POLITENICO DI TORINO

Corso di Laurea in
Ingegneria Elettronica

Tesi di Laurea Magistrale

Turbo decoding algorithm parallelization

Relatori
prof. Guido Masera
prof. Maurizio Martina

Candidato
Alessandro Bruscia

Anno Accademico 2018/2019

Ai miei genitori
per avermi supportato
in ogni istante senza
nessuna esitazione, anche
nei momenti più difficili
e a Giulia perché l'amore
supera i confini di una
vita

Acknowledgements

Ringrazio la mia famiglia per essermi stata vicina ogni giorno nonostante il mio carattere non facile. Ringrazio Mabel per avermi tranquillizzato durante i momenti in cui l'ansia stava per prendere il sopravvento. Ringrazio i miei colleghi per tutti i momenti di disperazione condivisi che ci hanno aiutato a crescere insieme. Infine ringrazio me stesso per non aver mai smesso di crederci anche quando le difficoltà sembravano insormontabili.

Contents

Chapter 1

Introduction

The goal of this master thesis is to evaluate the performance of a turbo decoding algorithm on a general purpose platform based on the open-source RISC-V instruction set architecture(ISA). The challenge is to modify the algorithm in order to reach a high throughput exploiting a parallel hardware platform. The turbo code is an error correction code used to encode an information that has to be transmitted on a communication channel. The decoding of the turbo encoded informations requires very complex operations and consequently it is difficult to build a decoder able to achieve a good throughput. Since the communication of informations requires very high speed many researchers are trying to increase the throughput of the decoding by creating high speed decoder architectures. Obviously this goal can be achieved easily by using a non programmable ASIC or a DSP but both architectures are characterized by following disadvantages [2]:

- High implementation cost
- Low portability
- long design cycles

Besides the ASIC implementation has very low flexibility that causes some problems when there is a communication between devices that exploit different protocols with different frame structures [1].

All the problems seen for DSP and ASIC implementations can be overcome by exploiting a general purpose architecture in which the turbo decoding algorithm can be implemented in software. The problem of a software implementation is that the common processors usually are not built to implement communication algorithms [3]. Consequently they are not able to provide the same throughput of a DSP or ASIC implementation. Two solutions proposed to overcome this problem can be found in [23] and [24]. Both exploits the big number of cores of a NVIDIA's GPU and some parallelization techniques to obtain a parallel computation of the turbo decoding algorithm. In this thesis a similar approach is exploited, what change is the platform on which the algorithm is mapped. The general purpose architecture used is called PULP. It is a microcontroller on which a cluster of *RISCV* processors is mounted. These processing elements can be exploited to implement a parallel configuration of a turbo decoding algorithm exploiting the utilities provided with the PULP system.

The standard turbo decoding algorithm was already implemented in C, but it was not already executable on PULP in a parallel way. Consequently the starting C program has been modified in order to implement the parallelization techniques of the turbo decoding algorithm with the PULP utilities.

The platform on which the algorithm has been mapped, is written in system verilog. It exploits a toolchain to map any C program on the platform (the installation steps are explained in subsection 2.3.1) and a development kit in which the functions necessary to exploit all the features of PULP system are defined (the installation steps are explained in subsection 2.3.2).

The algorithm has been mapped also to an other platform in which only one core is present. This system is called PULPino and it is a single core microcontroller that has a structure similar to PULP. They belongs to the same family called *PULP-platform*. Also PULPino is written in system verilog and has a toolchain necessary to map the C code on the system (installation steps can be found in section 2.2).

To simulate the turbo decoding application on PULPino and PULP QuestaSim 10.6c_1 has been used. Instead to synthesize PULP, as explained in chapter 5, Synopsys Design Compiler has been used.

The work has been divided in the following phases:

- The first phase consisted of a research of a multi core system based on a RISCV processor, characterized by a toolchain and a software utilities able to map easily a C program on the chosen platform.
- In the second step PULPino, PULP and their toolchains have been installed. In order to do this some tutorials have been followed for both systems. Some tips have been added in order to bypass some critical problems during the PULPino and PULP installations.
- In this third phase a behavioural study of a turbo decoding algorithm has been carried on. The goal was to understand the basic functioning of the algorithm and how it could be implemented in parallel in order to achieve a higher throughput.
- In the fourth phase the algorithm was mapped on the two platforms without exploiting any parallel computation.
- In the fifth step the utilities provided by the PULP development kit were exploited in order to compute in parallel the turbo decoding algorithm.
- In the sixth phase the PULP system has been synthesized in order to obtain its frequency and consequently evaluate the throughput in the next step.
- PULP and PULPino performance evaluation were carried on in the seventh step and a comparison with the reference designs has been done.
- The last phase included the conclusions and the possible improvements that could be applied to the software and hardware designs adopted in this thesis.

The chapters follow the phases described above. In particular this thesis is divided in seven chapters:

- The first chapter is the Introduction in which a general overview of the thesis is given. In particular the context in which the work is developed, its goals, methods and instruments used, the state of art used as reference, the phase in which the work is organised and the thesis structure are presented.
- The second chapter gives a general overview of the two systems adopted to implement the algorithm (PULP and PULPino) and how they are installed.
- The third chapter explains the main guidelines of the turbo code, a specific algorithm implementation of a turbo decoder, a complexity reduction and a parallel organization of the algorithm.
- In the fourth chapter the turbo decoding algorithm is mapped on PULPino and PULP. In particular for PULP the utilities to obtain a parallel implementation of the algorithm are defined and used following the parallelization techniques explained in the second chapter. In the end how the performance of the algorithm were evaluated is explained.
- In the fifth chapter the synthesis of PULP is analysed in order to find its maximum achievable frequency.
- The sixth chapter is divided in three parts: in the first two the throughput obtained with PULPino and PULP is shown, in the third a comparison between the results obtained and the considered state of art designs is carried on.
- In the end in the seventh part the conclusion and the possible software and hardware further improvements are presented.

Chapter 2

PULP-PLATFORM

The *PULP-PLATFORM* provides different microcontrollers but only the following two will be used:

- PULPino is an open-source configurable RISC-V microcontroller. Its core, developed by the ETH Zurich, exploits a 32-bit RISC-V ISA. In particular it is possible to choose two different cores: the RI5CY or the zero-riscy core [5].
- PULP is an open-source multi-core RISC-V microcontroller with a configurable cluster of processors in which the number of processing elements can range from one to sixteen. Also in this case it is possible to choose two different cores: the RI5CY or the zero-riscy core [9].

In both cases the RI5CY core is used.

2.1 RISC-V

RISC-V is a new open-source instruction set architecture that has the following characteristics:

- It allows to implement in hardware the designed objects.
- It is based on a RISC simple ISA that can be extended in order to implement more complex instructions.
- It allows parallel implementations exploiting more than one core.

Now the question is: why has it decided to use RISC-V despite the existence of commercial ISAs?
The reasons are [4]:

- ISAs created by a specific company usually are protected by the copyright. This makes difficult any kind of research.
- Commercial ISAs usually are created to meet certain needs and consequently they are able to offer a good support to the user only in certain sectors.

- The commercial ISAs could be abandoned by the creators and due to the copyright nobody could be able to support that ISA.
- Often it is not possible to use only the user level ISA to implement an application with a commercial ISA.
- The inclusion of new instructions in a popular ISA can complicate the instruction encoding.

Many researchers and companies are working on RISC-V and many processors based on this ISA have been created. The PULP platform was chosen between them, because is one of the most reliable and complete system available. Besides it includes many microcontrollers with increasing complexity in which the CPUs are based on the RISC-V ISA.

The RISC-V ISA is mainly divided in two parts: the base integer ISA and the extensions. The first part is the kernel of the RISC-V ISA, its functionalities are based on the common RISC ISAs plus some other features like the variable-width instructions. According to the instruction length it is possible to choose between two different integer ISAs: *RV32I* characterized by an instruction length equal to 32 bits and *RV64I* that provides 64 bits instruction length. The *RV32I* is the one used by the RI5CY that is the processor used by *PULP* and *PULPino*. The base ISA can be exploited to implement a simple computer without using any extension. This means that the base ISA is really independent from the extensions and this highlights one of the RISCV ISA best quality, the modularity. It has to be clear that the extensions can be used to customize the ISA in order to obtain an application specific ISA, but the base integer can not be change.

For what concern the extensions, they can be divided in two macro-categories:

- Standard extensions: they are well integrated with the base integer ISA and with the other extensions. In particular their usage guarantees no strife with the base integer and other extensions of the same type.
- Non-standard extensions: they can be used for systems that needs highly specified instructions. Some conflicts can happen between standard and non-standard instructions.

As said before it is possible to use instructions with a variable-length format. They are usually organized in blocks of sixteen bits. These instructions can be introduced only with the extensions because the integer part of the ISA is characterized by instructions represented on thirty-two bits. The approach used to store the instructions in the memory is called little endian that is also the one used by Intel. In particular the instruction is divided in parts of sixteen bits. Then it is stored in memory starting from the least significant part. In other words the least sixteen bits are stored in memory and shifted toward right. Then the next sixteen bits are stored in the upper part of the memory location. The next parts of the instruction are stored in the following memory locations by using the same method explained before.

The base ISA used for the RIC5Y is called *RV32I* it exploits thirty-two registers. It is characterized by mainly the following types of instructions: register-register , immediate, register-immediate, memory access, thread control, control flow etc. The format of some instructions is shown in table 2.1. Between them there is a particular instruction called FENCE, it is used when a synchronization between many threads running in the system is necessary. In particular each thread follows its instruction sequence and make its own memory operations. The FENCE allows to execute the instructions placed after the FENCE only when the memory access place before the FENCE are completed.

Further informations on the RISCV ISA can be found in [4].

Table 2.1. Instructions format [4]

Family	Instr-type								
Register	R-type	func7(7 bits)		rs3 (5 bits)		rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode (7 bits)
Imm	I-type	Imm (12 bits)				rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode (7 bits)
	S-type	Imm (7 bits)		rs2 (5 bits)		rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode (7 bits)
	B-type	Imm (1 bit)	Imm (6 bit)	rs2 (5 bits)	rs1 (5 bits)	funct3 (3 bits)	Imm (4 bits)	Imm (1 bit)	opcode (7 bits)
	U-type	Imm (20 bits)						rd (5 bits)	opcode (7 bits)
Control flow	Branch	Imm (1 bit)	Imm(6 bits)	rs2 (5 bits)	rs1 (5 bits)	funct 3 (3 bits)	Imm (4 bits)	Imm (1 bit)	opcode (7 bits)
	Jump	Imm (12 bits)				rs1 (5 bits)	funct 3 (3 bits)	Imm (4 bits)	opcode (7 bits)
Memory	Load	Imm (12 bits)				rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode (7 bits)
	Store	Imm (7 bits)		rs2 (5 bits)		rs1 (5 bits)	funct3 (3 bits)	Imm (5 bits)	opcode (7 bits)
Thread control	FENCE	0 (4 bits)	PI PO PR PW SI SO SR SW (8 bits)			rs1 (5 bits)	funct3 (3 bits)	rd (5 bits)	opcode (7 bits)

2.2 PULPino

PULPino is the first system used in this thesis, it is characterized by two single ports RAMs, one for the data and the other for the instructions. The boot code is placed in a ROM memory. The boot loader can use the SPI to load a program from an external device. The main SoC interconnections are based on the AXI protocol instead for simple peripherals the APB one is used. Both the AXI and APB buses feature 32 bits wide data channels. For debug purpose it is possible to read all core registers present in the architecture exploiting the AXI bus [5]. Its block diagram is shown in the figure 2.1.

In order to run C applications on PULPino it is necessary to satisfy some requirements:

- ModelSim/QuestaSim in a version equal to or higher than 10.2c.
- CMAKE in a version equal to or higher than 2.8.0.
- Python2 in a version equal to or higher than 2.6.
- Install riscv-toolchain. In particular the custom RISCV toolchain from ETH is suggested. The ETH toolchain allows to exploit all the RI5CY ISA extensions. It can be downloaded at the link indicated in [8]. Since RIC5Y is used any change has to be

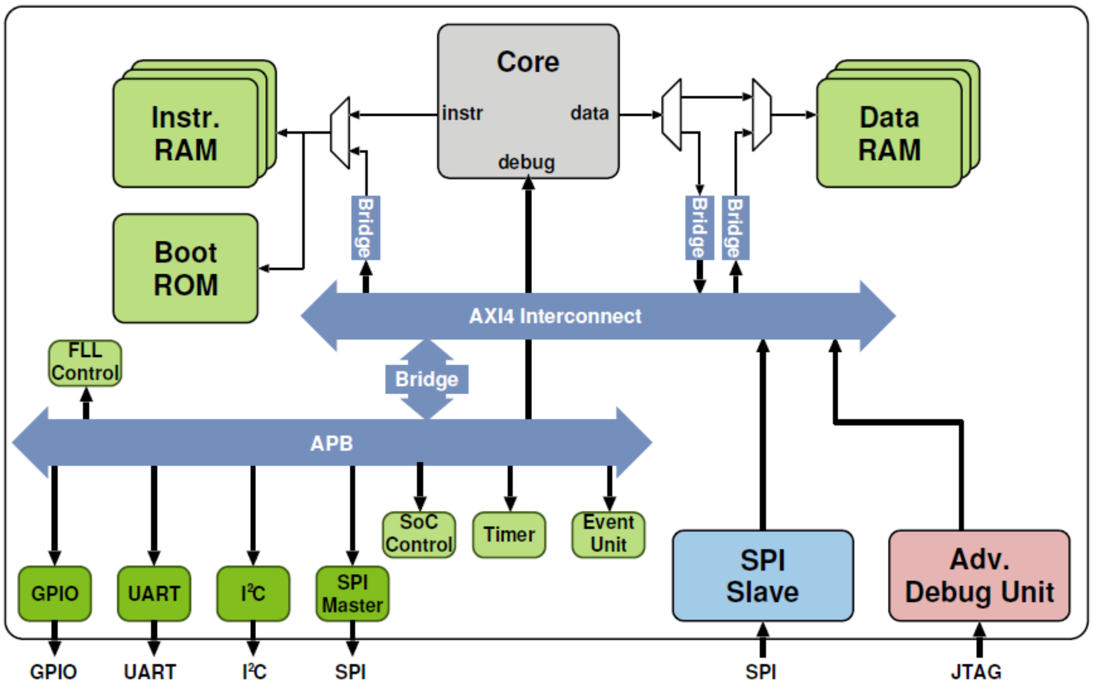

Figure 2.1. PULPino block diagram [7]

done to the downloaded files. To build the toolchain the command make has to be issued inside the folder that includes the toolchain itself. Some packages necessary to conclude the installation are: texinfo, g++, libgmp-dev, libmpc-dev and libmpfr-dev.

PULPino is organized in submodules by means of git. In order to download the submodules the following command has to be issued *./update-ips.py*. Between the files fetched with this command there are, for example, the files related to the processors architecture. It is possible to add other features by selecting specific commits, tags or branches. Now, in order to simulate the C applications, it is necessary to create a new folder called *build*. It could be created for example in the *sw* folder. Then one of the scripts cmake-configure.*.gcc.sh has to be copied in the build folder and executed there. These scripts are placed in the *sw* folder and are used to initialize whatever is necessary to simulate with Modelsim [5]. The available scripts are:

- *cmake_configure.riscv.gcc.sh* by means of which the RIC5Y core is chosen and the environment is set in order to support the PULP-extensions and the RV32IM ISA. The GCC march flag has to be assigned to *IMXpulpv2*

- *cmake_configure.riscvfloat.gcc.sh* by means of which the RIC5Y core is chosen and the environment is set in order to support the PULP-extensions and the RV32IMF ISA. In this case floating point support is added.The GCC march flag has to be assigned to *IMFXpulpv2*

- *cmake_configure.zeroriscy.gcc.sh* by means of which the zero-riscy core is chosen and the environment is set in order to support the RV32IM ISA. The GCC march flag has to be assigned to *RV32IM*
- *cmake_configure.microriscy.gcc.sh* by means of which the zero-riscy core is chosen and the environment is set in order to support the RV32E ISA. The GCC march flag has to be assigned to "RV32I". Besides to force the usage of the simplified RV32E ISA *-m16r* is provided to the compiler.

The first one was used during the system configuration.

