] = change_endian(incW_r=incW);
  argR = getNewTotB(totB_r=totB, nbit_r=nbit);
} -> {
  argR = st_tBiW_p(address);
}

// store of nbit at the end of execution
chess_view () {
  argR = getNewNbit(nbit_r = nbit);
```

```
} -> {
  argR = st_nbit_p();
}

// ---------------------------------------------------------------------------
// ------------------------- Jblock accelerator ------------------------------
// ---------------------------------------------------------------------------


//------------ Jblock accelerator resources ------------

fu jBlockFU;

reg k_reg <w08> read (k_r) write (k_w);
reg size_reg <w32> read (size_r) write (size_w);
hw_init size_reg = 0;

trn k <w08>;
trn sample1 <w32>;
trn sample2 <w32>;
trn hSample <w32>;



//------------ Jblock acceleratore instructions ------------

opn Jsize_pair(l0 : Jsize_acc_instr, l1 : alu1.alu_instrs) {
     dummy_syntax : l1;
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}

opn Jsize_acc_instr (Jsize_hSmp_instr | Jsize_calcSize_instr | Jsize_getSize_instr |
↪  Jsize_ldK_instr);

opn Jsize_hSmp_instr (op: majCUSTOM1_fn10, rd: mX_l0.mX1w_EX, rs1: mX_l0.mX1r_EX, rs2:
↪  mX_l0.mX2r_EX) {
    action{
        stage RA..EX:   argA`EX` = rs1;
                        argB`EX` = rs2;
        stage EX:
            switch (op) {
                    case getHsmp:
                            sample2 = argA;
                            sample1 = argB;
                            argR = hSample = get_hSample(sample1, sample2) @jBlockFU;
            }
        stage EX..WR: rd = argR`EX`;
    }
    syntax : op PADMNM " " rd "," PADOP1 rs1 "," PADOP2 rs2;
    image  : op[9..3] :: rs2 :: rs1 :: op[2..0] :: rd :: opc32.CUSTOM1;
}


opn Jsize_calcSize_instr (op: majCUSTOM1_fn10, rs1: mX_l0.mX1r_EX, rs2: mX_l0.mX2r_EX) {
    action{
        stage RA..EX:        argA`EX` = rs1;
                    argB`EX` = rs2;
        // ---
        stage EX:
            switch (op) {
```

```
                    case sz_wsh:
                            dm_addr = address = argA;
                            sample1 = argB;
                            DMw[dm_addr`EX`] = dmw_wr`EX` = sample1`EX`;
                            k = k_r = k_reg;
                            size_reg = size_w = addSize(size_r = size_reg, sample1, k) @jBlockFU;
                    case sz_nosh:
                            dm_addr = address = argA;
                            sample1 = argB;
                            DMw[dm_addr`EX`] = dmw_wr`EX` = sample1`EX`;
                            k = 0;
                            size_reg = size_w = addSize(size_r = size_reg, sample1, k) @jBlockFU;
            }
    }
    syntax : op PADMNM " " rs1 "," PADOP1 rs2;
    image  : op[9..3] :: rs2 :: rs1 :: op[2..0] :: "00000" :: opc32.CUSTOM1;
}


opn Jsize_getSize_instr (op: majCUSTOM1_fn10, rd: mX_l0.mX1w_EX) {
    action{
        stage EX:
            switch (op) {
                    case getSize:
                            argR = size_r = size_reg;
                            size_reg = size_w = 0;
            }
        stage EX..WR: rd = argR`EX`;
    }
    syntax : op PADMNM " " rd;
    image  : op[9..3] :: "00000" :: "00000" :: op[2..0] :: rd :: opc32.CUSTOM1;
}

opn Jsize_ldK_instr (op: majCUSTOM1_fn10, rs1: mX_l0.mX1r_EX) {
    action{
        stage RA..EX:        argA`EX` = rs1;
        // ---
        stage EX:
            switch (op) {
                    case ld_k:
                            k_reg = k_w = argA;
            }
    }
    syntax : op PADMNM " " rs1;
    image  : op[9..3] :: "00000" :: rs1 :: op[2..0] :: "00000" :: opc32.CUSTOM1;
}


//------------ Jblock accelerator chess_view ------------

// sampleSplitting size and memory store
chess_view () {
  DMw[dm_addr = address] = dmw_wr = sample1;
  size_w = addSize(size_r, sample1, k_r = k_reg);
} -> {
  sizeWshift(address, sample1);
}

// fundamentalSequence and halvedSamples size and memory store
chess_view () {
  DMw[dm_addr = address] = dmw_wr = sample1;
```

```
  k = 0;
  size_w = addSize(size_r, sample1, k);
} -> {
  sizeNOshift(address, sample1);
}

// get calculated size
chess_view () {
  argR = size_r = size_reg;
  size_reg = size_w = 0;
} -> {
  argR = getSize();
}

// calculate HalvedSample from two J-samples
chess_view () {
  hSample = get_hSample(sample1, sample2);
} -> {
  hSample = calc_hSampl(sample1, sample2);
}

// load k_reg auxiliary register
chess_view () {
  k_reg = k_w = argA;
} -> {
  ld_k(argA);
}


// ------------------------------------------------------------------------
// ----------------------------- preprocessor -----------------------------
// ------------------------------------------------------------------------


//------------ preprocessor top level opn ------------


opn UDPproc_pair(l0 : UDPproc_instr, l1 : alu1.alu_instrs) {
     dummy_syntax : l1;
     syntax : l0 PADINST " : " l1;
     image  : l0::"11"::l1::"11";
}

opn UDPproc_instr (UDP_process_instr | UDP_clear_instr | UDP_reg_instr);


//------------ preprocessor resources ------------

fu UDP;

reg UDPdSmp <w32> read (UDPdSmp_r) write (UDPdSmp_w);
hw_init UDPdSmp = 0;

reg UDPactive <t1u> read (UDPactive_r) write (UDPactive_w);
reg UDPxmax <w32> read (UDPxmax_r) write (UDPxmax_w);

trn data_to_preprocess <w32>;
trn preprocessed_data <w32>;

//------------ preprocessor instruction ------------
```

```
opn UDP_process_instr (op: majCUSTOM1_fn10, rs1: mX_l0.mX1r_EX, rs2: mX_l0.mX2r_EX, rd:
↪  mX_l0.mX1w_EX) {
  action {
  stage RA..EX:          argA`EX` = rs1;
                         argB`EX` = rs2;
  // ---
  stage EX:
    switch (op) {
      case UDPproc  :    dm_addr = address = argA;
                         data_to_preprocess = argB;
                         preprocessed_data = UDP_PEM(data_to_preprocess, UDPdSmp_r=UDPdSmp,
                         ↪  UDPactive_r=UDPactive, UDPxmax_r=UDPxmax) @UDP;
                         argR = preprocessed_data;
                         UDPdSmp = UDPdSmp_w = data_to_preprocess;
                         DMw[dm_addr`EX`] = dmw_wr`EX` = preprocessed_data`EX`;
                         ↪
    }
  // ---
  stage EX..WR:          rd = argR`EX`;
  }

  syntax : op PADMNM " " rd "," PADOP1 rs1 "," PADOP2 rs2;
  image  : op[9..3] :: rs2 :: rs1 :: op[2..0] :: rd :: opc32.CUSTOM1
       class(ccsds121);
}

opn UDP_clear_instr (op: majCUSTOM1_fn10) {
  action {
  stage EX:
    switch (op) {
      case UDPclS   :    UDPdSmp = UDPdSmp_w = 0;
    }
  }
  syntax : op;
  image  : op[9..3] :: "00000" :: "00000" :: op[2..0] :: "00000" :: opc32.CUSTOM1
       class(ccsds121);
}

//------------ preprocessor load instructions ------------

opn UDP_reg_instr (op: majCUSTOM1_fn10, rs1: mX_l1.mX1r_EX, rs2: mX_l1.mX2r_EX) {
  action {
  stage RA..EX:          argA`EX` = rs1;
                         argB`EX` = rs2;
  // ---
  stage EX:
    switch (op) {
      case ld_UDPr  :    UDPactive = UDPactive_w = argA;
                         UDPxmax = UDPxmax_w = argB;
    }
  }
  syntax : op PADMNM " " rs1 "," PADOP1 rs2;
  image  : op[9..3] :: rs2 :: rs1 :: op[2..0] :: "00000" :: opc32.CUSTOM1
       class(ccsds121);
}

//------------ preprocessor chess_view ------------

// preprocessor
chess_view () {
  dm_addr = address;
```

```
  preprocessed_data = UDP_PEM(data_to_preprocess, UDPdSmp_r=UDPdSmp, UDPactive_r=UDPactive,
  ↪  UDPxmax_r=UDPxmax);
  UDPdSmp = UDPdSmp_w = data_to_preprocess;
  DMw[dm_addr] = preprocessed_data;
} -> {
  preprocessed_data = UDPprocess(address, data_to_preprocess);
}

// clear UDPdSmp
chess_view () {
  UDPdSmp = UDPdSmp_w = 0;
} -> {
  UDPclear();
}

// load UDP registers
chess_view () {
  UDPactive = UDPactive_w = argA;
  UDPxmax = UDPxmax_w = argB;
} -> {
  UDPreg(argA, argB);
}
```

Code A.1: nML description of TCAs

```
class v33_uint32_t property (vector uint32_t [33]);

const v33_uint32_t lenD_mask = {
    "00000000000000000000000000000000",
    "00000000000000000000000000000001",
    "00000000000000000000000000000011",
    "00000000000000000000000000000111",
    "00000000000000000000000000001111",
    "00000000000000000000000000011111",
    "00000000000000000000000000111111",
    "00000000000000000000000001111111",
    "00000000000000000000000011111111",
    "00000000000000000000000111111111",
    "00000000000000000000001111111111",
    "00000000000000000000011111111111",
    "00000000000000000000111111111111",
    "00000000000000000001111111111111",
    "00000000000000000011111111111111",
    "00000000000000000111111111111111",
    "00000000000000001111111111111111",
    "00000000000000011111111111111111",
    "00000000000000111111111111111111",
    "00000000000001111111111111111111",
    "00000000000011111111111111111111",
    "00000000000111111111111111111111",
    "00000000001111111111111111111111",
    "00000000011111111111111111111111",
    "00000000111111111111111111111111",
    "00000001111111111111111111111111",
    "00000011111111111111111111111111",
    "00000111111111111111111111111111",
    "00001111111111111111111111111111",
    "00011111111111111111111111111111",
    "00111111111111111111111111111111",
```

```
    "01111111111111111111111111111111",
    "11111111111111111111111111111111"
};

w32 getWordToWrite(w64 data_i, w32 data_length, w32 incW_i, w32& incW_o, w08 nbit_i, w08& nbit_o,
↪  w32 totB_i, w32& totB_o){
  uint64_t buffer = incW_i::(uint32_t)0;
  uint32_t data = ((uint32_t) data_i) & lenD_mask[data_length];
  uint8_t bits_in_buffer = nbit_i + (uint8_t) data_length;

  buffer |= (uint64_t) (((uint64_t) data) << (64 - bits_in_buffer));


  if (bits_in_buffer < 32){
    incW_o = buffer[63:32];
    nbit_o = (uint8_t) bits_in_buffer;
    totB_o = totB_i;
  }
  else{
    incW_o = buffer[31:0];
    nbit_o = bits_in_buffer - 32;
    totB_o = totB_i + 4;
  }
  return change_endian(buffer[63:32]);
}

w32 change_endian(w32 a){
  return a[7:0]::a[15:8]::a[23:16]::a[31:24];
}

addr getWriteAddr(addr address, w32 totB_i){
  return address + (uint64_t) totB_i[31:2]::"00";
}

w64 getNewNbit(w08 nbit_i){
  return (uint64_t) ("0"::nbit_i[2:0]);
}

w64 getNewTotB(w32 totB_i, w08 nbit_i){
  return (uint64_t) (totB_i + nbit_i[7:3]);
}


// ------------------------------------------------------------------------
// -------------------------- Jblock accelerator --------------------------
// ------------------------------------------------------------------------

w32 addSize(w32 old_size, w32 sample, w08 k_i){
  return (uint32_t) ((uint32_t)old_size + (uint8_t)k_i + (((uint32_t)sample) >> k_i[4:0]) + 1);
}

w32 get_hSample(w32 sample1, w32 sample2){
  return (uint32_t) ( ((sample1 + sample2) * (sample1 + sample2 + 1)) >> 1) + sample2;
}


// ------------------------------------------------------------------------
// ---------------------- preprocessor accelerator ------------------------
// ------------------------------------------------------------------------

w32 UDP_PEM(w32 new_sample, w32 PredictedValue, t1u preprocessor_active, w32 x_max){
```

```
  if(preprocessor_active){
      int32_t PredictionError = new_sample - PredictedValue;
      int32_t difference = x_max - PredictedValue;
      int32_t theta = (PredictedValue < difference) ? PredictedValue : difference;
      int32_t PredictionErrorAbs = (PredictionError > 0) ? PredictionError : -PredictionError;
      int32_t PreprocessedSample;

      if(0 <= PredictionError && PredictionError <= theta)
      {
          PreprocessedSample = 2*PredictionError;
      }
      else if(-theta <= PredictionError && PredictionError < 0)
      {
          PreprocessedSample = 2*PredictionErrorAbs -1;
      }
      else{
          PreprocessedSample = theta + PredictionErrorAbs;
      }

      return PreprocessedSample;
  }
  return new_sample;
}
```

Code A.2: PDG implementation of TCA units

```
#ifndef INCLUDED_NOELV_P_CCSDS121_H_
#define INCLUDED_NOELV_P_CCSDS121_H_



namespace noelv_primitive {

  // ------------------------------------------------------------------------
  // ------------------------------ writeWord -------------------------------
  // ------------------------------------------------------------------------

  // accelerator
  void writeWord_p(addr, w64) property(alternate_store programmers_view);
  w32 getWordToWrite(w64, w32, w32, w32&, w08, w08&, w32, w32&);
  void reset_wWord_p() property(programmers_view);

  // load registers
  void ld_lD(w64) property(programmers_view);

  // store registers
  w64 st_tBiW_p(addr) property(alternate_store programmers_view);
  w64 st_nbit_p() property(programmers_view);
  w64 getNewNbit(w08);
  w64 getNewTotB(w32, w08);

  // common auxiliary primitives
  w32 change_endian(w32);
  addr getWriteAddr(addr, w32);


  // ------------------------------------------------------------------------
  // ------------------------- Jblock accelerator ---------------------------
```

```
  // ------------------------------------------------------------------------

  // load/store instructions
  void ld_k(w64) property(programmers_view);

  // size calculation
  w32 addSize(w32, w32, w08);

  // HalvedSample operations
  w32 get_hSample(w32, w32);

  // programmers_view primitives for the compiler
  void sizeWshift(addr, w32) property(alternate_store programmers_view);
  void sizeNOshift(addr, w32) property(alternate_store programmers_view);
  w64 getSize() property(programmers_view);
  w32 calc_hSampl(w32, w32) property(programmers_view);


  // ------------------------------------------------------------------------
  // ---------------------- preprocessor accelerator ------------------------
  // ------------------------------------------------------------------------

  // UDP preprocesso implementation
  w32 UDP_PEM(w32, w32, t1u, w32);

  // programmers_view primitives for the compiler
  w32 UDPprocess(addr, w32) property(alternate_store programmers_view);
  void UDPclear() property(programmers_view);
  void UDPreg(w64, w64) property(programmers_view);

}

#endif // INCLUDED_NOELV_P_CCSDS121_H_
```

Code A.3: TCA primitives declaration

```
#ifndef INCLUDED_NOELV_C_CCSDS121_H_
#define INCLUDED_NOELV_C_CCSDS121_H_

// ------------------------------------------------------------------------
// ------------------------------ writeWord -------------------------------
// ------------------------------------------------------------------------