Subsequently, ModelSim/QuestaSim is used to analyze the RTL libraries. This is done by issuing the command *make vcompile* inside the build folder. If ModelSim/QuestaSim used is a 32 bit version the flag -64 has to be removed from the *sw/apps/CMakeSim.txt.* Now the simulation of the designed application can be started by issuing *make application_name.vsim* (for example *make helloworld.vsim*) to execute a simulation in the Modelsim graphical user interface or *make application_name.vsimc* (for example *make helloworld.vsimc*) to execute a simulation in the ModelSim console [5]. If ModelSim uses it's native gcc instead of the toolchain one, it is necessary to insert the line *-dpicpppath /usr/bin/gcc* at the end of *cmd* in the file *vsim/tcl_files/config/vsim.tcl* as shown below:

```
set cmd "vsim -quiet $TB
-L pulpino_lib
$VSIM_IP_LIBS
+nowarnTRAN
+nowarnTSCALE
+nowarnTFMPC
+MEMLOAD=$MEMLOAD
-gUSE_ZERO_RISCY=$env(USE_ZERO_RISCY)
-gRISCY_RV32F=$env(RISCY_RV32F)
-gZERO_RV32M=$env(ZERO_RV32M)
-gZERO_RV32E=$env(ZERO_RV32E)
-t ps
-voptargs=+̈acc -suppress 2103¨
$VSIM_FLAGS -dpicpppath /usr/bin/gcc"
```

At the end of these steps the applications included in the PULPino folder should run correctly. Some additional work is necessary to run an external program: a new folder, in which the C program will be copied, has to be created in the path *sw/apps.* The files are managed by using *CMake* that exploits the *CMakeLists.txt* files to organize the application folders. In each of these a *CMakeLists.txt* has to be created and structured like the following example:
add_application(helloworld helloworld.c)
in which *helloworld* is the name used when we run the application and *helloworld.c* is the name of the program [5]. If the application exploits more than one source file the *CMakeLists.txt* has to be organized in this way:
set(SOURCES main_file.c other_file.c)
add_application(application_name "$SOURCES")

Also the *CMakeLists.txt* in the *sw/apps* has to be modified including the line *add_subdirectory*, in this way *CMake* becomes aware of the new application folders (for example *add_subdirectory(application_folder_name)*). If it is necessary to use a single source folder for more than one application, the additional argument *SUBDIR* of the command *add_application* has to be used. This guarantees that each program has its own folder in the build directory. An example can be *add_application(new_app new_app.c SUBDIR "new"))* in which the application *new_app* will be put in a subdirectory called *new* in the build folder structure. If some additional libraries are required it is necessary to:

- create a new folder called *new_lib* in which *new* is the name of the new library at the path */sw/libs*. Inside the *new_lib* directory two subfolders have to be placed: *src* and *inc*. In the first one there are the source files, instead in the second one the header files.
- In the library folder a CMakeLists.txt file has to be created. It has to be organized as shown below:

  ```
  set(SOURCES
  src/souce_file.c
  src/source_file.c
  )
  set(HEADERS
  inc/header_file.h
  inc/header_file.h
  )
  include_directories(inc/)
  add_library(Library_name STATIC ${SOURCES} ${HEADERS})
  ```

- The CMakeLists.txt files present in the paths */sw/apps* and */sw* have to be modified. The bold sentences indicated in *appendices* A.1 and A.2 [5] have to be modified in order to take into account the new libraries. The name of the new library has to be inserted where the statement *new* is placed.

2.3 PULP

PULP (Parallel Ultra-Low-Power) is an open-source multi-core microcontroller with a configurable number of cores. It was created mainly for IoT applications in which it has to manage data streams generated by multiple sensors. PULP is much more complex than PULPino, in fact it is characterized by an autonomous I/O subsystem and is able to manage very compute-intensive programs by exploiting its multi core cluster. The actual release of PULP is called Mr.Wolf which is divided in two parts: the SoC and Cluster domains.

The SoC domain consists in a MCU called Fabric Controller in which a 2-pipeline stages RISC-V processor is implemented. The SoC talks with the external world exploiting the following peripherals: Quad SPI (400 Mbit/s), I2C, 4 I2S (4 x 3 Mbit/s), a parallel camera interface (400 Mbit/s) and UART used for parallel capture of images, sounds and vibrations, a 4 channels PWM interface, GPIOs, and a JTAG interface for debug purposes. On-chip memory is extended via a DDR HyperBus interface (800 Mbit/s). In order to reduce the load of the RISC-V processor during the I/O operations, a DMA is used to manage the communication with the peripherals [13]. In order to obtain the wanted high computing efficiency also the memory organization is an important parameter. In this case a L2 memory of 512 KB is used. It is divided in six banks: four word-level interleaved 112 KB banks really shared between all pulp components and two 32 KB banks reserved to the FC. The two private banks were inserted to avoid a strong reduction of the bandwidth caused by the high bandwidth requested by the instructions retrieving. Each memory location in the SoC can be easily modified or read by every element in the chip with the right permissions [12].

The Cluster exploits dedicated voltage and frequency. It is used by the FC to execute high computational programs. Mr Wolf cluster includes eight RISC-V cores. The cluster exchanges data with a scratchpad memory called L1. It is a SRAM organized in sixteen banks of 4KB each. As stated in [12] *"the L1 memory can serve all memory requests in parallel with single-cycle access latency, thanks to a low-latency logarithmic interconnect featuring a word-level interleaving scheme with round-robin arbitration resulting into a low average contention rate (<10% even on data intensive kernels)."* These features are essential to exploit a parallel execution of the program by using, for example, a thread approach as will be done with the turbo decoding application. The cluster cores retrieve the instructions from a shared latch-based cache memory that guarantees a higher bandwidth than the SRAM approach but also a higher area occupation. This shared instruction memory is preferable to private memories because allows to maintain the same performances avoiding instructions replications and reducing the area. This is used to compensate the area overhead of the latch-based implementation. The cache memory is made up of four parts of 4 KB each. As explained in [12] *"each array has a single write port connected to the AXI-bus, used for refills, and 8 read ports connected to the prefetch buffers of the RI5CY cores."* There is also a controller used to manage the transfer of the code between the shared instruction cache and the L2 memory. When multiple misses happen in the same location at the same time, thanks to the controller, only a single data request will be done to the L2 memory and not one for each miss.[12].

In order to work on PULP three steps are required:

- Toolchain installation.
- Development kit for PULP (PULP-SDK) installation.
- PULP platform execution and simulation

The order in which the steps are proposed is not random but it has to be followed. The simulation of the applications requires a licensed version of Mentor ModelSim/QuestaSim as explained in subsection 2.3.3.

2.3.1 Toolchain installation

The toolchain used can be downloaded at the link in [10]. There are two different build approach that can be followed: the ELF/Newlib toolchain and the Linux-ELF/glibc toolchain.

In this thesis the first approach is described because it was the one used. First, as prerequisites, some packages has to be installed. On Ubuntu the packages are accessed by using the command:
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev
Instead on Fedora/CentOS/RHEL OS the command is:
$ sudo yum install autoconf automake libmpc-devel mpfr-devel gmp-devel gawk bison flex texinfo patchutils gcc gcc-c++ zlib-devel
The toolchain has to be downloaded by using git:
$ git clone –recursive https://github.com/pulp-platform/pulp-riscv-gnu-toolchain.
In order to install Newlib toolchain for all pulp variants, it is necessary to choose an install path (could be any) and add */install_path/bin* to the Linux environmental variable *PATH*.
In the end the following commands have to be issued:
./configure –prefix=/opt/riscv –with-arch=rv32imc –with-cmodel=medlow –enable-multilib make

To avoid compatibility problems it is necessary an older version of gcc (older than five and more recent than four) [10].

2.3.2 PULP-SDK installation

PULP-SDK is the PULP software development kit, it can be installed in several ways as stated in [10], but in this dissertation the default and easiest approach will be described. The dependencies are the toolchain that should have been already installed in the previous paragraph and licensed version of Mentor ModelSim/QuestaSim (as explained in subsection 2.3.3) which installation has to be performed before the next steps. Consequently only the SDK has to be installed.

PULP-SDK building needs some prerequisites:

- If Ubuntu 16.04 is used the following packages have to be installed:
  $ sudo apt install git python3-pip gawk texinfo libgmp-dev libmpfr-dev libmpc-dev swig3.0 libjpeg-dev lsb-core doxygen python-sphinx sox graphicsmagick-libmagick-dev-compat libsdl2-dev libswitch-perl libftdi1-dev cmake scons
  $ sudo pip3 install artifactory twisted prettytable sqlalchemy pyelftools openpyxl xlsxwriter pyyaml numpy

- If Scientfic Linux 7.4/CentOS 7 is the OS used, the following packages have to be installed:
  $ sudo yum install git python34-pip python34-devel gawk texinfo gmp-devel mpfr-devel libmpc-devel swig libjpeg-turbo-devel redhat-lsb-core doxygen python-sphinx sox GraphicsMagick-devel ImageMagick-devel SDL2-devel perl-Switch libftdi-devel cmake scons
  $ sudo pip3 install artifactory twisted prettytable sqlalchemy pyelftools openpyxl xlsxwriter pyyaml numpy

In order to install the SDK, two environmental variables must be set and consequently the following commands have to be issued:

$ export PULP_RISCV_GCC_TOOLCHAIN=<path to the folder containing the bin folder of the toolchain>
$ export VSIM_PATH=<pulp root folder>/sim

The SDK is organized by using the git modules, the master branch is the one used in this dissertation, the top module can be downloaded executing the following command:
$ git clone https://github.com/pulp-platform/pulp-sdk.git
With the actual configuration the sub modules are downloaded by using the HTTPS but if the SSH has to be used the following command has to be issued in order to have the right configuration:
$ export PULP_GITHUB_SSH=1
In this thesis the ssh was used.

The last steps necessary to build correctly the SDK (they have to be issued inside the downloaded SDK folder, that is *pulp-sdk*) are the following:

- in order to select the system on which the applications have to be run, the following command has to be issued:
  $ source configs/pulp.sh
- in order to select the platform (RTL simulator, FPGA or virtual platform) on which the applications have to be executed, the following command has to be issued:
  *$ source configs/****platform****.sh*
  for example, to use the RTL simulator ***platform*** has to be substituted with ***platform-rtl***.

Anytime a new terminal window is used or the system that has to run the application changes, the source commands analyzed in these last steps have to be issued.

In the end the SDK can be installed by executing the command *make all* inside the *pulp-sdk* folder.

The installed SDK has to be also set up issuing the following command:
$ source pkg/sdk/dev/sourceme.sh
This operation has to be done every time a new teminal is opened.

2.3.3 PULP platform execution and simulation

After the execution of all previous steps, the PULP platform can be downloaded from the link [9]. Also PULP is organized in submodules by means of git, consequently in order to download the submodules the command *./update-ips.py* has to be issued. Now the set up of the simulation system has to be done. It requires the ModelSim/QuestaSim function *VOPT* that is present only in the licensed version but not in the Altera starter edition.

In order to set up the simulation system three commands have to be issued:
source setup/vsim.sh
cd sim/
make clean lib build opt

The next step is the real simulation of a sample application. On [14] some examples can be downloaded. Each example has its own folder in which there are the program files and a makefile which contains some configuration flags. The simplest is located in *pulp-rt-examples-master/hello*. To execute it the command *make clean all run* has to be issued in the *hello* folder. In this case a simulation will be run on the ModelSim console and its evolution will be shown on the terminal. Instead if the simulation has to be shown on the ModelSim Graphic User Interface, before issuing *make clean all run*, the command *make conf gui=1* has to be issued. To install the standard configuration *make conf* has to be executed. By issuing the command *make help* some useful tips for the application configuration are shown. Besides other informations can be obtained, by typing *make help_flags* (list of the available flags that can be inserted in the makefile), *make help_config* (list of what can be configured in the simulation environment), *make help_opt* (list of the possible make options).

The most important flags that can be inserted in the makefile are the following:

- *PULP_APP="name of the application"*. This flag is necessary to compile any program that has to be run on the PULP platform.
- *PULP_APP_CL_SRCS="name of the files that will be executed on the cluster"*.
- *PULP_APP_FC_SRCS="name of the files that will be executed on the Fabric Controller"*.
- *PULP_CFLAGS="name of the compiler flags used to set, for example, the grade of gcc optimizations"*.
- *PULP_LDFLAGS="name of the linker flags"*. This flag is usually used to include some special libraries like, for example, *math.h* that needs the flag *-lm*.

As explained in the introduction of this section the cluster is started and controlled by the Fabric Controller, consequently the main file has to be always compiled by the Controller itself by means of the right flag. Then, in the main file, the cluster can be started. In the end it is possible to observe that a new application can be created by inserting in a new folder the program files and the makefile with all the necessary flags as specified before.

Chapter 3

Turbo decoding

3.1 Turbo coding

A communication system is generally composed by a certain number of elements that have to guarantee a safe transmission of the input informations sequence through a communication channel. The components used are shown in figure 3.1.

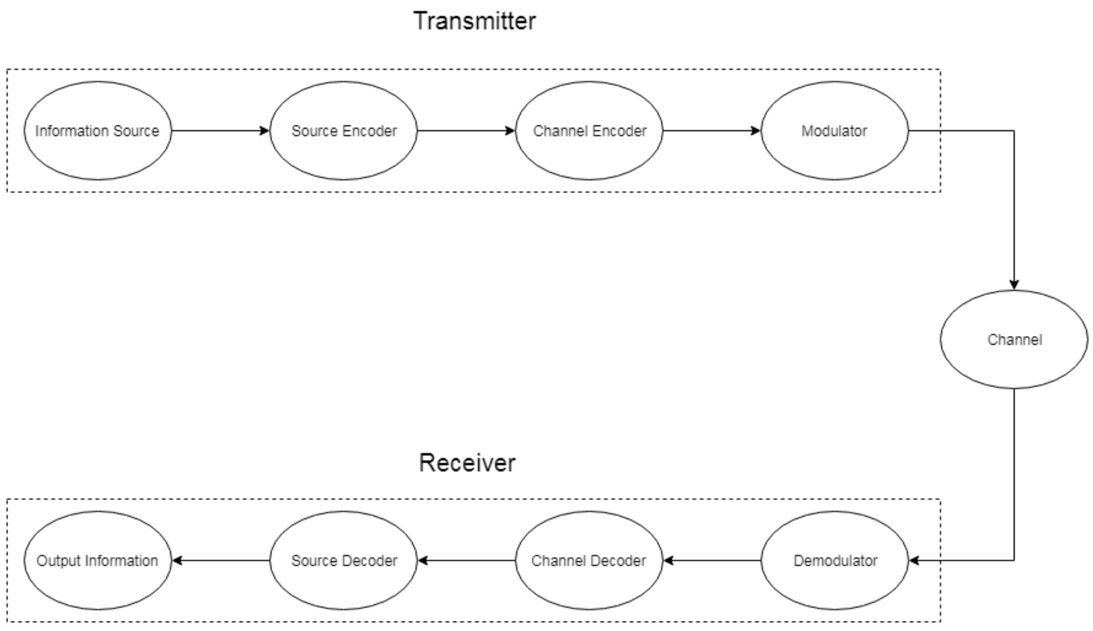

Figure 3.1. Communication System Block Diagram

They usually are:

- Information sources. they can be analog or digital.
- Source Encoder. It changes the data representation in order to achieve a more efficient transmission with less power consumption or higher performance.

- Channel Encoder. It transforms the data in a new sequence in order to obtain a transmission more robust against the noise. This is what a Turbo Encoder do.
- Modulator. It is used to traduce the data in a signal transmittable on the medium chosen for the channel.
- Channel. It is the medium used to transmit the data.
- Demodulator. It is used to traduce the transmitted signal in the data format used by the source.
- Channel Decoder. It executes an algorithm to restore the *source coded* data and eventually detects and corrects data transmission errors.
- Source Encoder. It restores the original data values.
- Output information. It provides the final data in output.

There are many algorithms that can be used to encode an information, the one used in this thesis belongs to the turbo codes group.

Turbo codes are the first error correcting codes capable of approaching the Shannon's capacity limits. Due to their strong performances they were used into many modern structures like 3G and 4G radio systems. The turbo encoding/decoding is used in the channel encoder/decoder presented just before. A general parallel concatenation convolutional turbo encoder scheme is shown in figure 3.2. It is built by two convolutional encoders and one interleaver. The two output encoder flows pass through a puncturing and multiplexing block in which some parity bits are removed. Then all encoded bits are sent to the decoder. The encoder sends only the parity bits and the encoded original sequence but not the interleaved one reducing consequently the necessary bandwidth. More informations on the turbo encoding can be found in [15].

A turbo decoding algorithm can be implemented in many ways according to the specifications provided. In this thesis the *parallel concatenated convolutional codes* (*PCCC*) was considered. The PCCC decoder scheme is represented in figure 3.3. It is characterized by two SISO decoders that take as inputs the Log Likelihood ratio (LLR) of the *parity* bits, the *systematic* sequence and the *a priori* information. The last is obtained from the interleaved/deinterleaved extrinsic LLRs provided respectively by the SISO decoder 1/2. The output can be provided by any of the two decoders. The decoding process is done iteratively, it can be started by any decoder. At the first iteration the *a priori* information of the starting decoder is set to zero. The iteration is performed sequentially so, first, the output of the starting decoder is computed and then, the extrinsic information provided by the starting element is exploited to compute the output of the next SISO decoder. This approach is exploited iteratively, in this way each decoder can use a new improved extrinsic information and give a more precise output at each new iteration. The optimal number of iterations depends on the size of the input data sequence and latency allowed. In [18] are shown some examples in which the optimal iterations number is computed for the 4G-LTE application for some input length hypothesis. The same computation are done for the UMTS application as shown in [19].

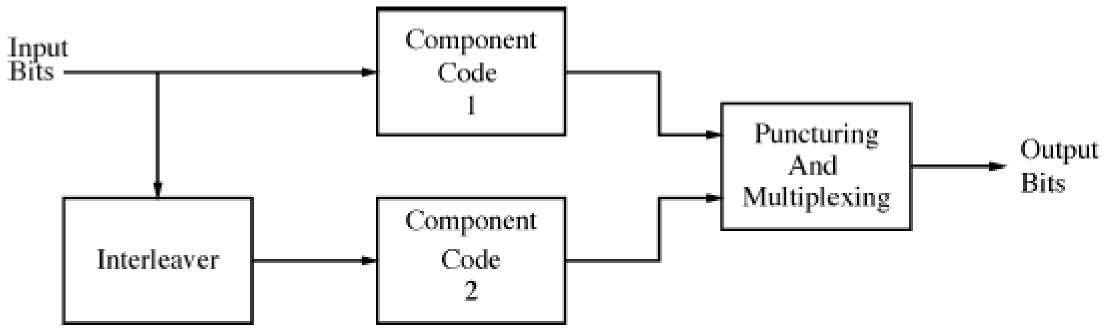

Figure 3.2. Turbo encoder [16]

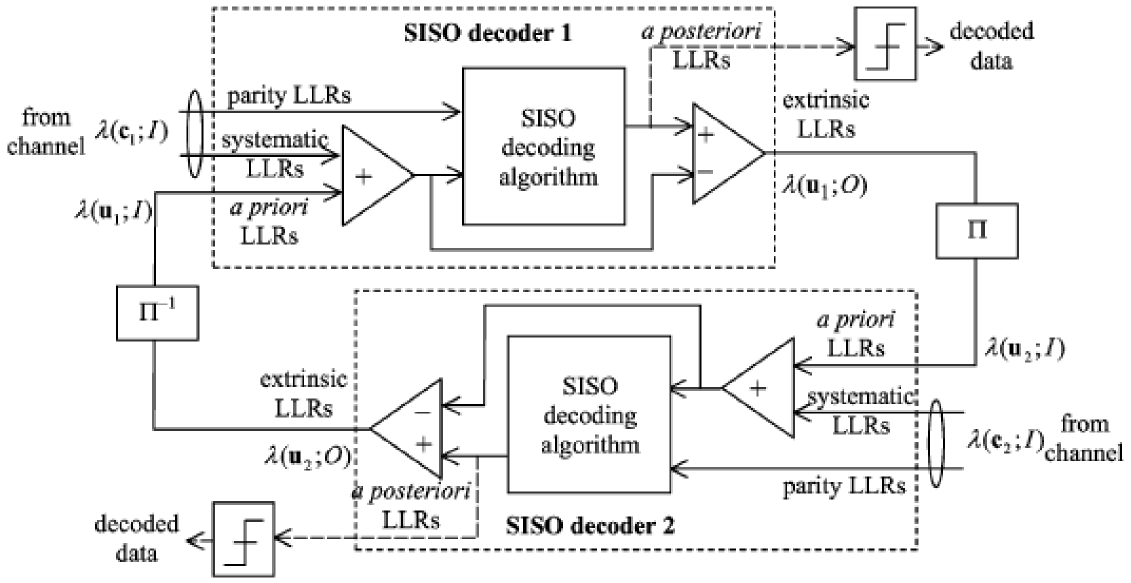

Figure 3.3. PCCC Decoder structure [15]

3.2 SISO decoder

After the encoding of the input information, the bits sequence was modulated by using, in the case treated in this thesis, the binary phase shift keying (BPSK) modulation that implies a mapping of the encoded bits by using a different set of digits. In particular, called x the modulated signal and h the encoded sequence, there are two different cases:

- if h is equal to *0* then x will be equal to *+1*
- if h is equal to *1* then x will be equal to *-1*

Besides there was the hypothesis of additive white Gaussian noise (AWGN) channel that provides an output signal given by $c{=}x{+}n$ where n is the white Gaussian noise.

The algorithm used by the SISO decoder, in this thesis, is called Maximum A Posteriori (MAP) algorithm. It is based on the Log Likelihood ratio that provides the soft information of each symbol. In particular it gives the probability, for each bit, to be *'1'* or *'0'*.

The soft output is given by a posteriori LLR that is the ratio of two conditional probabilities in which the assumptions are that the output can be *'1'* or *-1*. The formula is shown in 3.1. The term x is the possible output mapped as the modulated signal and instead c is the encoded input.

$$\lambda(x) = ln\left(\frac{P(x = +1|c)}{P(x = -1|c)}\right) \tag{3.1}$$

The decision on the output bits is taken according to the value of the output LLR:

- if it is equal to or higher than *0* the probability that the output is equal to *'0'* is higher than the probability that the output is equal to *'1'* and so the information symbol should be equal to *'0'*
- if it is lower than *0* the probability that the output is equal to *'1'* is higher than the probability that the output is equal to *'0'* and so the information symbol should be equal to *'1'*

In order to write an exploitable LLR equation it is necessary to introduce the trellis representation of the SISO algorithm.

3.2.1 SISO trellis representation

The encoding of the information sequence depends on the constraint length of the code considered. It can be defined as the number of bits used to compute a single encoded symbol. In the case treated in this thesis, the constraint length was equal to three.

The constraint length allows to compute the number of states necessary to decode a single information symbol. Called L the constraint length, the number of states are given by the relation 3.2. The constraint length can be considered equal to the number of memory elements exploited in the encoder process.

$$Number\ of\ states = 2^L \tag{3.2}$$

Consequently the number of states necessary for the 4G-LTE and UMTS applications considered was eight. If, for the entire input sequence, the state diagram of an information symbol is linked to the one related to the next element it is possible to obtain a *trellis diagram*. The encoding is given by a unique path along the *trellis diagram*. So the decoder has to recognize this path along the *trellis diagram* in order to rebuild the input sequence. In order to understand better this concept let's consider a certain instant that corresponds to a trellis stage *'i'*. In that stage there are eight different states (if the constraint length is equal to 3) in which the encoder can be. For each state there are two different states in which the encoder can go. The state belonging to the encoder path in the *'ith'* trellis stage depends on the previous choices, instead the state in the trellis stage *'i+1'* depends on the input symbol provided in the *'ith'* instant.

The number of trellis stages depends on the input sequence length, it changes according to the application considered. The ones taken into account in this thesis were the 4G-LTE for which the frame size (input sequence length) called K has been obtained using the equations described in 3.3 and the UMTS for which the frame size could assume each value in the range 40-5114.

$$K = \begin{cases} 40 + 8 \cdot (m - 0) & \text{if } m = 0 \ to \ 58 \\ 512 + 16 \cdot (m - 59) & \text{if } m = 59 \ to \ 90 \\ 1024 + 32 \cdot (m - 91) & \text{if } m = 91 \ to \ 122 \\ 2048 + 64 \cdot (m - 123) & \text{if } m = 123 \ to \ 187 \end{cases} \tag{3.3}$$

When the data sent through the channel are organized in blocks the decoder has to be able to recognize the end of a block. There are many ways to solve this problem, for example in the applications seen before the tail bits have been used. This approach implies the addition of some bits, and consequently some trellis stages, at the end of the encoded block. In particular, in the LTE and UMTS applications, twelve bits have been added. An other approach is the tail biting technique that is described in [15].

3.2.2 LLR computation

Considering the trellis representation seen in subsection 3.2.1, the probability of x_i (x at the ith trellis stage) can be written as shown in 3.4. Using the equation 3.4 the LLR described in 3.1 can be rewritten as shown in 3.5.

$$Pr(x_i) = \sum_{(S_i, S_{i+1})} Pr(x_i; S_i, S_{i+1}) = \sum_{(x_i; S_i, S_{i+1})} Pr(S_i, S_{i+1}) \tag{3.4}$$

$$\lambda(x_i) = ln \frac{\sum_{(x_i=1; S_i, S_{i+1})} Pr(S_i, S_i + 1; c)}{\sum_{(x_i=-1; S_i, S_{i+1})} Pr(S_i, S_i + 1; c)} \tag{3.5}$$

By making some computations, done in [18], the LLR can be written as shown in 3.6

$$\begin{aligned} \lambda(x_i) = ln\Bigg(\sum_{(x_i=0; S_i, S_{i+1})} e^{\alpha(S_i)+\gamma(S_i,S_{i+1})+\beta(S_{i+1})} \Bigg) - \\ -ln\Bigg(\sum_{(x_i=-1; S_i, S_{i+1})} e^{\alpha(S_i)+\gamma(S_i,S_{i+1})+\beta(S_{i+1})} \Bigg) \end{aligned} \tag{3.6}$$

Where α, β and γ are written respectively in 3.7, 3.8 and 3.9. In the equation 3.9 j indicates the state of the trellis i in which γ will be computed, λ_a is the LLR of the *a priori* information and L_c is the channel reliability that is equal to $\frac{2}{\sigma^2}$ in which σ^2 is the variance and σ can be computed by looking at the bit signal-to-noise ratio (SNR), usually called E_b/N_0.

$$\alpha(S_i) = ln\Bigg(\sum_{S_{i-1}} e^{\alpha(S_{i-1})+\gamma(S_{i-1},S_i)} \Bigg) \tag{3.7}$$

$$\beta(S_i) = ln\Bigg(\sum_{S_{i+1}} e^{\beta(S_{i+1})+\gamma(S_i,S_{i+1})}\Bigg) \tag{3.8}$$

$$\gamma(S_i, Si+1) = \frac{1}{2}\cdot x_i^{(j)} \cdot \lambda_a(x_i^{(j)}) + \frac{1}{2}\cdot L_c \cdot \sum_{j=0}^{1}[c_i^{(j)} \cdot x_i^{(j)}] \tag{3.9}$$

α is usually called forward metric because it depends on the α and γ computed at the previous trellis stage, so it is necessary a forward recursive approach to compute α at each trellis stage. Instead β is usually called backward metric because depends on β and γ computed at the next trellis stage and consequently it is necessary a backward recursive approach to compute β at each trellis stage.

3.2.3 Log-MAP algorithm

The algorithm seen in subsection 3.2.2 can be simplified by using an alternative way to make the exponentiation and logarithm operations. In fact these two operations are the reason why the MAP algorithm is so complex. They can be substituted by a new function called *max** as shown in 3.10. The definition of the *max** function is given by the formula 3.11. This approach can be used also with three terms as shown in 3.12. The logarithm term in the equation 3.11 can be implemented by using a lookup Table as done in the two applications treated in this thesis. The dimension of the lookup table is set according to the required accuracy. The approach just described is called Log-MAP algorithm.

$$ln(e^{(x)} + e^{(y)}) = max^*(x,y) \tag{3.10}$$

$$max^*(x,y) = max(x,y) + ln(1 + e^{-|x-y|}) \tag{3.11}$$

$$ln(e^{(x)} + e^{(y)} + e^{(z)}) = max^*(x,y,z) = max^*(max^*(x,y),z) \tag{3.12}$$

The lookup table can be discarded leading to a simpler algorithm called Max-Log-MAP, that can obviously bring to a suboptimal performances in terms of BER. In the 4G-LTE and UMTS applications the Log-Map algorithm has been used.

3.3 Windowing

As seen in 3.6 the LLR depends on the backward and forward metrics. The approach used to compute these two parameters are much different. The computation of alphas is done by using a forward recursion and consequently at each input sample the corresponding alphas can be computed, instead for betas the backward recursion has to be used and so the betas computation can start only when the last sample is arrived. This implies that both alphas and input samples have to be stored in memory until the computation of the betas becomes possible. In this way also the latency between the first input sample and the first decoded output becomes very high. The latency and memory overheads increase proportionally to the frame size. So for long input sequences the situation could become unsustainable. To avoid these problems the sliding window technique can be introduced. In this approach the frame is divided into small slices called windows that are executed in a natural order. The alpha metrics related to the last trellis of a certain window are used in order to initialize the first trellis stage of the next window. For the initialization of the backward metrics of each window is required a dedicated functional unit. The overhead is reduced because in this case the alphas and inputs that have to be stored are the ones related to a single window and not to an entire codeword.

3.4 Turbo Decoder Parallelization

After the windowing a further division can be applied. In particular it is possible to exploit a parallel architecture to obtain the results of more than one window at the same time. The windows should be equally distributed between the processing elements (PEs). The number of windows assigned to a single PE is obtained by dividing the total number of windows by the total number of PEs. This approach can lead to two problems:

- Data Dependency problem. Usually in the turbo decoding algorithm the alpha metrics are used to initialize the first trellis stage of the next window, while for the beta metrics a different method has to be applied. If a parallelization is applied also the forward metrics should require a different initialization approach. A way to solve the problem of the initialization of both metrics of the first trellis stage of each window is to use the metrics calculated in the previous iteration. In particular the alphas and betas computed respectively in the last and the first trellis stages of a certain window will be used to initialize respectively the forward recursion of the next window and the backward recursion of the previous window that will be performed at the next iteration. This approach implies that the forward an backward metrics computed respectively in the last and first trellis stages of each window have to be stored. Consequently if the window size is very small the number of windows become very high and so the alphas and betas stored at each iteration increase proportionally.

- Memory organization. Each data received in input by the decoder is stored in a memory, and it can be accessed in a natural order or interleaved. If N PEs are used to compute the decoded results, the input data have to be accessed by different elements. If the data should be addressed only in one order (for example the natural) the memory can be divided in N banks, one for each processing element. In this way each PE has its own private memory with all its data. But since there are two different orders in which the data can be addressed, a PE could require data from a bank different from the one reserved to it. Consequently each bank should be connected to each SISO and vice versa. Besides a memory conflict can happen if more than one SISO tries to access to the same memory bank. There are many ways to solve this problem [15]. In the architecture used in this thesis there is a memory that allows every kind of parallel access by the processing elements that it serves, this is highlighted in section 2.3.

Chapter 4

Turbo decoding implementation

As stated in chapter 1 the goal of this thesis is to implement on a general purpose platform a turbo decoding algorithm that requires to execute in parallel high intensive computational tasks. For this purpose the PULP-platform described in chapter 2 has been used. In particular the turbo decoding algorithm was implemented in C and, by using the toolchain, it was mapped on the PULP-platform. The first approach was done with PULPino, a RISCV single core microcontroller on which the turbo decoding algorithm has been executed. This was done in order to obtain a greater familiarity with the PULP-platform. In order to pursue the goal explained above the microcontroller PULP was adopted. It is characterized by a cluster of processors useful for the parallelization of the turbo decoding algorithm.

The turbo coding approach has been applied on two applications: LTE and UMTS communication protocols. The turbo decoding program was composed by two parts: turbo coding and turbo decoding. Since in this thesis only the decoding was treated the first part of the algorithm was eliminated. This implies that the encoded data, the vector obtained by exploiting the permutation law, and all the parameters used in the encoding process (frame size, window size, parallelism degree etc) were provided as inputs. The other modifications done to adapt the algorithm to the platforms used are described in sections 4.1 and 4.2.

4.1 PULPino approach and results

As stated above in order to approach gradually PULP-platform, first the simplest system was used. It is called PULPino and its features are described in section 2.2. To run the algorithm on this system some modifications to the code and to PULPino were necessary:

- In the code almost all the dynamic allocations were substituted with static ones in order to reduce the overhead and detect the allocation problems during the compilation phase.