// accelerator
promotion void writeWord(unsigned char *, int) = void writeWord_p(addr, w64);
promotion void reset_writeWord() = void reset_wWord_p();

// load registers
promotion void loadLenData(int) = void ld_lD(w64);

// store registers
promotion unsigned int storeLastWordNewTotBytes(unsigned char *) = w64 st_tBiW_p(addr);
promotion unsigned int storeNewNumBits() = w64 st_nbit_p();

// ------------------------------------------------------------------------
// ------------------------- Jblock accelerator ---------------------------
// ------------------------------------------------------------------------

// load/store
```

```
promotion void ld_k(int) = void ld_k(w64);

// size calculation
promotion void addSizeSampleSplitting(unsigned int*, unsigned int) = void sizeWshift(addr, w32);
promotion void addSizeFundamentalSequence(unsigned int*, unsigned int) = void sizeNOshift(addr,
↪  w32);
promotion void addSizeHalvedSamples(unsigned int*, unsigned int) = void sizeNOshift(addr, w32);
promotion unsigned int getCalculatedSize() = w64 getSize();

// HalvedSample calculation
promotion unsigned int calculateHalvedSample(unsigned int, unsigned int) = w32 calc_hSampl(w32,
↪  w32);


// ------------------------------------------------------------------------
// ---------------------- preprocessor accelerator ------------------------
// ------------------------------------------------------------------------

promotion unsigned int UDPprocess(unsigned int*, unsigned int) = w32 UDPprocess(addr, w32);
promotion void UDPclearDelayedStack() = void UDPclear();
promotion void UDPloadRegs(int, int) = void UDPreg(w64, w64);


#endif // INCLUDED_NOELV_C_CCSDS121_H_
```

Code A.4: Chess compiler header file to use the TCA functions

Appendix B

LCA model and source code

```
/*
-- File : dm.p
--
-- Contents : Definition of the noelv DM IO interface.
-- This IO interfaces merges the aligned access from DMb DMh DMw DMd
--
-- Copyright (c) 2019-2022 Synopsys, Inc. This Synopsys processor model
-- captures an ASIP Designer Design Technique. The model and all associated
-- documentation are proprietary to Synopsys, Inc. and may only be used
-- pursuant to the terms and conditions of a written license agreement with
-- Synopsys, Inc.  All other use, reproduction, modification, or distribution
-- of the Synopsys processor model or the associated  documentation is
-- strictly prohibited.
*/

#include "noelv_define.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ DM IO Interface
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// One io_interface unit
//  1) merge byte/half/word/double accesses


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Merge byte/half/word/double accesses
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// This IO Interface
// * merges the memory record aliases
// * interfaces to a single-cycle 64b wide memory with byte enables
// * supports only aligned addresses
// * has an external interface with double-word addresses

io_interface dm_merge (DMb) {

  // Assumption:
  // nml_side {
  //   mem DMb [SIZE,1]<w08,addr> access {
```

```
//     dmb_ld`0`: dmb_rd`1` = DMb[dm_addr`0`]`1`;
//     dmb_st`0`: DMb[dm_addr`0`]`0` = dmb_wr`0`;
//   };
//   mem DMh [SIZE,2]<w16,addr> alias DMb access {
//     dmh_ld`0`: dmh_rd`1` = DMh[dm_addr`0`]`1`;
//     dmh_st`0`: DMh[dm_addr`0`]`0` = dmh_wr`0`;
//   };
//   mem DMw [SIZE,4]<w32,addr> alias DMb access {
//     dmw_ld`0`: dmw_rd`1` = DMw[dm_addr`0`]`1`;
//     dmw_st`0`: DMw[dm_addr`0`]`0` = dmw_wr`0`;
//   };
//   mem DMd [SIZE,8]<w64,addr> alias DMb access {
//     dmd_ld`0`: dmd_rd`1` = DMd[dm_addr`0`]`1`;
//     dmd_st`0`: DMd[dm_addr`0`]`0` = dmd_wr`0`;
//   };
// }

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ External Interface
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

trn edm_st <uint8_t>; // byte write mask

mem eDM [2**61] <v8u8,addr> access {
  edm_ld`0`: edm_rd`1`           = eDM[edm_addr_rd`0`]`1`;
  edm_st`0`: eDM[edm_addr_wr`0`]`0` = edm_wr`0`;
};


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Accelerator Interface
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// slave input
outport hsel_slv_i    <uint1_t>;
outport haddr_slv_i   <uint32_t>;
outport hwrite_slv_i  <uint1_t>;
outport htrans_slv_i  <uint2_t>;
outport hsize_slv_i   <uint3_t>;
outport hwdata_slv_i  <uint32_t>;
outport hready_slv_i  <uint1_t>;

// slave output
inport hready_slv_o   <uint1_t>;
inport hresp_slv_o    <uint1_t>;
inport hrdata_slv_o   <uint32_t>;

// master input
outport hready_mst_i  <uint1_t>;
outport hgrant_mst_i  <uint1_t>;
outport hrdata_mst_i  <uint32_t>;
outport hresp_mst_i   <uint1_t>;

// master output
inport haddr_mst_o    <uint32_t>;
inport hwrite_mst_o   <uint1_t>;
inport htrans_mst_o   <uint2_t>;
inport hsize_mst_o    <uint3_t>;
inport hlock_mst_o    <uint1_t>;
inport hbusreq_mst_o  <uint1_t>;
```

```
// back-end memory port for accelerator results writing
inport wrport_addr_acc_o    <uint32_t>;
inport wrport_datain_acc_o  <uint32_t>;
inport wrport_en_acc_o      <uint1_t>;
inport wrport_write_acc_o   <uint1_t>;


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Local Storage
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

reg col_ff <uint3_t>; hw_init col_ff = 0;

// grant register:
// 0 -> normal operations memory operations and ahb slave communications
// 1 -> ahb master communications
reg grant_reg     <uint1_t>;

// ahb slave registers
reg ahb_slv_reg_data   <uint32_t>;
reg ahb_slv_reg_rd_ncc     <uint1_t>;
reg ahb_slv_reg_data_ph   <uint1_t>;


// ahb master registers
reg ahb_mst_reg_rd_ncc    <uint1_t>;

hw_init ahb_slv_reg_data = 0;
hw_init ahb_slv_reg_rd_ncc = 0;
hw_init ahb_slv_reg_data_ph = 0;
hw_init grant_reg = 0;

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Process response from memory
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

process process_result() {


  // dmb_rd, dmh_rd, dmw_rd, dmd_rd
  u08 b0 = edm_rd[col_ff];                    // 0, 1, 2, ...
  // dmh_rd, dmw_rd, dmd_rd
  u08 b1 = edm_rd[col_ff[2:1]::"1" ];         // 1, 3, 5, 7
  // dmw_rd, dmd_rd
  u08 b2 = edm_rd[col_ff[2]  ::"10"];         // 2, 6
  u08 b3 = edm_rd[col_ff[2]  ::"11"];         // 3, 7


  //default values of ahb signals
  hsel_slv_i = "0";
  haddr_slv_i = 0;
  hwrite_slv_i = 0;
  htrans_slv_i = "00";
  hsize_slv_i = "00";
  hready_slv_i = 0 ;
  hwdata_slv_i = 0;
  hready_mst_i = 0;

  // ~~~~~~ normal memory op & ahb slave interaction ~~~~~~
  if (grant_reg == "0"){
```