- In order to run the Turbo decoding algorithm on PULPino a new folder called *Turbo* was created in the applications directory (*sw/apps/*). Inside the Turbo folder the main file of the algorithm (called *main2.c*) and a *CMakeLists.txt* file were inserted. In the *CMakeLists.txt* file the statement *add_application(Turbo main2.c)* was written. It indicates that in order to execute the program in the modelsim GUI or console the command *make Turbo.vsim* or *make Turbo.vsimc* has to be typed on terminal respectively. Also the CMakeLists.txt in *sw/apps* folder was modified by adding the line *add_subdirectory(Turbo)*. All the other files included in the application have been added as library, so a new folder called *Turbo_lib* has been defined in the path *sw/libs*. Inside the *Turbo_lib* directory two new folders were created *inc* and *src* in which the header and source files were inserted respectively. In the *Turbo_lib* folder a new CMakeLists.txt file has been created. It is filled with the following statements:

```
set(SOURCES
src/decoder.c
src/myservs.c
)
set(HEADERS
inc/decoder.h
inc/myservs.h
inc/myretcodes.h
)
include_directories(inc/)
add_library(Turbo STATIC ${SOURCES} ${HEADERS})
```

In the end the CMakeLists.txt files present in the path *sw* and *sw/apps* were modified by substituting the *new* statement with *Turbo* in the bold sentences reported in *Appendices* A.2 and A.1 [5].

- The data quantity in the worst case in which a frame size of 6144 (for the LTE application) is used exceeds the data memory already integrated in PULPino system. In order to change the memory size three steps are required:
  - Change the file *core_region.sv* in *rtl* folder. In the row 22 the value of the *parameter DATA_RAM_SIZE* has to be modified from *32768* to the desired value (in this case 524288).
  - Change the linker script *link.common.ld* in */sw/ref* path. In row 7 the *parameter LENGTH* has to be modified in accordance with the modifications done in the previous step. Since the size of the memory in this case was set to 524288 and the origin address of the data memory was *0x00100000* the length has been set to *0x7E000*. This is due to the fact that also the stack size is included in the memory size set in the previous step (the length of the stack is *0x2000*), in fact the origin of the stack indicated by the parameter *ORIGIN* has to be also modified. The stack start address should be set to the data memory origin address plus its size. For example, in this case, the stack origin was set to the value *0x0017E000*.
  - Change the file *s19toslm.py* (which is used to load the data in memory during the modelsim simulation) placed in */sw/utils* path. In row 117 the value assigned to

the *parameter tcdm_bank_size* has to be changed in accordance to the previous steps. In this value only the data memory size is taken into account without considering the stack size. Consequently for the application under analysis the value was set to 129024.

In order to measure the performance of the algorithm *performance counters* integrated in the system were used. They are able to count the total number of cycles, instructions, load hazards, load instructions etc necessary to terminate the program. The libraries provided with PULPino allows to access easily to these counters. In particular the following functions were used:

- *static inline void cpu_perf_conf_events(unsigned int eventMask).* Configure the events that have to be counted in the program and start the count.
- *static inline void perf_reset(void).* All performance counters are set to zero. The counts are not stopped.
- *static inline void perf_stop(void).* All performance counters are stopped.
- *static inline unsigned int cpu_perf_get(unsigned int counterID).* It provides the value stored in the counter associated to the specified *counterID.*

The event can be set in two ways:

- Using a different macro according to the wanted event.
- Using the function *SPR_PCER_EVENT_MASK* that allows to make an OR of the events for which the count is needed. In particular this function takes as argument an integer number (here called *event ID*) that identifies the event considered.

Some of the possible events that can be counted are the following:

- Clock cycles by using the macro *SPR_PCER_CYCLES* or the *event ID 0.*
- Instructions by using the macro *SPR_PCER_INSTR* or the *event ID 1.*
- Stall caused by load instructions by using the macro *SPR_PCER_LD_STALL* or the *event ID 2.*
- Stall caused by jump instructions by using the macro *SPR_PCER_JMP_STALL* or the *event ID 3.*
- Instruction miss in the cache by using the macro *SPR_PCER_IMISS* or the *event ID 4.*
- Jumps by using the macro *SPR_PCER_JUMP* or the *event ID 9.*
- Branch by using the macro *SPR_PCER_BRANCH* or the *event ID 10.*

These and other events can be found [5] in the file *convolution.h* at the path
/sw/apps/sequential_tests/convolution .

For the applications analysed only the number of clock cycles was needed. This parameter was used to evaluate the performance of the algorithm on the system used.

4.2 PULP approach and results

As written in the introduction of this chapter the goal of this thesis is to exploit the PULP parallel architecture to compute faster the high intensive computational tasks of the Turbo decoding algorithm. The steps necessary to achieve the wanted result are the following:

- Algorithm mapping on PULP.
- Algorithm Parallelization.
- Bits representation.
- Dynamic allocation.
- Performance Evaluation.
- Frames storing

4.2.1 Algorithm mapping

In order to execute the algorithm on PULP using the cluster a new folder called *Turbo* was created inside the directory *pulp-rt-examples*. In the new folder a *Makefile* necessary to run the program on PULP was created. It includes in general the statements indicating which platform (cluster or Fabric Controller) has to compile the files necessary to implement the algorithm, the flags used to include some special libraries and the flags used to indicate the optimization grade of gcc. In this case the *Makefile* was organized as following:

```
PULP_APP = test
PULP_APP_CL_SRCS = myservs.c execution.c decoder.c
PULP_APP_FC_SRCS = test.c
PULP_LDFLAGS = -lm
PULP_CFLAGS = -O3 -g

include $(PULP_SDK_HOME)/install/rules/pulp_rt.mk
```

Also the files characterizing the program had to be placed in the folder *Turbo*. The variables were allocated statically in order to reduce the overhead and detect the allocation problems during the compilation phase. The static allocation can be done by declaring all the parameters as global variables with the statement *RT_L2_DATA* (in order to allocate the data in the memory L2) or *RT_L1_DATA* (in order to allocate the data in the memory L1 of the cluster). When a declaration is done statically, at the beginning of the execution of the program the data are precharged in the L2 memory by exploiting the JTAG independently on the memory on which the data have to be stored. In particular the data destined to the cluster L1 are stored in one of the private banks of the fabric controller, then when the cluster is powered up they are transferred to the cluster L1 memory. The L1 static allocation can be done in the files assigned to the cluster in the *Makefile*. If instead an L1 (the data memory inside the cluster) allocation has to be done in a function compiled by the Fabric Controller, a dynamic or static allocation, after switching on the cluster, has to be used. For simplicity in the first implementations of the program all the data have been allocated in L2 memory.

In order to use the cluster it was necessary to modify the program provided. In particular a new file, called test.c, was added. This new file was executed by the Fabric Controller which is able to turn on the cluster using the functions provided by the PULP development kit:

- *void rt_cluster_mount(int mount, int cid, int flags, rt_event_t *event).* This function is used to turn on or turn off the cluster that is turned off by default. Let's analyse the parameters specified in the function:
  - *int mount.* If set to 0 the cluster will be turned off, if set to 1 the cluster will be powered up.
  - *int cid.* It indicates the cluster that has to be turned on. In the application under analysis just one cluster was used and consequently this parameter was set to zero (cluster id of the first cluster).
  - *int flags.* This parameter is not actually used and consequently will be always set to 0.
- *int rt_cluster_call(rt_cluster_call_t *call, int cid, void (*entry)(void *arg), void *arg, void *stacks, int master_stack_size, int slave_stack_size, int nb_pe, rt_event_t *event).* This function assigns to the core 0 (master core) the execution of a certain function called entry. During its elaboration the other cores are available for a parallel computation. After calling the cluster the Fabric Controller waits until the execution of the entry ends and the cluster returns. The parameters that have to be provided to this function are the following:
  - *rt_cluster_call_t *call.* This parameter is used only when two calls have to be pipelined, otherwise can be set to *NULL*. In the application under analysis was set to *NULL*.
  - *int cid.* It specifies the cluster for which the call is done. In the turbo decoding algorithm only one cluster was exploited and its cluster id was equal to 0. Consequently the *cid* was set to 0.
  - *void (*entry)(void *arg).* The function assigned to the master core. In the specified applications this function was called execution.
  - *void *arg.* The *entry* function can have an argument that is a void pointer. If the data pointed by the argument have to be placed in the cluster L1 memory, they should be declared after that the cluster is turned on exploiting a dynamic or static allocation. This is due to the fact that the cluster L1 memory can be accessed only when the cluster is turned on.
  - *void *stacks.* It is the total stack assigned to all available cores. If the default stack size is desired, the parameter has to be assigned to *NULL*. In the last approach the stack will be freed at the next *call*. The *NULL* assignment was used in the application under analysis.

  - *int master_stack_size* . This parameter specifies the stack provided to the master core. If it is set to zero the default stack size is assigned to the core 0. The last option was used to implement the turbo decoding algorithm.
  - *int slave_stack_size.* It is used to assign the stack to each slave core. They will have the same stack size. Again if the parameter is set to *NULL* a default stack size is provided. This approach was followed in the implemented application.
  - *int nb_pe.* It indicates the number of cores available for the parallel execution of the entry function. When this parameter is set to 0 the maximum number of cores are involved. The platform used to run the decoding algorithm could use at maximum eight cores.

Both functions together with other utilities are explained in [20].

To sum up, in order to implement the turbo decoding algorithm, the two functions described above have been used as specified in 4.1.

Listing 4.1. Cluster power on

```
rt_cluster_mount(1, 0, 0, NULL);}
rt_cluster_call(NULL, 0, execution, NULL, NULL, 0, 0, P, NULL);}
rt_cluster_mount(0, 0, 0, NULL);}
```

Where P was the number of core used that was equal to the parallelism degree of the algorithm. Adding other features requires some modifications to the code in 4.1 as explained in subsections 4.2.4 and 4.2.6. It is also possible to not use the cluster but only the FC. Consequently the Fabric Controller have to compile all the files and the functions for the cluster management don't have to be used. Clearly in this case the file *test.c* is not useful.

4.2.2 Algorithm Parallelization

In this subsection the approach used to implement a parallel version of the turbo decoding algorithm will be analysed. Even in this case the development kit utilities have been used and unfortunately the classic thread functions provided by the library *pthread.h* could not be used. Mainly two functions are necessary to obtain the parallelization :

- *static inline void rt_team_fork(int nb_cores, void (*entry)(void *), void *arg).* It wakes up the wanted number of cores (called team) which will execute the same function. If not specified the number of cores is equal to the one provided by the function *rt_cluster_call* explained in subsection 4.2.1. Instead if the number of cores is set, also the default one is substituted with the one specified in the *rt_team_fork* function. Let's analyse the parameters highlighted in the declaration of the function:
  - *int nb_cores.* It indicates the number of cores that will execute the function specified by the parameter *void (*entry)(void *).* When the number of cores is set to zero the default one is used (the one specified in the function *rt_cluster_call* or in the previous *rt_team_fork*).
  - *void (*entry)(void *).* This parameter represents the function that will be executed by the wanted number of cores.

  - *void *arg.* It is the argument of the function specified in the previous parameter.

- *static inline void rt_team_barrier().* This function allows to stop each core until every member of the team has reached the statement *rt_team_barrier().*

Now it is necessary to analyse how these two functions have been used to implement the parallel computation of a decoded frame in the application under analysis. As seen in section 3.4, the windowing can be exploited to execute in parallel the turbo decoding algorithm. In particular each core should be able to compute the results associated to a different window at the same time. The windows has to be equally distributed between the cores as shown in figure 4.1. The specified number of cores has been provided as input parameter and it is called *parallelism degree*. Clearly also the window size and the frame size had to be provided as inputs. The number of windows per core has been found by using the equation 4.1.

$$\text{Number of window per processing element} = \frac{\text{Frame size}}{\text{Window size} \cdot \text{Parallelism degree}} \tag{4.1}$$

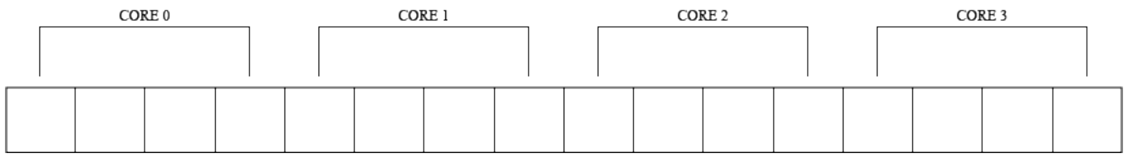

Figure 4.1. Windowing

The parallel computation in the C program used to implement the decoding algorithm was obtained using the utility *rt_team_fork* analysed before. This function has been placed in a specific section of the program. In particular after turning on the cluster, the *fork* function has been exploited to assign the main of the decoding algorithm (called *parallelization1*) to the cluster. The parameters provided to the fork function were the following:

rt_team_fork(P,parallelization1,NULL);

Where *P* was the number of cores used for the parallel computation. The shared variables included the arrays used to store the alpha and beta metrics, the extrinsic informations and the integer decoded results. Each core had to write only in the part of the array associated to the computed window. In order to assign the right window to each core the decoding kernel was organized in two loops as can be seen in 4.2.

Listing 4.2. Decoding Kernel Loops

```
rt_team_barrier();
for (siso_idx=0; siso_idx<P; siso_idx++)
{
        if(rt_core_id()==siso_idx)
        {
```

```
/// windows loop
for (wind_idx1=0; wind_idx1<NWP; wind_idx1++)
{
```

The first loop was related to the parallelism degree, consequently each iteration of this loop had to be assigned to a different core. In order to do this the function *rt_core_id()* was exploited. It provides the core id of the PE that is executing the code. In fact each core has a unique core id that is equal to a value between zero and the number of processing elements minus one. So since *siso_idx* was a variable that counts from zero to the number of cores minus one, by imposing a condition for which each core could execute the iteration in which its *core_id* was equal to the variable *siso_idx*, each processing element should be able to compute the decoded results for a number of windows equal to *NWP*. In this way the number of windows has been divided equally between the cores.

Another problem that had to be managed regarded the parallel flow of the code. Since each core is totally independent, the time spent for the execution could be different according to the processing element considered. Consequently it was necessary to make sure that the parallel section ended only when each core has finished its execution. The consequences of the highlighted problem are shown in the flow in figure 4.2. In particular in the first step the master core executes the *entry section* in which all cores are dispatched to the function that will be computed in parallel. In the second step the processing elements (in the flow taken as example are just two) execute the parallel section and in the third they should compute the exit section (it is the end of the parallel section). If unfortunately the master core finishes the execution of the exit section before the core 1, it will start automatically the reminder section. If in the last section the master core needs the results that the core 1 has to provide during the parallel section, its computations will be wrong. Besides if the core zero is enough fast it could finish the execution of the program while the core 1 is still executing the parallel or exit section. Consequently the core 1 will not able to finish the execution of the portion of program assigned to it.