```
    // data phase
    if (ahb_slv_reg_data_ph){
      hwdata_slv_i = ahb_slv_reg_data;
      hready_slv_i = 1;
    }
    // address phase
    uint1_t st_op = dmd_st | dmw_st | dmh_st | dmb_st;
    uint1_t ld_op = dmd_ld | dmw_ld | dmh_ld | dmb_ld;
    uint1_t mem_op = ld_op | st_op;
    if (hready_slv_o && mem_op && (dm_addr >= ACC_SPACE_START) && (dm_addr <= ACC_SPACE_END)){
      hsel_slv_i = "1";             // accelerator selected
      haddr_slv_i = dm_addr[31:0];  // address bus
      hwrite_slv_i = st_op;         // if the memory requested a store the operation is a write
      htrans_slv_i = "10";          // "10" means non-sequential transfer
      hsize_slv_i = "010";          // "010" means 32 bits transfer
      hready_slv_i = 1;             // 1 since there are no other slaves that can interrupt the
      ↪  accelerator
    }

    if (ahb_slv_reg_rd_ncc){
      if(hready_slv_o){
        dmd_rd = (uint64_t) hrdata_slv_o;
        dmw_rd = hrdata_slv_o;
        dmh_rd = hrdata_slv_o[15:0];
        dmb_rd = hrdata_slv_o[7:0];
      }
    }
    else{
      dmd_rd = edm_rd[7]::edm_rd[6]::edm_rd[5]::edm_rd[4]::b3::b2::b1::b0;
      dmw_rd = b3::b2::b1::b0;
      dmh_rd = b1::b0;
      dmb_rd = b0;
    }
  }


  // ~~~~~~ ahb master interaction ~~~~~~
  else if (grant_reg == "1"){

    hready_mst_i = 1;
    if(ahb_mst_reg_rd_ncc){
      hrdata_mst_i = b3::b2::b1::b0;
    }

  }

  // grant signal passed to ahb master
  hgrant_mst_i = grant_reg;

}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Process request from core
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

process process_request() {

  uint1_t st_op = dmd_st | dmw_st | dmh_st | dmb_st;
  uint1_t ld_op = dmd_ld | dmw_ld | dmh_ld | dmb_ld;
  uint1_t mem_op = ld_op | st_op;
```

```
// ~~~~~~ normal memory op & ahb slave interaction ~~~~~~
if(grant_reg == "0"){

  // split address
  addr    row = dm_addr[63:3];
  uint3_t col = dm_addr[2:0];

  // addr (read or write)
  edm_addr_rd = row;
  edm_addr_wr = row;

  // read enable
  edm_ld = ld_op;

  // register read info
  col_ff = col;

  // write enable
  uint8_t          t1 = "00000000";
  if      (dmd_st) t1 = "11111111";
  else if (dmw_st) t1 = "00001111" << (col[2]  ::"00");
  else if (dmh_st) t1 = "00000011" << (col[2:1]::"0" );
  else if (dmb_st) t1 = "00000001" << (col);
  edm_st = t1;

  // write data
  if (dmd_st) {
    edm_wr = dmd_wr;
  } else if (dmw_st) {
    edm_wr = dmw_wr::dmw_wr;
  } else if (dmh_st) {
    edm_wr = dmh_wr::dmh_wr::dmh_wr::dmh_wr;
  } else if (dmb_st) {
    edm_wr = dmb_wr::dmb_wr::dmb_wr::dmb_wr::dmb_wr::dmb_wr::dmb_wr::dmb_wr;
  }

  // accelerator data phase
  uint1_t ahb_op = mem_op && (dm_addr >= ACC_SPACE_START) && (dm_addr < ACC_SPACE_END);
  ahb_slv_reg_data_ph = 0;
  ahb_slv_reg_rd_ncc = 0;

  if (ahb_op){
    //deactivate memory
    edm_st = "00000000";
    edm_ld = 0;
    if (hready_slv_o){

      // data is stored to be passed to the accelerator next cycle
      if (dmd_st) {
        ahb_slv_reg_data = dmd_wr[31:0];
      } else if (dmw_st) {
        ahb_slv_reg_data = dmw_wr;
      } else if (dmh_st) {
        ahb_slv_reg_data = (uint32_t) dmh_wr;
      } else if (dmb_st) {
        ahb_slv_reg_data = (uint32_t) dmb_wr;
      }
      ahb_slv_reg_data_ph = 1;      // load ff to write in the next cycle
      ahb_slv_reg_rd_ncc = ld_op;   // if the operation is a load the result will be loaded the
      ↪  next cycle
```

```
        }
        else{
          insert_wait_this_cycle();
          ahb_slv_reg_data_ph = ahb_slv_reg_data_ph;
          ahb_slv_reg_rd_ncc = ahb_slv_reg_rd_ncc;
          ahb_slv_reg_data = ahb_slv_reg_data;
        }
      }

      if(hbusreq_mst_o){
        if(!mem_op || (mem_op && !hready_slv_o)){
          grant_reg = "1";
        }
      }

    }


    // ~~~~~~ ahb master interaction ~~~~~~
    else if(grant_reg == "1"){

      // if the processor request a memory operation a wait must be inserted
      if(mem_op){
        insert_wait_this_cycle();
        if(!hlock_mst_o){
          grant_reg = "0";
        }
      }

      // check if the operation is valid
      uint1_t ahb_mst_op = (htrans_mst_o == "10") && (hsize_mst_o == "010");

      // if the operation is a read, it must start immediatly
      edm_ld = !hwrite_mst_o & ahb_mst_op;
      edm_addr_rd = haddr_mst_o[31:3];
      col_ff = haddr_mst_o[2:0];
      ahb_mst_reg_rd_ncc = !hwrite_mst_o & ahb_mst_op;

      // if the operation is a store, the second port is used
      if(wrport_en_acc_o & wrport_write_acc_o){
        edm_wr = wrport_datain_acc_o::wrport_datain_acc_o;
        edm_addr_wr = (uint64_t) wrport_addr_acc_o[31:3];
        edm_st = "00001111" << (wrport_addr_acc_o[2] ::"00");
      }
    }
  }


  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ~~~ DC Synthesis embedded options
  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef SYNTHESIS_NO_UNGROUP
  vlog synthesis_options() {%

    // Ungroup this design. It is in the critical path.
    %% synopsys dc_tcl_script_begin
    %% set_ungroup [current_design]
    %% synopsys dc_tcl_script_end

  %}
```

```
#endif


  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ~~~ Debug Access
  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  void dbg_access_DMb(AddressType a, w08& val, bool read) {
    addr    row = a[61:3];
    uint3_t col = a[2:0];

    u08 v = val;
    dbg_access_eDM(row,col,v,read);
    val = v;
  }
}
```

Code B.1: PDG implementation of the memory interface

```
#include "noelv_define.h"


#define FS_CID_INT 0
#define SE_CID_INT 30
#define ZB_CID_INT 31
#define NC_CID_INT 32

// for mask
class v33_uint32_t property (vector uint32_t [33]);

// for sample splitting register
class v30_uint32_t property (vector uint32_t [30]);

// for LOD
class v15_uint1_t property (vector uint1_t [15]);

// for J register
class v64_uint32_t property (vector uint32_t [64]);

// for H register
class v32_uint32_t property (vector uint32_t [32]);

const v33_uint32_t w32_mask = {
    "000000000000000000000000000000000",
    "000000000000000000000000000000001",
    "000000000000000000000000000000011",
    "000000000000000000000000000000111",
    "000000000000000000000000000001111",
    "000000000000000000000000000011111",
    "000000000000000000000000000111111",
    "000000000000000000000000001111111",
    "000000000000000000000000011111111",
    "000000000000000000000000111111111",
    "000000000000000000000001111111111",
    "000000000000000000000011111111111",
    "000000000000000000000111111111111",
    "000000000000000000001111111111111",
    "000000000000000000011111111111111",
    "000000000000000000111111111111111",
```

```
    "00000000000000011111111111111111",
    "00000000000000111111111111111111",
    "00000000000001111111111111111111",
    "00000000000011111111111111111111",
    "00000000000111111111111111111111",
    "00000000001111111111111111111111",
    "00000000011111111111111111111111",
    "00000000111111111111111111111111",
    "00000001111111111111111111111111",
    "00000011111111111111111111111111",
    "00000111111111111111111111111111",
    "00001111111111111111111111111111",
    "00011111111111111111111111111111",
    "00111111111111111111111111111111",
    "01111111111111111111111111111111",
    "11111111111111111111111111111111"
};


io_interface loose_acc (){


  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ~~~ External Interface
  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  // slave input
  inport hsel_slv_i    <uint1_t>;
  inport haddr_slv_i   <uint32_t>;
  inport hwrite_slv_i  <uint1_t>;
  inport htrans_slv_i  <uint2_t>;
  inport hsize_slv_i   <uint3_t>;
  inport hwdata_slv_i  <uint32_t>;
  inport hready_slv_i  <uint1_t>;

  // slave output
  outport hready_slv_o   <uint1_t>;
  outport hresp_slv_o    <uint1_t>;
  outport hrdata_slv_o   <uint32_t>;

  // master input
  inport hready_mst_i  <uint1_t>;
  inport hgrant_mst_i  <uint1_t>;
  inport hrdata_mst_i  <uint32_t>;
  inport hresp_mst_i   <uint1_t>;

  // master output
  outport haddr_mst_o    <uint32_t>;
  outport hwrite_mst_o   <uint1_t>;
  outport htrans_mst_o   <uint2_t>;
  outport hsize_mst_o    <uint3_t>;
  outport hlock_mst_o    <uint1_t>;
  outport hbusreq_mst_o  <uint1_t>;

  // back-end memory port for accelerator results writing
  outport wrport_addr_acc_o    <uint32_t>;
  outport wrport_datain_acc_o  <uint32_t>;
  outport wrport_en_acc_o      <uint1_t>;
  outport wrport_write_acc_o   <uint1_t>;

  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// ~~~ Local Accelerator Storages
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

reg regJ <v64_uint32_t>;
reg regH <v32_uint32_t>;

// algorithm parameters
reg n_bits                <uint6_t>;
reg j_blocksize           <uint7_t>;
reg r_samplesInterval     <uint13_t>;
reg preprocessor_active   <uint1_t>;
reg tot_samples           <uint32_t>;
reg samples_address       <uint32_t>;
reg cdata_address         <uint32_t>;

// counters
reg cnt_smp               <uint32_t>;
reg cnt_j_wr              <uint7_t>;
reg cnt_j_rd              <uint7_t>;
reg cnt_r                 <uint13_t>;

// preprocessing
reg smp_in                <uint32_t>;
reg smp_in_del            <uint32_t>;
reg ppsmp                 <uint32_t>;

// delayed preprocessed sample for H-sample calculation
reg ppsmp_del             <uint32_t>;

// size calculation
reg size_se               <uint32_t>;
reg size_k                <v30_uint32_t>;
trn size_lk               <uint32_t>;

// ZeroBlock registers
reg all_0_block_ff        <uint1_t>;
reg nZeros                <uint6_t>;

// compression technique identification
reg cid_int               <uint6_t>;

// memory write registers
reg nbit                  <uint6_t>;
reg incW                  <uint32_t>;
reg totB                  <uint32_t>;
trn smp_to_wr             <uint32_t>;
trn cdata_to_wr           <uint32_t>;

// SS phase 1 register
reg ss_phase2_ff          <uint1_t>;

// execution status ff
reg start_comp_ff         <uint1_t>;
reg start_rd_ff           <uint1_t>;
reg rd_smp_ff             <uint1_t>;
reg pproc_smp_ff          <uint1_t>;
reg size_calc_ff          <uint1_t>;
reg end_jproc_ff          <uint1_t>;
reg wr_cdata_ff           <uint1_t>;
reg end_wr_cdata_ff       <uint1_t>;
```

```
// auxiliary registers for correct interface timing
reg acc_bsy               <uint1_t>;
reg ahb_pipe              <uint32_t>;
reg ahb_slv_wr_ncc_ff     <uint1_t>;
reg ahb_slv_op_ncc_ff     <uint1_t>;

// auxiliary transitories to communicate ahb actions
trn ahb_req_mem           <uint1_t>;
trn start_write           <uint1_t>;
trn start_read            <uint1_t>;
trn locked_idle           <uint1_t>;

// initial values for registers
hw_init smp_in_del = 0;
hw_init nbit = 0;
hw_init incW = 0;
hw_init totB = 0;
hw_init nZeros = 0;
hw_init pproc_smp_ff = 0;
hw_init size_calc_ff = 0;
hw_init all_0_block_ff = 1;
hw_init end_jproc_ff = 0;
hw_init wr_cdata_ff = 0;
hw_init ss_phase2_ff = 0;
hw_init cnt_j_wr = 0;
hw_init cnt_j_rd = 0;
hw_init cnt_smp = 0;
hw_init size_se = 0;
hw_init size_k = 0;
hw_init acc_bsy = 0;
hw_init ahb_pipe = 0;
hw_init ahb_slv_wr_ncc_ff = 0;
hw_init ahb_slv_op_ncc_ff = 0;
hw_init start_comp_ff = 0;
hw_init rd_smp_ff = 0;
hw_init start_rd_ff = 0;
hw_init cnt_r = 0;
hw_init end_wr_cdata_ff = 0;

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Process response from the accelerator
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

process process_result() {

  // default signal values
  haddr_mst_o = 0;
  hwrite_mst_o = "0";
  htrans_mst_o = "00";
  hsize_mst_o = "000";
  hlock_mst_o = "0";
  wrport_addr_acc_o = 0;
  wrport_datain_acc_o = 0;
  wrport_en_acc_o = "0";
  wrport_write_acc_o = "0";

  hbusreq_mst_o = ahb_req_mem;

  if(start_write){
    wrport_addr_acc_o = totB[31:2]::"00" + cdata_address;
```

```
    wrport_datain_acc_o = cdata_to_wr;
    wrport_en_acc_o = "1";
    wrport_write_acc_o = "1";
  }

  if(start_read){
    haddr_mst_o = samples_address + cnt_smp::"00";
    htrans_mst_o = "10";
    hsize_mst_o = "010";
    hlock_mst_o = "1";
  }
  else if(locked_idle){
    hlock_mst_o = "1";
  }

  if(acc_bsy){
    hready_slv_o = 0;
  }
  else if(ahb_slv_op_ncc_ff && !ahb_slv_wr_ncc_ff){
    hready_slv_o = 1;
    hresp_slv_o = 0;
    switch(ahb_pipe){
      case TOTB_ADDR:
        hrdata_slv_o = totB;
        break;
      case SPARE_BITS_ADDR:
        hrdata_slv_o = nbit[2:0];
        break;
      default:
        break;
    }
  }
  else{
    hready_slv_o = 1;
  }
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// ~~~ Process request from core
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


process process_request() {

  ahb_req_mem = 0;
  start_read = 0;
  start_write = 0;
  locked_idle = 0;
  uint32_t cdata_to_wr_tmp = 0;

  uint6_t j_bs_minus1 = j_blocksize-1;

  // ahb memory mapped loosely-coupled accelerator control
  ahb_slv_op_ncc_ff = 0;
  ahb_pipe = 0;
  if ((hsel_slv_i == "1") && (htrans_slv_i == "10") && (hsize_slv_i == "010") && hready_slv_i &&
  ↪  !acc_bsy){
    ahb_pipe = haddr_slv_i;
    ahb_slv_wr_ncc_ff = hwrite_slv_i;
    ahb_slv_op_ncc_ff = 1;
```