To prevent the problems related to the parallel flow of the code it is necessary to use the function *rt_team_barrier()* explained before. In this way the master core can start the reminder section only when each core has finished its execution. The correct flow is shown in figure 4.3. This approach has been used for the parallel implementation of the turbo algorithm. In particular in the exit section a *rt_team_barrier()* was inserted, in this way the reminder was executed just when each core has finished its computations inside the parallel section. The exit section in this application was the one just after the conclusion of the loops which beginning is shown in the code fragment 4.2. The same approach was also followed at the beginning of the program in which there were some initialization statements (in which, for example,the metrics of the first trellis step have been set), that have been used to initialize some variables shared between the cores. Due to this fact the initialization has been done only by one core (core 0) in order to avoid any synchronization problem. A *rt_team_barrier* was used in order to block the other cores until the core zero has ended the initialization steps as shown in 4.2

4.2.3 Bits Representation

Until now all the data used in the algorithm have been stored in the memory L2 of PULP by exploiting the statement *RT_L2_DATA* just before all the global variables declared in

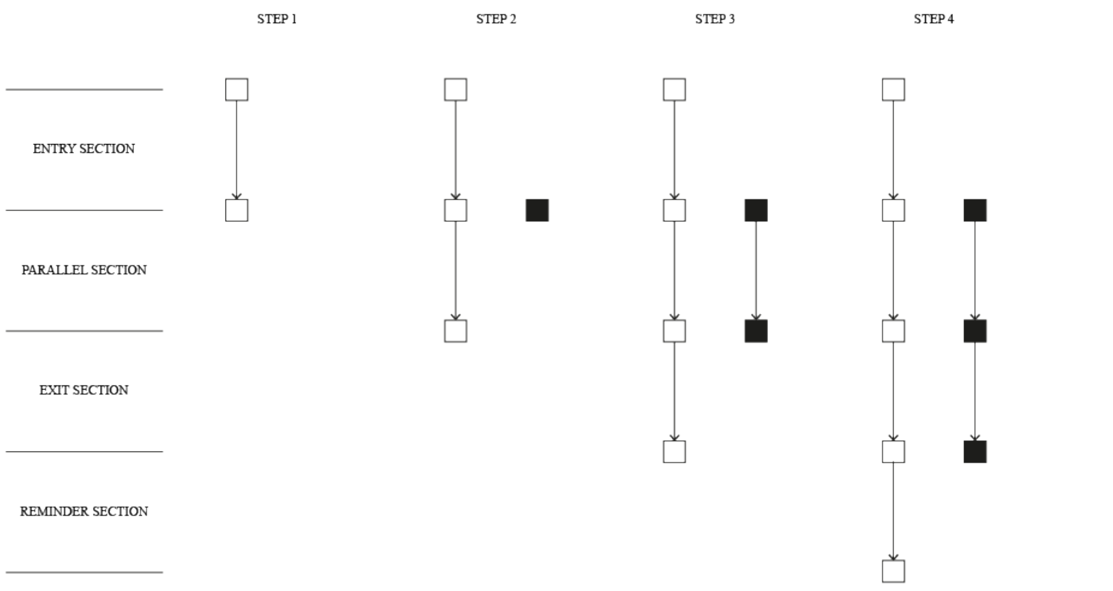

Figure 4.2. Wrong Parallelization

the code. As seen in section 2.3 the memory L2 is placed in the SOC domain outside the cluster and consequently the communications necessary to load and store the data could be a bottleneck for the performance of an algorithm. A solution for this problem could be: allocating all the data in the memory L1 placed inside the cluster. Unfortunately the quantity of data necessary for the execution of the turbo decoding algorithm was much higher than the size of the cluster L1. Consequently in order to reduce the quantity of data can be useful studying the parallelism of the algorithm by computing the bits necessary to represent each data used by the decoder. Until now all the variables were declared as *int* but in almost every case the parallelism provided by the *int* data type was not necessary.

The parameters are usually set during the encoding phase and provided to the decoding structure. Among this parameters there are also the ones that characterized the parallelism of the data. In particular the state metrics were represented by using twelve bits, instead for the intrinsic information five bits were sufficient, in the end since the numeric range of the extrinsic information was [-128-127], eight bits should be sufficient for its binary representation. It is also possible to derive the parallelism of the variable in which the

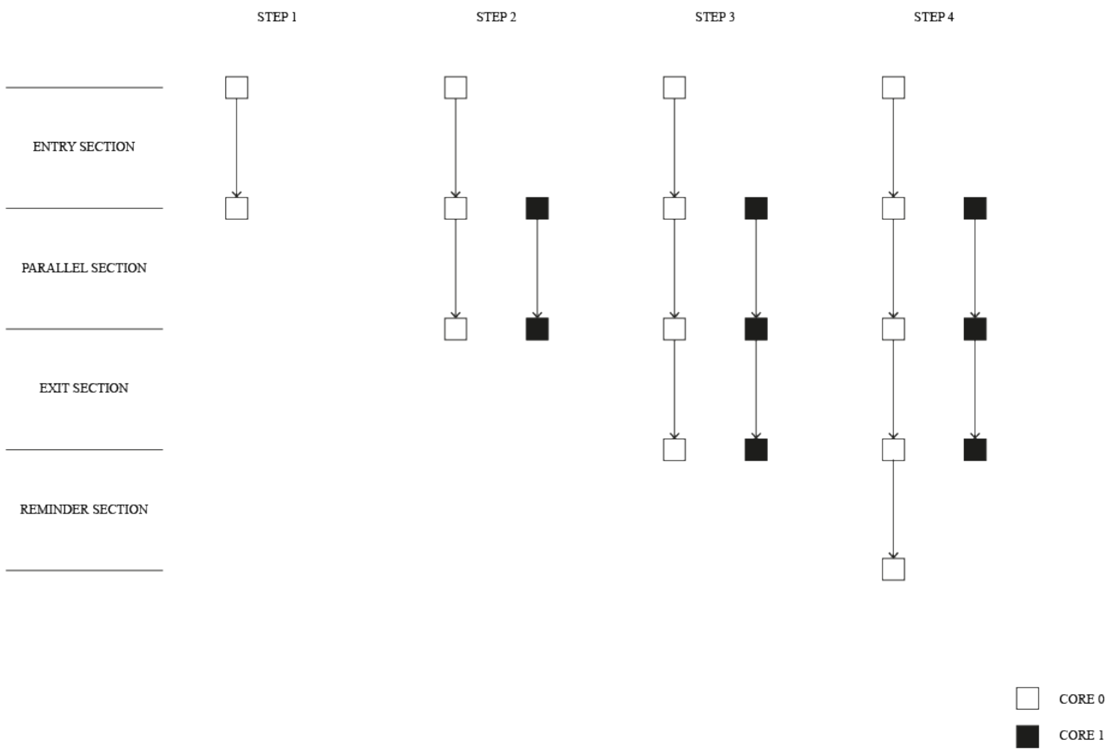

Figure 4.3. Right Parallelization

SISO decision was stored. The output of the SISO decoder was equal to the sum of the extrinsic and the a priori informations that could be represented by using eight bits. Consequently it needed of at least nine bits for a correct representation. In the end also the parallelism of the permutation vector has been reduced, in particular it had to store only values in the range [0-6144] and consequently it has been represented by using thirteen bits. The same reduction of parallelism applied to the decoding structure, inputs and outputs could be done for all the variables used in the program obtaining less significant results. In fact most of the space was occupied by the informations used by the decoder and which parallelism was just studied. Consequently a better sizing of the local variables was done but it won't be analysed in this dissertation. Since the algorithm was written in C it was not possible to select the precise number of bits necessary to represent a variable but only some fixed data types could be used. Consequently the informations used by the decoder were defined in the following way:

- since the state metrics required twelve bits, they could be defined with the sixteen bits *short* data type. Same approach for the decoded outputs and the permutation vector that required respectively nine and thirteen bits.

- As seen above the extrinsic and intrinsic informations required only eight bits and consequently a *signed char* data type has been used.

4.2.4 Dynamic allocation

As said before the performance can be improved by moving the data from the L2 memory to cluster L1 one. Unfortunately this could represent a problem since the cluster memory L1 is accessible only when the cluster is mounted. Consequently, as explained in subsection 4.2.1, the data at the beginning (until the cluster is turned on) are stored in one of the private bank of the memory (the one called private bank 0) that is characterized by a size lower than the cluster L1. Consequently if the quantity of data that the program has to store in the cluster L1 memory is higher than the size of the private bank 0, it will be saturated during the preloading done by using the jtag. This happened with the data required by the turbo algorithm and consequently the static allocation in the cluster L1 memory could not be used. To solve this problem a dynamic allocation has been used.

In order to allocate dynamically a data in the cluster memory two utilities have to be used:

- *void *rt_alloc(rt_alloc_e flags, int size)* is a function used to allocate a certain memory region which usage indicates in what memory the region will be allocated. Let's analyse the parameters used in this function:
  - *rt_alloc_e flags*. This flag indicates the specific usage of the allocated memory section. According to this flag the section can be allocated in different memories or memory areas. The possible flags are:
    - *RT_ALLOC_FC_CODE*. It indicates that the memory allocated will be used to store some code that has to be executed by the fabric controller. Consequently the allocated section will belong to the L1 memory of the fabric controller.
    - *RT_ALLOC_FC_DATA*. When this flag is provided to the *rt_alloc* function the section allocated will be used to store fabric controller data and consequently the allocation will be done in the L2 memory.
    - *RT_ALLOC_FC_RET_DATA* This flag treats the fabric controller retentive data. In this case the allocated section will be placed in the L2 memory outside the cluster.
    - *RT_ALLOC_CL_CODE*. It indicates that the memory allocated will be used to store some code for the cluster. In this case the allocated section will be placed in the L2 memory.
    - *RT_ALLOC_CL_DATA* It indicates that the memory allocated will be used to store data for the cluster. In this case the allocated section will be placed in the L1 memory of the cluster.
    - *RT_ALLOC_L2_CL_DATA*. This flag treats again the data used by the cluster. But in this case the allocated section will be placed in the L2 memory outside the cluster.

        * *RT_ALLOC_PERIPH.* It indicates that the memory allocated will be exploited to store the data that will be used by the peripherals. Again in this case the allocated section will be placed in the L2 memory.

    - *int size.* This parameter indicates the size in bytes of the section that will be allocated.

- *void rt_free(rt_alloc_e flags, void *chunk, int size)* is a function used to free a memory region allocated previously. The parameters received by this function have to be the same provided to the related allocation function:

    - *rt_alloc_e flags.* This flag indicates the specific usage of the memory section that has to be freed. According to this flag the section that has to be freed could be located in different memories or memory areas. The possible flags are the same reported for the *rt_alloc* function.
    - *void *chunck.* It is simply the pointer to the start address of the memory section that has to be freed. It is returned by the *rt_alloc* function used to allocate the region that has to be freed.
    - *int size.* This parameter indicates the size in bytes of the section that will be freed.

The allocation that exploits the two functions exposed before can be done only by the fabric controller. Consequently if some data have to be allocated in the cluster L1 memory the *rt_alloc* function has to be used on the fabric controller side after the issue of the command *rt_cluster_mount* (used to turn on the cluster). Then these data can be provided to the cluster exploiting the function *rt_cluster_call.* In fact, as seen in subsection 4.2.1, the *rt_cluster_call* calls an entry function that will be executed by the core 0 of the cluster. This entry routine can receive as argument a *void ** parameter that can be exploited to provide to the cluster the data allocated on the FC side.

All this things have been exploited to implement a turbo decoding algorithm able to retrieve data mainly from the L1 memory. In particular a structure with all the parameters used by the decoder was defined (4.3).

Listing 4.3. Parameters structure

```
struct allocation
{
        short K;
        signed char P;
        signed char W;
        signed char min_iter;
        signed char max_iter;
        signed char max_frame;
        signed char NBF_INT;
        signed char EXT_MAX_VAL;
        signed char EXT_MIN_VAL;
        signed char NB_SM;
```

```
            signed char f_num;
            signed char NS;
            short NWP;
            short N;
            signed char flag;
            signed char *llr1_out;
            signed char *llr2_out;
            short *dec1;
            signed char *llr_in[2];
            siso_mem_t *siso_memory[2];
            short *beta_init[2];
};
```

Then this structure was allocated in L1 memory by exploiting the *rt_alloc* utility. Same approach for all the pointers declared inside the structure. Obviously all this allocations were done after the cluster has been mounted as indicated in 4.4.

Listing 4.4. Cluster mounting and structure allocation

```
rt_cluster_mount(1, 0, 0, NULL);
struct allocation *fc_side;
fc_side = (struct allocation *) rt_alloc(RT_ALLOC_CL_DATA,
sizeof(struct allocation));
```

Then the allocated structure was provided to the cluster by means of the function *rt_cluster_call* with the follow implementation:
*rt_cluster_call(NULL, 0, execution,(void *) fc_side, NULL, 0, 0, fc_side->P, NULL).*New features could require a variation of the *rt_cluster_call* implementation as will be done in subsection 4.2.6.

On the cluster side instead a pointer to the structure allocated in 4.4 was created. This pointer has been declared as a global variable and consequently it could be used by all the functions declared inside the same file. In particular the pointed structure could be accessed by the function called using *rt_cluster_call* and by the function executed in parallel by all cluster cores because both were declared in the same file. In this context it could be useful recalling that the function used to execute something in parallel is *rt_team_fork* in which the number of cores, the function that will be executed and the argument of the specified function have to be indicated.

4.2.5 Performance evaluation

The performance of the algorithm has been evaluated exploiting the so called performance counters. They are able to provide accurate measurements of many events useful for the evaluation of the performance of every algorithm executed on the PULP microcontroller. The number of counters that can be accessed at the same time depends on the architecture chosen and on the platform on which the architecture is mapped. The default number of counters is twelve, it is defined in the file *ips/riscv/rtl/riscv_cs_register.sv.* The number of counters presents in the architecture limits the number of events that can be counted. The utilities provided by the toolchain to manage these counters are:

- *void rt_perf_init(rt_perf_t *perf)* is used to initialize to zero all the performance counters present in the **perf* structure. It is important to highlight that this function is not able to reset the hardware counters, but this can be done only with the reset function specified below. *rt_perf_t *perf* is a pointer used to access to the structure that contains the events configuration of the counters and the results of the counts. This structure can be accessed by more than one core but mutual exclusion has to be guaranteed.

- *void rt_perf_conf(rt_perf_t *perf, unsigned events)* This function is used to configure which event has to be monitored by the performance counters. As said before the number of events that can be monitored concurrently depends on the architecture and platform on which the algorithm is running. The function requires the following parameters:

  – *rt_perf_t *perf*. It is a pointer used to access to the structure that contains the events configuration of the counters and results of the counts.

  – *unsigned events*. This parameter indicates which events have to be monitored. In particular it has to be equal to a mask in which each bit corresponds to an event that can be monitored. This mask is set exploiting the performance event identifiers that indicate which bit has to be equal to one in order to monitor a certain event. Consequently these identifiers highlight how many bits toward left a one has to be shifted in order to activate the desired event. In other words the parameter has to be equal to *1«"EVENT_IDENTIFIER"* where *EVENT_IDENTIFIER* has to be substituted with the identifier related to the event that has to be monitored. It is also possible, when there are enough counters, to monitor more than one event by making the bitwise *OR* between two or more events. This is done in this way:

    1«"EVENT_IDENTIFIER1" | 1«"EVENT_IDENTIFIER2"

    There many possible event identifiers and just some of them will be described in this thesis (the complete list can be found in [20]):

    ∗ *RT_PERF_CYCLES*. This indicates the shift necessary to monitor the total number of clock cycles used to execute the code. This count takes into account also the cycles in which the monitored core is in standby. The timer used to monitor this event is shared between all the cluster cores and consequently any action on this timer will be perceived by all cores. This event is the only one for which a shared timer is used.

    ∗ *RT_PERF_ACTIVE_CYCLES*. This identifier allows to count the number of clock cycles in which a core is active.

    ∗ *RT_PERF_INSTR* configures a performance counter to count the number of instructions executed.

- *static inline void rt_perf_reset (rt_perf_t *perf)* is used to set the count of all performance counters to 0. This function operates only on the hardware performance counters and not on what is stored in the *rt_perf_t perf* structure. *rt_perf_t *perf* is a pointer used to access to the structure that contains the events configuration of the counters and the results of the counts.

- *static inline void rt_perf_start (rt_perf_t *perf).* This function is used to start the count of the events set by using the configuration utility. By using this function it is possible to start counting in every point of the code. Again *rt_perf_t *perf* is a pointer used to access to the structure called *perf.*

- *static void rt_perf_stop (rt_perf_t *perf).* It stops the monitoring of the events set by means of the configuration function. The argument *rt_perf_t *perf* is a pointer used to access to the structure called *perf.*

- *static unsigned int rt_perf_read (int id).* This function allows to get the count results from directly the performance counters. It allows to obtain the count value with few instructions. The return value is the result of the count. Instead the argument of the function specifies the event for which the reading has to be done. It has to be one of the event identifiers seen before.

By placing correctly the start and stop functions in the program it is possible to limit the count to a specific portion of code. In particular this feature was used to measure the performance of the turbo decoding algorithm. In fact after the initialization and configuration that have been done only at the beginning of the program, the start and stop functions were used to evaluate the performance of the decoding loop that was also the parallel section of the program (the concept of parallel section is explained in subsection 4.2.2). In particular the start function has been placed suddenly before the loop in which the iterative decoding algorithm was executed, instead the stop function was placed after the conversion of the final LLRs, obtained in the last decoding iteration, in bits. Just before the start function a reset function has been used, in order to initialize to zero the counters. In this case the event that has been measured was the total number cycles. Consequently the start and stop functions were performed only by the core zero, because as said before the timer used to evaluate the total number of cycles is shared between the cores and consequently the mutual exclusion has to be guaranteed. Since, as seen in subsection 4.2.2, each core can conclude the execution of a parallel section in a different moment, in order to evaluate the complete execution of the parallel section it is necessary to wait that each core finishes its execution before stopping the counter. Consequently just before the performance counter stop function a *rt_team_barrier* should be inserted. By exploiting this approach if the event under analysis is the total number of cycles, the core that performs the functions to start, stop and get the count is not important. In fact if the core that manages the count finishes its execution before the others the *rt_team_barrier* function sends it to sleep until the other cores terminate their execution, and as said before the total number of cycles includes also the cycles in which the core is in standby. In the turbo decoding algorithm the core 0 was chosen because it was the core that has been used to manage the conversion of LLRs in bits and since this part of the program took part in the decoding process the cycles necessary to its execution had to be taken into account. The code that highlights all this stuff is shown in 4.5. The reason why the decoding loop was executed only if *rt_core_id()* is lower than P is explained in subsection 4.2.6.