```
}

if(ahb_slv_op_ncc_ff && ahb_slv_wr_ncc_ff){
  switch (ahb_pipe){
    // load configuration registers
    case NBITS_ADDR:
      n_bits = hwdata_slv_i[5:0];
      break;
    case JSIZE_ADDR:
      j_blocksize = hwdata_slv_i[6:0];
      break;
    case RSINT_ADDR:
      r_samplesInterval = hwdata_slv_i[12:0];
      break;
    case PPACT_ADDR:
      preprocessor_active = hwdata_slv_i[0];
      break;
    case TOT_SMP_ADDR:
      tot_samples = hwdata_slv_i;
      break;
    case SAMPLE_ADDR:
      samples_address = hwdata_slv_i;
      break;
    case CDATA_ADDR:
      cdata_address = hwdata_slv_i;
      break;
    // accelerator operations
    case START_COMP_ADDR:
      if (hwdata_slv_i == 1){
        ahb_req_mem = 1;
        start_comp_ff = 1;
      }
      break;
    case RST_ADDR:
      if (hwdata_slv_i == 1){
        smp_in_del = 0;
        totB = 0;
        n_bits = 0;
        cnt_j_wr = 0;
        cnt_j_rd = 0;
        cnt_smp = 0;
        incW = 0;
        size_se = 0;
        v30_uint32_t size_k_tmp;
        for(int k = 0; k<30; k++){
          size_k_tmp[k] = 0;
        }
        size_k = size_k_tmp;
      }
      break;
    default:
      break;
  }
}

if(start_comp_ff){
  // accelerator becomes busy
  acc_bsy = 1;

  if(hgrant_mst_i == "1"){
    // start a new read and increment the sample counter
```

```
    start_read = 1;
    cnt_smp += 1;

    // process the first read and start the second one
    start_rd_ff = 1;
    rd_smp_ff = 1;

    // reset register
    start_comp_ff = 0;
  }
  else{
    // if the bus is not granted keep asking for it
    ahb_req_mem = 1;
  }
}


if(start_rd_ff && hready_mst_i){

  // start a new read and increment the sample counter
  start_read = 1;
  cnt_smp += 1;

  // update registers
  rd_smp_ff = 1;
  uint6_t reminder;
  switch (j_blocksize){
    case 8:
      reminder = (uint6_t) cnt_smp[2:0];
      break;
    case 16:
      reminder = (uint6_t) cnt_smp[3:0];
      break;
    case 32:
      reminder = (uint6_t) cnt_smp[4:0];
      break;
    case 64:
      reminder = cnt_smp[5:0];
      break;
    default:
      break;
  }
  if(reminder == j_bs_minus1){
    start_rd_ff = 0;
  }
  else{
    start_rd_ff = 1;
  }
}


if(rd_smp_ff){
  // insert the received sample in the pipe register
  smp_in = hrdata_mst_i;
  pproc_smp_ff = 1;

  // keep the control of the bus with locked idle transmission when read is ended
  locked_idle = !start_rd_ff;

  // update registers
  rd_smp_ff = start_rd_ff;
```

```
}


if(pproc_smp_ff){

  // preprocessor section
  ppsmp = UDP_PEM(smp_in, smp_in_del, preprocessor_active, w32_mask[n_bits]);
  smp_in_del = smp_in;

  // proceed to size calculation and start new read
  size_calc_ff = 1;

  // keep the control of the bus with locked idle transmission when read is ended
  locked_idle = !start_rd_ff;

  // update registers
  pproc_smp_ff = rd_smp_ff;
}

if(size_calc_ff){

  // check if pre-processed sample is 0 for zero block
  if(ppsmp != 0){
    all_0_block_ff = 0;
  }

  // store the preprocessed sample in J register
  wr_Jreg(ppsmp);

  // fundamental sequence and sample splitting section
  v30_uint32_t size_k_tmp;
  for(int k = 0; k<30; k++){
    size_k_tmp[k] = size_k[k] + (ppsmp>>k) + k + 1;
  }
  size_k = size_k_tmp;

  // second-extension section

  // H sample calculation
  uint32_t hsmp = get_hSample(ppsmp, ppsmp_del);
  if(cnt_j_wr[0]){
    size_se = size_se + hsmp + 1;

    //store H sample in H register
    wr_Hreg(hsmp);
  }

  // delayed pre-processed sample
  ppsmp_del = ppsmp;


  // check if the block has been fully analyzed
  if(cnt_j_wr == j_bs_minus1){
    cnt_r += 1;
    end_jproc_ff = 1;
    cnt_j_wr = 0;
  }
  else{
    cnt_j_wr = cnt_j_wr+1;
  }
```

```
  // keep the control of the bus with locked idle transmission when read is ended
  locked_idle = !start_rd_ff;

  // update registers
  size_calc_ff = pproc_smp_ff;
}

if(end_jproc_ff){

  // with dual-port the write is done on separate port, the bus must be hold
  locked_idle = 1;

  uint6_t cid_int_tmp;
  uint6_t cid_nc_tmp;
  uint6_t compression_identifier;
  uint3_t compression_identifier_size;

  if (n_bits < 3){
    compression_identifier_size = 1;
    cid_nc_tmp = 0x1;
  }
  else if (n_bits < 5){
    compression_identifier_size = 2;
    cid_nc_tmp = 0x3;
  }
  else if (n_bits <= 8){
    compression_identifier_size = 3;
    cid_nc_tmp = 0x7;
  }
  else if (n_bits <= 16){
    compression_identifier_size = 4;
    cid_nc_tmp = 0xF;
  }
  else{
    compression_identifier_size = 5;
    cid_nc_tmp = 0x1F;
  }

  // find leading one
  v15_uint1_t diff_signs_vector;
  for(int k = 0; k < 15; k++){
    uint32_t diff = size_k[2*k] - size_k[2*k+1];
    diff_signs_vector[k] = diff[31];
  }
  uint4_t leading1 = find_leading_1(diff_signs_vector);

  // find best sample-splitting (0 == fundamental sequence)
  uint5_t candidate1 = (uint5_t) leading1::"0";
  uint5_t candidate2 = ((uint5_t) leading1::"0") - 1;
  uint5_t candidate3 = candidate2[4:1]::"0";
  uint5_t index_smallest;
  if(leading1 == 0){
    index_smallest = 0;
  }
  else if((size_k[candidate3] <= size_k[candidate2]) && (size_k[candidate3] <=
  ↪  size_k[candidate1])){
    index_smallest = candidate3;
  }
  else if(size_k[candidate2] <= size_k[candidate1]){
    index_smallest = candidate2;
  }
```

```
else{
  index_smallest = candidate1;
}
uint5_t index_size_lk = (index_smallest <= n_bits-3) ? index_smallest : n_bits-3;
size_lk = size_k[index_size_lk];

// no copression size
uint32_t size_nc = j_blocksize * n_bits;

// find best compression method
if(all_0_block_ff){
  // for ZB J and H mem are not needed
  // if not ended samples start a new read
  start_rd_ff = (cnt_smp != tot_samples);

  cid_int_tmp = ZB_CID_INT;
  compression_identifier = 0;
}
else if (size_nc <= size_se && size_nc <= size_lk){
  // writing has same timing than reading, since writing starts earlier
  // there is no risk of loosing data not yet compressed
  start_rd_ff = (cnt_smp != tot_samples);

  nZeros = 0;
  cid_int_tmp = NC_CID_INT;
  compression_identifier = cid_nc_tmp;
}
else if(size_se < size_nc && size_se <= size_lk){
  // writing has same timing than reading, since writing starts earlier
  // there is no risk of loosing data not yet compressed
  start_rd_ff = (cnt_smp != tot_samples);

  nZeros = 0;
  cid_int_tmp = SE_CID_INT;
  compression_identifier = 1;
}
else{
  // if FS start reading, if SS wait to finish phase 1
  start_rd_ff = (index_size_lk == FS_CID_INT) && (cnt_smp != tot_samples);

  nZeros = 0;
  cid_int_tmp = index_size_lk;
  compression_identifier = index_size_lk+1;
}

cid_int = cid_int_tmp;


// If the selected technique is ZB or the SE, the compression_identifier_size is +1
if(cid_int_tmp == ZB_CID_INT || cid_int_tmp == SE_CID_INT){
  compression_identifier_size += 1;
}

// write compression identifier in memory
uint32_t tmp_cdata = getWordToWrite(compression_identifier, compression_identifier_size);
cdata_to_wr_tmp = change_endian(tmp_cdata);
start_write = 1;

// start compressed samples writing process
wr_cdata_ff = 1;
```

```
  // if a R-interval has been analyzed, reset the predictor
  if(cnt_r == r_samplesInterval){
    cnt_r = 0;
    smp_in_del = 0;
  }

  // reset registers
  all_0_block_ff = 1;
  end_jproc_ff = 0;
  size_se = 0;
  v30_uint32_t size_k_tmp;
  for(int k = 0; k<30; k++){
    size_k_tmp[k] = 0;
  }
  size_k = size_k_tmp;
}

if(wr_cdata_ff){

  // with dual-port the write is done on separate port, the bus must be hold
  // ininfluent if the new reading has started
  locked_idle = 1;

  uint32_t tmp_cdata;

  // zero block
  if(cid_int == ZB_CID_INT){
    tmp_cdata = get_fs_word(nZeros);

    nZeros += 1;

    // end write procedure
    end_wr_cdata_ff = (cnt_smp == tot_samples);

    // reset status ff and counters
    wr_cdata_ff = 0;
    cnt_j_rd = 0;
  }

  // second extension
  else if(cid_int == SE_CID_INT){
    smp_to_wr = rd_Hreg();
    if(!size_calc_ff){
      wr_Hreg(0);
    }
    tmp_cdata = getWordToWrite(smp_to_wr, 32);
    if(cnt_j_rd == ("00"::j_blocksize[5:1])-1){

      // end write procedure
      end_wr_cdata_ff = (cnt_smp == tot_samples);

      // reset status ff and counters
      wr_cdata_ff = 0;
      cnt_j_rd = 0;
    }
    else{
      cnt_j_rd = cnt_j_rd+1;
    }
  }

  // no compression
```

```
else if(cid_int == NC_CID_INT){
  smp_to_wr = rd_Jreg();
  if(!size_calc_ff){
    wr_Jreg(0);
  }
  tmp_cdata = getWordToWrite(smp_to_wr, n_bits);
  if(cnt_j_rd == j_bs_minus1){

    // end write procedure
    end_wr_cdata_ff = (cnt_smp == tot_samples);

    // reset status ff and counters
    wr_cdata_ff = 0;
    cnt_j_rd = 0;
  }
  else{
    cnt_j_rd = cnt_j_rd+1;
  }
}

// fundamental sequence
else if(cid_int == FS_CID_INT){
  smp_to_wr = rd_Jreg();

  tmp_cdata = get_fs_word(smp_to_wr);
  if(cnt_j_rd == j_bs_minus1){
    // end write procedure
    end_wr_cdata_ff = (cnt_smp == tot_samples);

    // reset status ff and counters
    wr_cdata_ff = 0;
    cnt_j_rd = 0;
  }
  else{
    if(!size_calc_ff){
      wr_Jreg(0);
    }
    cnt_j_rd = cnt_j_rd+1;
  }
}

// sample splitting
else{
  smp_to_wr = rd_Jreg();
  uint32_t shifted_smp = smp_to_wr >> cid_int;
  if(!ss_phase2_ff){
    tmp_cdata = get_fs_word(shifted_smp);
  }
  else{
    tmp_cdata = getWordToWrite(smp_to_wr, cid_int);
  }
  if(cnt_j_rd == j_bs_minus1){
    if(!ss_phase2_ff){

      cnt_j_rd = 0;
      // start phase 2 of sample splitting writing
      wr_Jreg(smp_to_wr);
      ss_phase2_ff = 1;

      // here also new reading can start
      // writing has same timing than reading, since writing starts
```

```
          // earlier there is no risk of loosing data not yet compressed
          start_rd_ff = (cnt_smp != tot_samples);
        }
        else{

          // end write procedure
          end_wr_cdata_ff = (cnt_smp == tot_samples);

          // reset status ff and counters
          cnt_j_rd = 0;
          ss_phase2_ff = 0;
          wr_cdata_ff = 0;
        }
      }
      else{
        if(!size_calc_ff){
          wr_Jreg(smp_to_wr);
        }
        cnt_j_rd = cnt_j_rd+1;
      }
    }

    // start the next writing in the memory
    cdata_to_wr_tmp = change_endian(tmp_cdata);
    start_write = 1;

  }


  if(end_wr_cdata_ff){
    // this also works as the recommended
    // idle cycle after end of locked transmission

    // write the remaining incW content in memory and update totB
    cdata_to_wr_tmp = change_endian(incW);
    start_write = 1;
    totB += nbit[5:3];

    // signals the end of the processing
    acc_bsy = 0;

    // reset register
    end_wr_cdata_ff = 0;
  }

  cdata_to_wr = cdata_to_wr_tmp;
  //wr_ncc_ff = start_write;
}

void wr_Jreg(uint32_t smp_to_write){
  v64_uint32_t regJ_tmp;
  regJ_tmp[0] = smp_to_write;
  for(int i = 0; i < 63; i++){
    regJ_tmp[i+1] = regJ[i];
  }
  regJ = regJ_tmp;
}

uint32_t rd_Jreg(){
  uint32_t j_out;
  switch(j_blocksize){
```

```
    case 8:
      j_out = regJ[7];
      break;
    case 16:
      j_out = regJ[15];
      break;
    case 32:
      j_out = regJ[31];
      break;
    case 64:
      j_out = regJ[63];
      break;
  }
  return j_out;
}

void wr_Hreg(uint32_t smp_to_write){
  v32_uint32_t regH_tmp;
  regH_tmp[0] = smp_to_write;
  for(int i = 0; i < 31; i++){
    regH_tmp[i+1] = regH[i];
  }
  regH = regH_tmp;
}

uint32_t rd_Hreg(){
  uint32_t h_out;
  switch(j_blocksize){
    case 8:
      h_out = regH[3];
      break;
    case 16:
      h_out = regH[7];
      break;
    case 32:
      h_out = regH[15];
      break;
    case 64:
      h_out = regH[31];
      break;
  }
  return h_out;
}

uint32_t UDP_PEM(int32_t new_sample, int32_t PredictedValue, uint1_t preprocessor_active,
↪  uint32_t x_max){
  if(preprocessor_active){
      int32_t PredictionError = new_sample - PredictedValue;
      int32_t difference = x_max - PredictedValue;
      int32_t theta = (PredictedValue < difference) ? PredictedValue : difference;
      int32_t PredictionErrorAbs = (PredictionError > 0) ? PredictionError : -PredictionError;
      int32_t PreprocessedSample;

      if(0 <= PredictionError && PredictionError <= theta)
      {
          PreprocessedSample = 2*PredictionError;
      }
      else if(-theta <= PredictionError && PredictionError < 0)
      {
          PreprocessedSample = 2*PredictionErrorAbs -1;
      }
```

```
      else{
          PreprocessedSample = theta + PredictionErrorAbs;
      }

      return PreprocessedSample;
  }
  return new_sample;
}


uint32_t get_hSample(uint32_t sample1, uint32_t sample2){
  return (uint32_t) ( ((sample1 + sample2) * (sample1 + sample2 + 1)) >> 1) + sample2;
}

uint32_t get_fs_word(uint32_t data_length){
  uint32_t word_to_write;

  uint32_t new_val_bits_minus1 = nbit + data_length;

  uint27_t n_words = new_val_bits_minus1[31:5];
  if(n_words == 0){
    uint6_t n_zeros = "0"::data_length[4:0];
    word_to_write = getWordToWrite(1, n_zeros+1);
  }
  else{
    word_to_write = incW;
    totB += n_words::"00";

    uint8_t new_nbit = "0"::new_val_bits_minus1[4:0] + 1;

    incW = (uint32_t) 0x80000000 >> new_val_bits_minus1[4:0];
    nbit = new_nbit;