Listing 4.5. Performance counters usage

```
if(rt_core_id()==0)
{
        rt_perf_reset(&perf);
        rt_perf_start(&perf);
}

if(rt_core_id()<cluster_side->P){
/// iteration decoding loop
for (curr_iter=cluster_side->min_iter;
curr_iter<cluster_side->max_iter; curr_iter++)
{

SISO1(1, cluster_side->NS, llr2_out,
llr_in[(curr_frame-1)%2]+2*(cluster_side->K+3), SM_INIT,
cluster_side->P, cluster_side->W, cluster_side->NWP,
cluster_side->EXT_MAX_VAL, cluster_side->EXT_MIN_VAL,
siso_memory[1], dec1, ic, ic_size, beta_init[1]);

if(rt_core_id()==0)
        descrambler(perm, llr2_out, llr1_out, cluster_side->K);


SISO1(0, cluster_side->NS, llr1_out, llr_in[(curr_frame-1)%2],
SM_INIT, cluster_side->P, cluster_side->W, cluster_side->NWP,
cluster_side->EXT_MAX_VAL, cluster_side->EXT_MIN_VAL, siso_memory[0],
dec1, ic, ic_size, beta_init[0]);

if(rt_core_id()==0)
        scrambler(perm, llr1_out, llr2_out, cluster_side->K);
}

}
rt_team_barrier();
\\LRRs conversion
if(rt_core_id()==0)
{
        for (i=0; i<cluster_side->K; i++)
        {
        if (dec1[i] >= 0)
                bit_dec = 1;
        else
                bit_dec = 0;

        }
```

```
            rt_perf_stop(&perf);
            printf("Core=%d Number of total cycles:%d\n",
            rt_core_id(), rt_perf_read(RT_PERF_CYCLES));
    }
```

4.2.6 Frame storing

Until now the algorithm was able to decode only one frame every execution of the program. In order to simulate better the situation in which an encoder sent the encoded frames to the decoder some modifications to the program have been done. In particular one of the possible eight cores has been used to provide the encoded frames. This implies that the maximum number of cores that could be used in parallel without modifying the hardware was seven. In C this was done by providing to the fork function a number of cores equal to the ones used for the parallelization plus one. The added one should be used to provide the encoded frames. So in this situation this core has been considered as the encoder. It didn't make the encoding operation but it took the already encoded frames. Besides, a structure in which the memory, that stored the encoded inputs, had enough space to store two frames was considered. Ideally the last feature requires the structure shown in figure 4.4. It is characterized by two buffers that take as inputs the frames provided by the encoder. The frames are written alternatively in the two buffers. In other words if a buffer contains a frame not already decoded the next frame has to be written in the free buffer. If both buffers are full the encoder should wait until almost one of the two frames stored in the two buffers is completely decoded. On the other side, also the decoder has to wait until at least an encoded frame is available. This approach needs two flags for each buffer:

- the first flag is used to signal to the encoder if the frame stored in the considered buffer has already been decoded. If its decoding was already completed the frame can be substituted with the next encoded frame otherwise the next buffer has to be taken into account.
- the second flag instead is used to signal to the decoder if any frame is present in the buffer under analysis. If the buffer is empty the decoder has to consider the next one. If all buffers are empty the decoder has to wait until at least an encoded frame is produced and stored by the encoder.

This approach has been implemented by using the L1 or L2 memories. In particular the flags had to be stored always in L1, instead the input frames could be stored in any of the two memories according to the wanted performance. The management of the two flags has been done by using *test and set* utilities provided by the PULP development kit. As known the test and set routine makes two operations atomically:

- A normal load to the address of the variable that is under test. In this way it is possible to test if the flag was already set previously.
- A store operation to the flag address. The value stored indicates if any process is in the critical section.

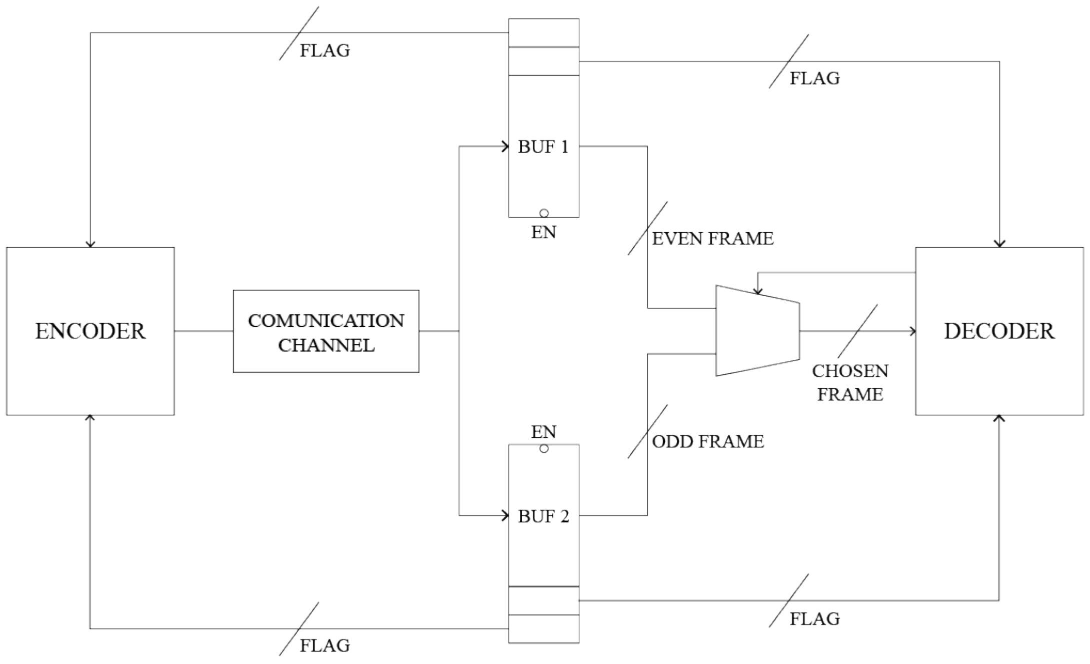

Figure 4.4. Double Frame structure

The test and set functions are usually used to implement a binary semaphore called lock. It allows to make a synchronization between different processes. In this approach the test and set flag is commonly called lock. Let's define that a region is locked when the value of the flag is one and unlocked when the value of the flag is zero. Consequently when a process runs the test and set instruction it sets the lock to one and the next processes that should execute a section protected by that binary semaphore have to wait until the region is unlocked or in an other words until the lock flag is set to zero. The values that indicate the status of the semaphore depend on the platform on which the test and set utility is implemented. As said before when a process finds out that the lock flag is set to one has to wait until the region that it wants to execute is unlocked. Consequently the lock flag should be tested continuously. In other words a sort of polling of the lock flag has to be done. In order to do this a while loop is usually exploited. A pseudo code of the lock and unlock procedures is shown in 4.6.

Listing 4.6. Lock function

```
int TAS(int *lock)
{
```

```
            int old_value = *lock
            *lock = 1;
            return old_value;
}

int main()
{
            while(TAS(lock)==1);
            protected section;
            *lock=0;
}
```

In Pulp the functioning of the TAS utility is the same explained before. In this system the value assigned to the lock flag when the protected region is locked is equal to '-1'. Any other value can be provided to the lock flag when the protected region is not locked. The functions that implement the TAS are the following

- *static inline void int rt_tas_lock_32(unsigned int addr).* It makes atomically two actions: a load in the address specified by the argument *unsigned int addr* and a writing of the value '-1' in the same address. This locks the region protected by the TAS utility. The value loaded before is also the return value of the function.
- *static inline void int rt_tas_unlock_32(unsigned int addr, unsigned int value).* This function is used to unlock the protected region. In particular this function makes an atomic writing of the value indicated in the parameter *unsigned int value* at the address set in the argument *unsigned int addr.*

The TAS explained above are implemented exploiting a 32 bits flag. There are also other two TAS utilities that exploit a 8 bits or 16 bits flag. Unfortunately they present some bugs and for this reason the 32 bits TAS has been used.

As said before these two functions can be used to manage the two flags necessary to control the flux of the encoded frames. By considering the fact that a core should be always used to fill the memory with the frames (this core in this case represents the encoder) a synchronization between the encoder and decoder cores has been implemented. Consequently four flags, two for each "buffer" (that in this platform were two memory locations), has been used. In order to check the "buffers" alternatively, it has been decided that a "buffer" had to be dedicated to the even frames instead the other one had to be used for the odd frames. This approach required two counters. They were exploited to count respectively the number of encoded and decoded frames. By taking the rest of the division by two of the counters it was possible to detect if the frame under analysis was even or odd and consequently to select the right "buffer" in which store or from which retrieve the frame. In order to simulate better the encoder-decoder data exchange, at the beginning the encoded frame (even or odd) was stored in the L2 memory, and then when the "buffer" (even or odd) was empty the frame was transferred from the L2 memory to the right "buffer". The pieces of code that implement the buffer filling and the synchronization are shown in 4.7 and 4.8. They show that, as said before, there are four flags called tas1, tas2, tas3 and tas4. The flags tas1 and tas2 are used to regulate how the data are retrieved from the memory. They are initialized to "-1" to indicate that no frame is present in the memory at the beginning. Then when the encoder core fills one of the memory location with a frame, if the frame is even tas1 is set to zero otherwise tas0 is set to zero. As shown in 4.8, according to the frame counter, if the frame is not already available in memory the core zero remains locked. Since no core can start the decoding procedure until each core is available, blocking the core zero will stop all the cores used to execute the parallel section. The other two flags tas3 and tas4 are used to regulate the encoding rate. In fact if the decoder has not already decoded the frames stored in the memory, they can not be substituted. When a frame is stored one of the two flags is set to "-1". If the frame is even tas3 will be set to "-1" otherwise "tas4" will be set to "-1" as shown in 4.7. When the frame is decoded the related flag will be set to zero as shown in 4.8. At this point a new frame can be stored in the memory location under analysis.

Note that the cores used to implement the decoding process are the first P cores (from 0 to P-1). Instead the core that fills the memory with the encoded frames is the Pth core. P is the parameter that sets the parallelism degree of the algorithm.

A qualitative block diagram of the structure exploited to exchange data between the core P and the other cores is shown in figure 4.5.

Since an extra core has been used the parameters provided to *rt_cluster_call* should become:

*rt_cluster_call(NULL, 0, execution,(void *) fc_side, NULL, 0, 0, fc_side->(P+1), NULL)*

Listing 4.7. Encoder core

```
if(rt_core_id()==cluster_side->P)
{
        \\even frame
        if(((curr_frame-1)%2)==0){
                while (rt_tas_lock_32((unsigned int)&tas3) == -1);
                for(i=0;i<cluster_side->N;i++)
                \\moving of the even frame in the L1 memory
                \\llr_in_L2 = encoded frame in L2
                \\llr_in = encoded frame in L1
                        llr_in[(curr_frame-1)%2][i]=llr_in_L2[i];
                rt_tas_unlock_32((unsigned int)&tas1,0);
        }
        \\odd frame
        if(((curr_frame-1)%2)==1){
                while (rt_tas_lock_32((unsigned int)&tas4) == -1);
                for(i=0;i<cluster_side->N;i++)
                \\moving of the odd frame in the L1 memory
                \\llr_in_L2 = encoded frame in L2
                \\llr_in = encoded frame in L1
```

```
                    llr_in[(curr_frame-1)%2][i]=llr_in_L2[i];
                rt_tas_unlock_32((unsigned int)&tas2,0);
        }
}
```

Listing 4.8. Decoder synchronization

```
if(rt_core_id()==0)
{
        //some code
    if(((curr_frame-1)%2)==0)
                while (rt_tas_lock_32((unsigned int)&tas1) == -1);
        if(((curr_frame-1)%2)==1)
                while (rt_tas_lock_32((unsigned int)&tas2) == -1);
        //some code
}

//decoding procedure

if(rt_core_id()==0)
{
        //some code

        if(((curr_frame-1)%2)==0)
        rt_tas_unlock_32((unsigned int)&tas3,0); //even frame decoded
        if(((curr_frame-1)%2)==1)
        rt_tas_unlock_32((unsigned int)&tas4,0); //odd frame decoded
}
```

÷

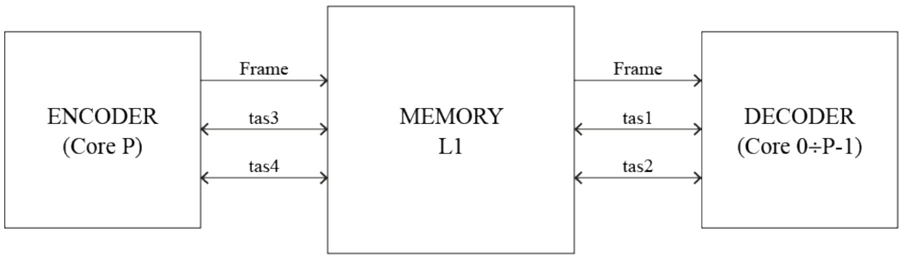

Figure 4.5. Synchronization exchanges

Chapter 5

PULP Synthesis

In order to evaluate the performance of the algorithm it was necessary to find out the maximum achievable frequency of the PULP system on which the Turbo Decoder has been run. Since the execution of the decoding algorithm needed mainly the cluster, only that part of the system has been synthesized. In order to avoid that the memories included in the cluster domain were synthesized as flip flops they have been removed from the design. This has been done by commenting their module instantiations in the top entity (that in this case was the cluster). Since the main goal of this thesis is to evaluate the performance of the algorithm, the clock gating, used for reducing the power consumption, is not so important. Consequently it has been decided to remove the clock gating implemented in the design by modifying the cell used to implement it. In particular in the clock gating cell the input has been simply connected to the output removing all the logic used to implement the clock gating mechanism. In this way only one file has been modified and not all the files in which the clock gating module is used. The only drawback was that the control signals used to implement the clock gating remained unconnected. This has been easily solved by using a compiler command during the synthesis.

The synthesis has been done by using *Synopsys Design Compiler* and the technology library UMC 65nm. The flow of commands used on Design Compiler was the following:

- Step1: import all the PULP source system verilog files by issuing the *analyze* command.
- Step2: force Design compiler to preserve the rtl names in the netlist.
  This allows to estimate power consumption easily. The command used is
  set power_preserve_rtl_hier_names true.
- Step3: in this step Design Compiler uses general gates to implement the entity specified in the *elaborate* command. In order to represent the entity by using the gates seen before a graph is used. In this case, since only the cluster has been synthesized, the following command has been issued: *elaborate pulp_cluster -architecture verilog -library WORK*. In this command *pulp_cluster* was the top entity that represents the cluster.
- Step4: if some ports in the design are not connected in order to remove them from

the netlist the following command is issued *remove_unconnected_ports [get_cells -hierarchical *]*. Despite the fact that usually unconnected ports don't have any effect on the synthesis in terms of functionality, removing them could be a good practice in order to eliminate the warnings that they produce [21].

- Step5: applying clock constraint to the design. In order to do this a symbolic clock signal is created and then bounded to the real clock pin present in the entity under analysis. Finally the frequency constraint has to be chosen. In this thesis the goal is to find the maximum frequency of the design, consequently it was possible to start by setting a clock period equal to zero by issuing *create_clock -name MY_CLK -period 0 clk_i* where *MY_CLK* was the symbolic clock signal and *clk_i* was the clock pin declared in the top entity. The slack obtained from the timing analysis gave in theory the minimum clock period achievable by the design. But this was not true because if a bigger frequency was set, *Design Compiler* could be able, by making some modification in the implementation of the design, to achieve that frequency with a slack equal to zero. So it is possible to say that in general the slack obtained by setting the clock period to zero is the starting point for a series of synthesis used to find out the real maximum frequency of a design.