  }
  return word_to_write;
}

uint4_t find_leading_1(v15_uint1_t diff_signs_vector){
  uint4_t l1 = 14;
  uint1_t found = 0;
  for(uint4_t i = 0; i < 15; i++){
    if(diff_signs_vector[i] && !found){
      l1 = i;
      found = 1;
    }
  }
  return l1;
}

uint32_t getWordToWrite(uint32_t data_i, uint6_t data_length){
  uint64_t buffer = incW::(uint32_t)0;
  uint32_t data = data_i & w32_mask[data_length];
  uint8_t bits_in_buffer = (uint8_t) nbit + (uint8_t) data_length;

  buffer |= (uint64_t) (((uint64_t) data) << (64 - bits_in_buffer));


  if (bits_in_buffer < 32){
    incW = buffer[63:32];
    nbit = (uint5_t) bits_in_buffer;
  }
```

```
    else{
      incW = buffer[31:0];
      nbit = bits_in_buffer - 32;
      totB = totB + 4;
    }
    return buffer[63:32];
  }

  uint32_t change_endian(uint32_t a){
    return a[7:0]::a[15:8]::a[23:16]::a[31:24];
  }


  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ~~~ Debug Access
  // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  void dbg_access_LCA(AddressType a, w08& val, bool read) {
    // not implemented
  }
}
```

Code B.2: PDG implementation of LCA

Bibliography

[1] Consultative Committee for Space Data Systems (CCSDS). *CCSDS 121.0 Lossless Data Compression*. Tech. rep. CCSDS 121.0-B-3. Blue Book – Issue 3. CCSDS, Nov. 2022. URL: https://public.ccsds.org/Pubs/121x0b3.pdf (cit. on p. 4).

[2] Consultative Committee for Space Data Systems (CCSDS). *CCSDS 120.0-G-3 Informational Report*. Tech. rep. CCSDS 120.0-G-3. Green Book – Issue 3. CCSDS, Apr. 2023. URL: https://public.ccsds.org/Pubs/120x0g3s.pdf (cit. on p. 4).

[3] OBPMark. *On-Board Processing Benchmark Suite*. 2024. URL: https://obpmark.github.io/ (cit. on p. 5).

[4] RISC-V Foundation. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. Version 20240411. Available at https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications. 2024 (cit. on p. 6).

[5] Synopsys, Inc. *ASIP Designer – Application-Specific Instruction-Set Processor Development Tool*. 2024. URL: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer (cit. on pp. 6, 8).

[6] Jan Andersson. «Development of a NOEL-V RISC-V SoC Targeting Space Applications». In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 66–67. DOI: 10.1109/DSN-W50199.2020.00020 (cit. on p. 9).

[7] Jan Andersson, Magnus Hjorth, Fredrik Johansson, and Sandi Habinc. «LEON Processor Devices for Space Missions: First 20 Years of LEON in Space». In: *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. 2017, pp. 136–141. DOI: 10.1109/SMC-IT.2017.31 (cit. on p. 9).

[8] Cobham Gaisler. *GRLIB IP Library User's Manual*. 2023. URL: https://download.gaisler.com/products/GRLIB/doc/grip.pdf (cit. on pp. 9, 17).

[9] ARM Ltd. *AMBA Specification (Rev. 2.0)*. 2022. URL: https://developer.arm.com/documentation/ihi0011/latest/ (cit. on pp. 10, 11).

[10] Cobham Gaisler. *GRLIB IP Library User's Manual*. 2024. URL: https://download.gaisler.com/products/GRLIB/doc/grlib.pdf (cit. on p. 11).

[11] Cobham Gaisler. *NOEL-V Bare-Metal Cross Compiler*. 2025. URL: https://www.gaisler.com/products/noel-bare-metal-cross-compiler (cit. on p. 36).

[12] GNU Project. *GCC, the GNU Compiler Collection*. 2025. URL: https://gcc.gnu.org/ (cit. on p. 36).

[13] Cobham Gaisler. *LEON Bare-Metal Cross Compiler*. 2025. URL: https://www.gaisler.com/products/leon-bare-metal-cross-compiler (cit. on p. 39).

[14] Consultative Committee for Space Data Systems (CCSDS). *Lossless Multispectral & Hyperspectral Image Compression*. CCSDS 123.0-B-2, Blue Book, Issue 2. Nov. 2022. URL: https://public.ccsds.org/Pubs/123x0b2e2c3.pdf (cit. on p. 53).

[15] Miguel Hernández-Cabronero, Aaron B. Kiely, Matthew Klimesh, Ian Blanes, Jonathan Ligo, Enrico Magli, and Joan Serra-Sagristà. «The CCSDS 123.0-B-2 "Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression" Standard: A comprehensive review». In: *IEEE Geoscience and Remote Sensing Magazine* 9.4 (2021), pp. 102–119. DOI: 10.1109/MGRS.2020.3048443 (cit. on p. 53).

[16] Yubal Barrios, Antonio J. Sánchez, Lucana Santos, and Roberto Sarmiento. «SHyLoC 2.0: A Versatile Hardware Solution for On-Board Data and Hyperspectral Image Compression on Future Space Missions». In: *IEEE Access* 8 (2020), pp. 54269–54287. DOI: 10.1109/ACCESS.2020.2980767 (cit. on pp. 53, 68, 69).

[17] Alvaro Jover-Alvarez, Ivan Rodriguez, Leonidas Kosmidis, and David Steenari. «Space Compression Algorithms Acceleration on Embedded Multi-core and GPU Platforms». In: *Ada Lett.* 42.1 (Dec. 2022), pp. 100–104. ISSN: 1094-3641. DOI: 10.1145/3577949.3577969. URL: https://doi.org/10.1145/3577949.3577969 (cit. on p. 68).

[18] European Space Agency (ESA). *SHyLoC Datasheet v1.0*. 2021. URL: https://amstel.estec.esa.int/tecedm/ipcores/SHyLoC_Datasheet_v1.0.pdf (cit. on p. 68).

[19] Samuel Torres-Fau, Antonio J. Sánchez, Yubal Barrios, and Roberto Sarmiento. «CCSDS121-based High-Performance Hardware Architecture for Real-Time Data Compression». In: *2023 European Data Handling & Data Processing Conference (EDHPC)*. 2023, pp. 1–8. DOI: 10.23919/EDHPC59100.2023.10395929 (cit. on pp. 68, 69).

[20] L. Miles et al. «Over 3 Gb/s Universal Lossless Compressor for Space Use». In: *Proceedings of the ReSpace/MAPLD 2011 Conference*. New Mexico, USA, 2011 (cit. on pp. 68, 69).

[21] Nektarios Kranitis, Ioannis Sideris, Antonis Tsigkanos, Georgios Theodorou, Antonis Paschalis, and Raffaele Vitulli. «Efficient field-programmable gate array implementation of CCSDS 121.0-B-2 lossless data compression algorithm for image compression». In: *Journal of Applied Remote Sensing* 9 (May 2015), p. 097499. DOI: 10.1117/1.JRS.9.097499 (cit. on pp. 68, 69).

[22] Ian Blanes, Aaron Kiely, Miguel Hernández-Cabronero, and Joan Serra-Sagristà. «Performance Impact of Parameter Tuning on the CCSDS-123.0-B-2 Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression Standard». In: *Remote Sensing* 11.11 (2019). ISSN: 2072-4292. DOI: 10.3390/rs11111390. URL: https://www.mdpi.com/2072-4292/11/11/1390 (cit. on p. 69).

[23] Antonio Sánchez, Ian Blanes, Yubal Barrios, Miguel Hernández-Cabronero, Joan Bartrina-Rapesta, Joan Serra-Sagristà, and Roberto Sarmiento. «Reducing Data Dependencies in the Feedback Loop of the CCSDS 123.0-B-2 Predictor». In: *IEEE Geoscience and Remote Sensing Letters* 19 (2022), pp. 1–5. DOI: 10.1109/LGRS.2022.3213975 (cit. on p. 69).

[24] Ruben Picos and Sorin Olaru. «Optimized multicore implementation and benchmark for on-board image compression algorithm - on dedicated space HW». In: *Data Systems In Aerospace (DASIA) 2018*. European Space Agency (ESA). 2018. URL: https://indico.esa.int/event/239/contributions/2247/attachments/1848/2151/1100b_-_Picos__Olaru.pdf (cit. on p. 69).