- Step6: the clock is not a normal signal and consequently no DC optimization and buffer are used on that signal. This was obtained by using the command
  set_dont_touch_network MY_CLK.

- Step7: an other constraint that has to be set is the jitter of the clock signal. This was done by issuing *set_clock_uncertainty 0.07 [get_clocks MY_CLK]*

- Step8: both inputs and outputs can be affected by a delay with respect to the clock signal. Consequently the maximum constraint for this delay was set by issuing the commands:
  set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
  set_output_delay 0.5 -max -clock MY_CLK [all_outputs]

- Step9: in the last step the load of the outputs of the design has to be set. In the technology library a buffer called BUFM4R has been found. Its characteristics can be found in the file that describes the technology library. The input port of this buffer is called *A* and the outputs has been linked to it by issuing these two commands:
  set OLOAD [load_of uk65lscllmvbbr_120c25_tc/BUFM4R/A]
  set load $OLOAD [all outputs]

After all these steps the synthesis can be run issuing the command *compile*. In order to see clearly the results of the synthesis the following commands has to be issued:

- *report_timing* is used to obtain the results of the timing analysis carried out during the synthesis. In particular it shows if the frequency constraint imposed during the creation of the clock is met by the longest path in the design. This is clearly indicated by a *VIOLATED* or *MET* wording. In both cases the *slack* indicates how much has to be reduced or increased the frequency to achieve the minimum clock period supported by the selected design. As said before if the set clock frequency meets the longest path delay and the slack is zero, it does not mean that the highest frequency has been reached. Design compiler could be able to achieve a higher frequency by making more optimizations.

- *report_power* is used to obtain power consumption of the design.
- *report_area* gives the combinational, non combinational and total area of the design.

By applying the synthesis procedure to the PULP cluster it has been obtained a minimum period equal to 2.57 ns that corresponds to a frequency of 389 MHz. Just for information the power consumption and area of the design are reported respectively in tables 5.1 and 5.2.

Table 5.1. Power consumption estimated with Synopsys

Power report	
Cell Internal Power	156.1201 mW
Net Switching Power	3.3403 mW
Total Dynamic Power	159.4604 mW
Cell Leakage Power	100.7011 μW

Table 5.2. Area estimated with Synopsys

Area	μm^2
Combinational	977558.769464
Non combinational	431657.282804
Total cell Area	1409216.052269

Chapter 6

Results and Comparison

In this chapter the results in terms of performance obtained by running the algorithm on PULPino and PULP are evaluated. Then a comparison with other systems implementing the turbo decoding algorithm on a general purpose platform will be carried out.

6.1 PULPino results

In order to evaluate the performance of the turbo decoding algorithm running on PULPino platform two main elements have been be computed:

- the number of clock cycles necessary to conclude the execution of the algorithm.
- the frequency of PULPino that is the system on which the algorithm was run.

The clock cycles necessary to complete the algorithm have been computed by exploiting the performance counters presented in section 4.1. The two functions *cpu_perf_conf_events* and *perf_reset* have been placed before the decoding loop, instead the function *perf_stop* was placed just after the piece of code in which the output LLRs were converted in bits. Then the result of the performance counter was retrieved by using the function *cpu_perf_get* and printed on the UART by exploiting the function *printf*. The event that has been counted was the number of clock cycles that is identified by the *event ID 0*. This ID has been provided to the configuration function by exploiting the utility *SPR_PCER_EVENT_MASK*. Instead the function used to retrieve the results took the event ID directly as input. The code that implements this approach is shown in 6.1.

Listing 6.1. Performance counter usage

```
cpu_perf_conf_events(SPR_PCER_EVENT_MASK(0));
perf_reset();
/// iteration loop
for (curr_iter=min_iter; curr_iter<max_iter; curr_iter++)
{
        SISO1(1, llr2_out, llr_in+2*(K+3), SM_INIT,
        siso_memory[1], dec1, ic, ic_size, beta_init[1]);

        descrambler(perm, llr2_out, llr1_out);
        SISO1(0, llr1_out, llr_in, SM_INIT, siso_memory[0],
        dec1, ic, ic_size, beta_init[0]);

        scrambler(perm, llr1_out, llr2_out);
}
for(i=0;i<K;i++)
{
        if(dec1[i]>=0)
                bit_dec =1;
        else
                bit_dec=0;
}
perf_stop();
printf("cpu cycles=%d\n",cpu_perf_get(0));
```

The clock cycles have been computed in four different cases, for P equal to one, two , four and six, where P is the degree of parallelism of the algorithm. The other decoding parameters were:

- Uncoded block size that was named as K and it has been set to 1536.
- Window size that was named as W and it has been set to 32.
- Minimum number of iterations that was named as *min_iter* and it has been set to 1.
- Maximum number of iterations that was named as *max_iter* and it has been set to 8.
- The number of frames that had to be decoded that was called *max_frame* and it has been set to 1 since the evaluation has been carried out for only one frame.
- The number of fractional bits used to represent the intrinsic information. This parameter was called *NBF_INT* and it was set to one.
- Maximum value of the extrinsic information that was called *EXT_MAX_ VAL* and it has been set to 127.
- Minimum value of the extrinsic information that was called *EXT_MIN_ VAL* and it has been set to -128.
- The number of bits used to represent the state metrics *alpha* and *beta*. It was called *NB_SM* and it has been set to 12.
- The first frame number that was called *f_num* and it was set to 1.
- The number of states for each trellis stage that was called *NS* and it has been set to 8.

Table 6.1. PULPino clock cycles estimation for a Window equal to 32

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	10694608	1336826.00
2	11238794	1404849.25
4	11237312	1404664.00
6	11188259	1398532.38

The results obtained are the ones shown in table 6.1.

For what concern the second requirement, the specification of the system under analysis can be found in [22]. In particular since the algorithm exploited mainly the processor implemented in the microcontroller, the processor frequency should be sufficient to make a good estimation of the algorithm performance. For a 65nm technology the internal delay of PULPino processor (called RI5CY) is equal to 1.15 ns. Since in this analysis the memories were not considered the internal delay was sufficient to carry on a performance evaluation. Consequently the frequency of the processor has been considered equal to 869.56 MHz. By multiplying the critical path delay of the RI5CY for the clock cycles necessary to conclude the execution of the algorithm the results indicated in table 6.2 have been obtained.

Table 6.2. PULPino performance estimation for a Window equal to 32

P	Total decoding delay ms	Decoding delay for iteration ms	Mbits/s
1	12.30	1.54	1.00
2	12.92	1.62	0.95
4	12.92	1.62	0.95
6	12.87	1.61	0.96

From the tables 6.1 and 6.2 it is possible to see how, by changing the parallelism degree, the performance remains almost the same with a very small and negligible difference in terms of delay and clock cycles. This is due to the fact that the system is characterized by a single core and consequently it is not able to exploit the parallelism degree provided by the algorithm. This is the main reason why also the PULP system has been used for the implementation of the decoding algorithm.

The same results have been computed for two other window sizes: 16 and 64. The number of clock cycles used to conclude a decoding operation is shown in tables 6.3 and 6.4. As seen previously the frequency at which PULPINO works is 869.56 MHz that corresponds to a period of 1.15 ns. Consequently by multiplying the critical path delay of the RI5CY for the clock cycles necessary to conclude the execution of the algorithm the results indicated in tables 6.5 and 6.6 have been obtained. It can be notice that, by changing the window size, the number of clock cycles necessary to conclude the decoding operation has a very small variation. The maximum variation is about 2.28 %. The percentages of which the number of clock cycles varies passing from a smaller window to a bigger one are reported in table 6.7. In the table two windows are divided by an arrow. All the percentages have been computed exploiting the formula 6.1. A negative percentage means that by setting a window the number of clock cycles necessary to terminate the decoding program is lower than the one necessary when the window size set is bigger. In the other cases the percentage is positive.

$$\text{Percentage} = \frac{(\text{clock cycles smaller window} - \text{clock cycles bigger window}) \cdot 100}{\text{clock cycles smaller window}} \tag{6.1}$$

Table 6.3. PULPino clock cycles estimation for a Window equal to 16

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	10815544	1351943.00
2	11393064	1424133.00
4	11382773	1422846.62
6	11349172	1418646.50

Table 6.4. PULPino clock cycles estimation for a Window equal to 64

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	10618448	1327306.00
2	11133790	1391723.75
4	11177894	1397236.75
6	11214235	1401779.38

6.2 PULP results

The approach exploited to evaluate the performance that the turbo decoding algorithm can achieve on the PULP system is very similar to the one used in section 6.1. The main difference is that the frequency of the system was not evaluated exploiting an external source but synthesizing PULP. The performance evaluation process exploited two main steps:

- Evaluation of the clock cycles necessary to execute the turbo decoding algorithm.
- Evaluation of the frequency of the system on which the algorithm had to be executed.

The clock cycles have been evaluated exploiting the so called performance counters. They are explained in subsection 4.2.5 and the code sequence in which they are implemented is indicated in 4.5.

Table 6.5. PULPino performance estimation for a Window equal to 16

P	Total decoding delay ms	Decoding delay for iteration ms	Mbits/s
1	12.44	1.55	0.99
2	13.10	1.64	0.94
4	13.09	1.64	0.94
6	13.05	1.63	0.94

Table 6.6. PULPino performance estimation for a Window equal to 64

P	Total decoding delay ms	Decoding delay for iteration ms	Mbits/s
1	12.21	1.53	1.01
2	12.80	1.60	0.96
4	12.85	1.61	0.96
6	12.90	1.61	0.95

The clock cycles evaluation has been carried on for P equal to 1, 2, 4 and 6 and for W equal to 16, 32 and 64. P is defined as the parallelism degree of the algorithm. W is the window size. The other decoding parameters were the same set in subsection 4.2.5:

- Uncoded block size that was named as K and it has been set to 1536.
- Minimum number of iterations that was named as *min_iter* and it has been set to 1.
- Maximum number of iterations that was named as *max_iter* and it has been set to 8.
- The number of frames that had to be decoded that was called *max_frame* and it has been set to 1 since the evaluation has been carried out for only one frame.
- The number of fractional bits used to represent the intrinsic informations. This parameter was called *NBF_INT* and it was equal to one.
- Maximum value of the extrinsic informations that was called *EXT_MAX_VAL* and it has been set to 127.
- Minimum value of the extrinsic informations that was called *EXT_MIN_VAL* and it has been set to -128.
- The number of bits used to represent the state metrics *alpha* and *beta*. It was called *NB_SM* and it has been set to 12.
- The first frame number that was called *f_num* and it was equal to 1.
- The number of states for each trellis stage that was called *NS* and it has been set to 8.

Table 6.7. PULPino clock cycles comparison between different windows

P	16->64 (%)	16->32 (%)	32->64 (%)
1	1.82	1.12	0.71
2	2.28	1.35	0.93
4	1.8	1.28	0.53
6	1.19	1.42	-0.23

The following reasoning has been carried on for a window size equal to 32. A similar approach has been used for the other two window sizes considered.

As explained in section 4.2 the way in which the algorithm has been implemented on the PULP system has been modified many times. In particular there were mainly three phases:

- First case: all the data of the algorithm were declared as int and placed in the L2 memory.
- Second case: the parallelism of the data was reduced according to the bits necessary for their representation but they were still located in the L2.
- Third case: all the data have been moved in the L1 memory.

The performance achieved with these three different approaches has been evaluated in order to see how much the parallelism and the placement of the data inside the system affect the performance of the algorithm. The number of clock cycles necessary to conclude the algorithm in all the three cases is shown respectively in tables 6.8, 6.9 and 6.10.

Table 6.8. PULP first case clock cycles estimation for W equal to 32

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	30191474	3773934.25
2	15794946	1974368.25
4	8621165	1077645.62
6	6250461	781307.62

The second step necessary to evaluate the performance of the algorithm was to derive the frequency of the system on which it had to be run. In this case the system was PULP and in particular its cluster which was the only part necessary for the decoding. In order to derive the critical path delay of the cluster a synthesis with Synopsys Design Compiler has been done as explained in chapter 5. The critical path delay obtained was equal to 2.57 ns that corresponds to a frequency of 389 MHz. By multiplying the delay computed with the synthesis for the clock cycles necessary to conclude the execution of the algorithm, the results indicated in tables 6.11 for the first case, 6.12 for the second case and 6.13 for the third case have been obtained.

Table 6.9. PULP second case clock cycles estimation for W equal to 32

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	31900190	3987523.75
2	16604321	2075540.12
4	8968990	1121123.75
6	6440463	805057.88

Table 6.10. PULP third case clock cycles estimation for W equal to 32

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	11949566	1493695.75
2	6085739	760717.38
4	3153822	394227.75
6	2180823	272602.88

The first thing that can be seen from the results displayed in tables 6.11, 6.12 and 6.13 is that, independently from the approach used, the increase of the parallelism degree improves the performance. This is due to the fact that the platform used is characterized by a number of processors equal to eight and consequently it is possible to exploit the parallelism degree of the algorithm. In particular the number of cores used depends on the value assumed by P. To understand the real performance boost obtained by using more than one core it is possible to compute the ratio between the total delay obtained exploiting more than one core, and the delay with just one core. These results are shown in tables 6.11, 6.12 and 6.13. They show that, as expected, the performance boost obtained by increasing the parallelism degree is not equal to the number of cores used but it is a little bit lower. This is due to the fact that α and β initialization and interleaving/deinterleaving operations can not be executed in parallel.

The values shown in the three tables 6.11, 6.12 and 6.13 show an other important area of reflection. The throughput obtained by storing all the data in the L1 memory is much higher than the one obtained in the other two cases. This was expected since the L1 memory is much closer to the cluster than the L2 one and the overhead necessary to exchange data is much lower. In fact when a data stored in L2 has to be used by the cluster, it has to be moved to the L1 memory from which can be retrieved. Obviously this represents an overhead because the data used in the algorithm will be moved many times between the L2 and the L1 memories. Instead by storing the data directly in the L1 memory they will be read and written directly in the L1 reducing drastically the data exchange between L1 and L2 memories.

The variation in terms of performance between the two cases in which the data were stored in the L2 memory are relatively negligible, this is probably due to the fact that the functional units of the PULP system have a fixed parallelism and consequently their delay don't change by changing the length of the data stored in the memory.

Table 6.11. PULP first case performance estimation for W equal to 32

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	77.59	9.70	1.00	0.16
2	40.59	5.07	1.91	0.30
4	22.16	2.77	3.50	0.55
6	16.06	2.01	4.82	0.76

Table 6.12. PULP second case performance estimation for W equal to 32

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	81.98	10.25	1.00	0.15
2	42.67	5.33	1.92	0.29
4	23.05	2.88	3.56	0.53
6	16.55	2.07	4.95	0.74

For the other window sizes a similar reasoning has been carried on. All the values derived for a window size equal to 32 have been computed for a W equal to 16 and 64. The number of clock cycles for each of the three cases with W equal to 16 is shown in tables 6.14, 6.15 and 6.16. The same results for W equal to 64 are shown in tables 6.17, 6.18 and 6.19. By multiplying the delay of the critical path of PULP with the number of clock cycles the results in tables 6.20, 6.21, 6.22, 6.23, 6.24 and 6.25 have been obtained. This results bring to the same conclusions stated when the window size was equal to 32. In particular the throughput variation between the first and the second cases is negligible. Instead by using the memory L1, the throughput increases drastically with respect to the one obtained in the first two cases. It is possible to see that in all three cases reducing P implies an increase of the throughput of a factor almost equal to P (the factor is lower than P due to the overhead explained before). Instead regarding the window variations, by varying W the number of clock cycles necessary to conclude the decoding operation has a very small change. The maximum variation is about 7.26 %. The percentages of which the number of clock cycles changes by passing from a smaller window to a bigger one for the three cases described before are reported in tables 6.26, 6.27 and 6.28. In the tables are indicated two windows divided by an arrow. All the percentages have been computed exploiting the formula 6.1. A negative percentage means that by setting a smaller window the number of clock cycles necessary to terminate the decoding program is lower than the one necessary when the window size set is bigger. In the other cases the percentage is positive.

Table 6.13. PULP third case performance estimation for W equal to 32

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbit/s
1	30.71	3.84	1.00	0.40
2	15.64	1.96	1.96	0.78
4	8.10	1.01	3.79	1.52
6	5.60	0.70	5.48	2.19

Table 6.14. PULP first case clock cycles estimation for W equal to 16

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	31096972	3887121.50
2	16347574	2043446.75
4	8997226	1124653.25
6	6568559	821069.88

6.3 Results comparison

The results obtained in section 6.2 have been compared with the throughput provided by other designs on which the turbo decoding algorithm has been run. In order to make consistent comparisons the works taken into account have been selected following two guidelines:

- The algorithm used to implement the turbo decoder had to be the *Log-MAP* or the *Max-Log-MAP*. Obviously with the *Max-Log-MAP* algorithm the throughput was higher than the one obtained by using the *Log-MAP*. This is due to the fact that in the *Max-Log-MAP* the look-up table was no more present and consequently the readings from the memory were reduced.
- More than one processing element had to be present in the design and they had to be exploited to execute concurrently the SISO algorithm on different pieces of the code-word.

The designs selected are two GPU implementations of the decoding algorithm documented in [23] and [24].

The first GPU chosen exploited the CUDA approach in which there are many cores each one characterized by a SIMD architecture in which there are 8 ALUs that can be exploited to make a parallel computation inside the core itself. The cores were exploited to execute different pieces of the code word simultaneously, instead the 8 ALUs inside each core were exploited to compute the metrics of each trellis state simultaneously. This obviously increases the speed with respect to the approach used in this thesis in which each core computes serially the metrics of the piece of codeword assigned to it.

Table 6.15. PULP second case clock cycles estimation for W equal to 16

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	32508588	4063573.50
2	16974635	2121829.38
4	9220836	1152604.50
6	6653309	831663.62

Table 6.16. PULP third case clock cycles estimation for W equal to 16

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	12131620	1516452.50
2	6194231	774278.88
4	3226362	403295.25
6	2242503	280312.88

A comparison between the results obtained in [23] and the ones computed in table 6.13 has been done. The throughput computed in table 6.13 and the one obtained in [23] are related to a single decoder iteration and to the Log-MAP approach. The results are reported in table 6.29.

The values in table 6.29 show that the throughput obtained with the GPU is much higher than the PULP one. This high difference is related to many factors:

- The frequency of the Nvidia TESLA C1060 is equal to 1.3 GHz that is about 3.34 times higher than the one reached by PULP.
- The number of cores in the GPU is equal to 240. Consequently the parallelism that can be exploited is much higher than the one used by PULP. In particular in the table the minimum number of processors exploited is equal to 48 that is eight times the maximum parallelism exploited by PULP.
- The processors used by the GPU are characterized by a SIMD architecture with eight ALUs that have been exploited to compute the state metrics in a parallel way leading to a higher performance.

From the last column of the table 6.29 can be observed the throughput of the GPU with a frequency equal to the PULP one. In this case the difference in terms of throughput between the Nvidia TESLA C1060 with a parallelism degree equal to 48 and the PULP implementation with a parallelism degree equal to 6 is not so much higher. Probably doubled the number of cores used by PULP could lead to a higher normalized throughput than the one achieved by the TESLA C1060 with 48 cores. Obviously this can not be predicted in a deterministic way because the throughput usually increases of a factor lower than the number of cores used and consequently the performance that can be achieved

Table 6.17. PULP first case clock cycles estimation for W equal to 64

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	29735299	3716912.38
2	15518157	1939769.62
4	8434307	1054288.38
6	6091959	761494.88

Table 6.18. PULP second case clock cycles estimation for W equal to 64

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	31593019	3949127.38
2	16418922	2052365.25
4	8844403	1105550.38
6	6332858	791607.25

with a higher number of cores can not be predicted precisely. It is necessary to take into account that the PULP frequency doesn't consider a real implementation of the memory that if it is not enough fast can cause an increase of the critical path. Besides the values used in the GPU are float and consequently this requires much complex arithmetic unit with a higher latency.

The second GPU taken into account is a GTX Titan used to obtain the experimental results that can be seen in [24]. Even in this case the CUDA architecture is exploited with all the features seen in the first approach shown above. The main difference is that the αs and βs in each state were not computed concurrently instead the forward and backward recursions have been done simultaneously. The algorithm adopted was the Max-Log-MAP. Also in this case a parallelism degree has been exploited in order to divide the code words in sub blocks performed concurrently. In particular a P of 128 has been set. Besides in order to perform forward and backward recursions concurrently further PEs have been used. Consequently the total parallelism was 256. The results highlighted that the throughput didn't increase of a factor equal to 256 due to the overhead related to the computations that could not be parallelized in the algorithm as written in [24].

The GTX Titan has a clock rate equal to 837 MHz [25] consequently a normalization with respect to the frequency used by PULP has been done. The results of the design used in this thesis and of the one described in [24] are shown in table 6.30. This table shows that the throughput of the implementation with the GTX Titan is much higher than the one reached by PULP. This is mainly due to four facts:

- The parallelism degree exploited with the Nvidia GTX Titan is much higher than the one used by PULP.
- The design that exploited the Titan GPU performed the forward and backward recursions concurrently.

Table 6.19. PULP third case clock cycles estimation for W equal to 64

P	Clock cycles	$\frac{\text{Clock cycles}}{\text{Number of iterations}}$
1	11856981	1482122.62
2	6030114	753764.25
4	3122395	390299.38
6	2150189	268773.62

Table 6.20. PULP first case performance estimation for W equal to 16

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	79.92	9.99	1.00	0.15
2	42.01	5.25	1.90	0.29
4	23.12	2.89	3.46	0.53
6	16.89	2.11	4.73	0.73

- Higher frequency of the Titan.
- The faster Max-Log-MAP algorithm was exploited.

The ratio between the normalized throughput of the Nvidia GTX Titan and the one of the PULP configuration with the highest P is equal to 4.78. This is much lower than the ratio between the parallelism degree of the two configurations under analysis, that is equal to 42.67 (in this computation the total parallelism of the Titan, equal to 256, is considered). This probably means that by increasing the parallelism degree of the configuration used in this thesis could be possible to match and maybe overcome the normalized throughput obtained with the Nvidia GTX Titan.

Table 6.21. PULP second case performance estimation for W equal to 16

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	83.55	10.44	1.00	0.15
2	43.62	5.45	1.92	0.28
4	23.70	2.96	3.53	0.52
6	17.10	2.14	4.88	0.72

Table 6.22. PULP third case performance estimation for W equal to 16

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbit/s
1	31.18	3.90	1.00	0.39
2	15.92	1.99	1.96	0.77
4	8.29	1.04	3.75	1.48
6	5.76	0.72	5.42	2.13

Table 6.23. PULP first case performance estimation for W equal to 64

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	76.42	9.55	1.00	0.16
2	39.88	4.98	1.92	0.31
4	21.68	2.71	3.52	0.57
6	15.66	1.96	4.87	0.78

Table 6.24. PULP second case performance estimation for W equal to 64

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbits/s
1	81.19	10.15	1.00	0.15
2	42.20	5.27	1.92	0.29
4	22.73	2.84	3.57	0.54
6	16.28	2.03	5.00	0.76

Table 6.25. PULP third case performance estimation for W equal to 64

P	Total decoding delay ms	Decoding delay for iteration ms	$\frac{\text{total delay for one core}}{\text{total delay for P cores}}$	Mbit/s
1	30.47	3.81	1.00	0.40
2	15.50	1.94	1.96	0.79
4	8.02	1.00	3.81	1.53
6	5.52	0.69	5.52	2.22

Table 6.26. PULP first case clock cycles comparison between different windows

P	16->64 (%)	16->32 (%)	32->64 (%)
1	4.38	2.91	1.51
2	5.07	3.38	1.75
4	6.26	4.18	2.17
6	7.26	4.84	2.54

Table 6.27. PULP second case clock cycles comparison between different windows

P	16->64 (%)	16->32 (%)	32->64 (%)
1	2.82	1.87	0.96
2	3.27	2.18	1.12
4	4.08	2.73	1.39
6	4.82	3.20	1.67

Table 6.28. PULP third case clock cycles comparison between different windows

P	16->64 (%)	16->32 (%)	32->64 (%)
1	2.26	1.50	0.77
2	2.65	1.75	0.91
4	3.22	2.25	1.00
6	4.12	2.75	1.40

Table 6.29. Comparison between PULP and Nvidia TESLA C1060

Implementation	P	Mbit/s	Normalized throughput with respect to 389 MHz
PULP	1	0.40	0.40
	2	0.78	0.78
	4	1.52	1.52
	6	2.19	2.19
Nvidia TESLA C1060	48	12.19	3.65
	64	19.50	5.84
	96	23.87	7.14
	192	36.59	10.95

Table 6.30. Comparison between PULP and Nvidia GTX Titan

Implementation	P	Mbit/s	Normalized throughput with respect to 389 MHz
PULP	1	0.40	0.40
	2	0.78	0.78
	4	1.52	1.52
	6	2.19	2.19
Nvidia GTX Titan	128	22.46	10.46

Chapter 7

Conclusions and further improvements

In the past many researchers have demonstrated that by exploiting a parallel implementation of the turbo decoding it is possible to achieve a throughput comparable with the ASIC and DSP implementations but with much more flexibility, lower design cost and a higher portability. This for example can be seen in the papers [23], [24], taken as reference for the comparison, and in other works that can be found searching the Web. But in this thesis something different has been carried on. The systems used for the implementations seen in [23] and [24] exploit a Nvidia's GPU with a predefined architecture and utilities. Instead in this dissertation an open-source architecture with an open-source ISA has been exploited. This means that it is possible to change the architecture of the microcontroller by increasing, for example the number of cores, the memory size and the number of functional units present in a single core. On the other side it is also possible to add some special instructions that allow to execute faster a specific work. This modularity allows to customize the system according to the specifications of the application that has to be implemented on the system itself. Obviously system customization implies higher design cost and longer design cycle. In this thesis the hardware modularity was not so much exploited, because the priority was to give a first parallel implementation of the algorithm on the PULP-platform in order to have a starting point for future developments. Despite the fact that the actual hardware implementation is much simpler than the others seen in [23] and [24] (the number of cores is at maximum forty times lower than the one used in the reference designs) the throughput obtained is at maximum ten times lower than the one achieved by the reference implementations. This implies that only by tuning the cluster adding some other cores probably could be possible to reach the throughput achieved by other designs. Thanks to the described hardware modularity and to the large number of papers on the turbo decoding algorithm available it is possible to find some further improvements:

- In order to decrease the latency of the decoding operation it is possible to adopt some early stopping criteria that are used to stop the decoder before the end of the iterative loop. In this way the number of iterations necessary to decode a single frame is not chosen statically but dynamically according to a rule set a priori. This allows to reduce the iterations required to complete the decoding operation. Some stopping criteria are listed in [15].

- The throughput can be also increased by using the Max-Log-MAP algorithm briefly described in the subsection 3.2.3 and treated more in detail in [16]. Obviously this algorithm implies a BER performance reduction.

- The throughput can be increased by adding a further parallelization level: computing the forward and the backward metrics concurrently [23] [24]. Obviously this approach requires some hardware modifications. In particular since the states are eight and the metrics are two (α and β) sixteen concurrent executions are required for each block in which the code word is divided. In order to do this a SIMD like implementation can be exploited by placing some additional functional units able to make the highlighted computations concurrently.

- A further improvement to what said in the previous point is related to the LLR computations. If a certain parallelism was introduced in order to parallelize the metrics computation, it could be a good investment exploiting the introduced functional units to obtain a parallel computation of the LLRs. If for example a sixteen parallel degree has been introduced in the computation of the metrics the LLRs will be computed only when sixteen αs and βs are available for the LLRs computation. So for example when the αs and βs from k-8 to k+8 trellis stages have been computed the LLRs related to that trellis stages can be calculated. Obviously it could be possible to wait until all the α and β computations are finished, but in this way the number of metrics to store increases and consequently also the memory.

- An other approach that can be followed exploits the interleaving technique [15]. Since usually the input informations are organized in frames it is possible to switch continuously between n frames, this implies that in the structure will be present results of all the n frames. Since each frame has to work only with its intermediate values it is necessary to add n registers. These registers can be exploited to pipeline the SISO decoder reducing the critical path and consequently increasing the frequency of the design. The drawback is that the latency and the memory necessary to store the results increase of a factor equal to n.

- An other improvement could be increasing the number of slices in which the codeword is divided. Obviously to exploit this approach also the number of cores included in the system has to be increased. This can be a situation in which the modularity of the PULP-platform can be exploited to obtain a higher throughput.

- The last improvement could be implementing a hardware accelerator able to make the most important sequence of operations of the turbo decoding: add, compare and selection [18]. Once this hardware element is added to the design, the ISA can be modified by adding the ACS (Add Compare and Select) instruction to it.

The solutions proposed above are only a very small part of the approaches that can be found searching Web. In fact the turbo decoding is a very hot argument that is becoming even hotter with the recent introduction of the 5G standard.

The goal of this thesis was mainly to provide a starting point for future developments on this open platform. In particular a first evaluation of the performance achieved by the basic design is essential to understand if further improvements in this direction can be a good investment. From the results obtained and from the comparison with the reference designs seen in [23] and [24] it can be seen that the throughput achieved is not so lower than the one achieved by much more complex implementations. This implies that probably applying some of the techniques indicated above could lead to a much higher throughput that could probably overcome the once obtained with the Nvidia's GPUs seen in [23] and [24]. Consequently it can be stated that the results obtained in this thesis provide a good reason to develop the platform and the algorithm used in order to achieve a throughput that matches the requirements of the latest standards available.

Appendix A

CMakeLists.txt

A.1 sw/apps folder

```
set(UTILS_DIR $CMAKE_SOURCE_DIR/utils)

include(CMakeSim.txt)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# main application macro
# Optional arguments:
# - SUBDIR prefix (prefix application with prefix)
#
# Attention: Every application name has to be unique and must have its own
# build folder, so if you have multiple applications in one folder,
# use SUBDIR to separate them
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # #
macro(add_application NAME SOURCE_FILES)
# optional argument parsing
set(oneValueArgs SUBDIR TB TB_TEST LABELS FLAGS)
set(multiValueArgs LIBS)
cmake_parse_arguments(ARG "$options" "$oneValueArgs" "$multiValueArgs" $ARGN)
set(SUBDIR $ARG_SUBDIR)

include_directories(
$CMAKE_SOURCE_DIR/libs/malloc_lib/inc
$CMAKE_SOURCE_DIR/libs/string_lib/inc
$CMAKE_SOURCE_DIR/libs/sys_lib/inc
$CMAKE_SOURCE_DIR/libs/bench_lib/inc
$CMAKE_SOURCE_DIR/libs/new_lib/inc
)

if($ARDUINO_LIB)
include_directories(
$CMAKE_SOURCE_DIR/libs/Arduino_lib/core_libs/inc
$CMAKE_SOURCE_DIR/libs/Arduino_lib/separate_libs/inc
)
endif()

if($BEEBS_LIB)
include_directories(
$CMAKE_SOURCE_DIR/libs/beebs_lib/inc
)
endif()

add_executable($NAME.elf $<TARGET_OBJECTS:crt0> $SOURCE_FILES)

# set subdirectory for add_executable
if(NOT "$SUBDIR" STREQUAL "")
set_target_properties($NAME.elf PROPERTIES RUNTIME_OUTPUT_DIRECTORY
$SUBDIR)
endif()

# set optional compile flags per application
if(NOT "$ARG_FLAGS" STREQUAL "")
set_target_properties($NAME.elf PROPERTIES COMPILE_FLAGS $ARG_FLAGS)
endif()

set(CMAKE_EXE_LINKER_FLAGS "$CMAKE_EXE_LINKER_FLAGS -lm")

if($RISCY_RV32F)
set(MATH_FNS_LIB "math_fns")
else()
set(MATH_FNS_LIB "")
endif()

if($BEEBS_LIB)
set(BEEBS_LIB_NAME "beebs")
else()
set(BEEBS_LIB_NAME "")
endif()

if($ARDUINO_LIB)
set(ARDUINO_CORE "Arduino_core")
set(ARDUINO_SEP "Arduino_separate")
else()
```

```
set(ARDUINO_CORE "")
set(ARDUINO_SEP "")
endif()

# link libraries
target_link_libraries($NAME.elf $ARG_LIBS $ARDUINO_SEP $ARDUINO_CORE
$BEEBS_LIB_NAME bench $MATH_FNS_LIB string sys string new m)

# this specifies the testbench to use for simulation
if(ARG_TB)
set_target_properties($NAME.elf PROPERTIES TB $ARG_TB)
else()
set_target_properties($NAME.elf PROPERTIES TB run.tcl)
endif()

# this specifies the TEST parameter argument for the testbench (if any)
if(ARG_TB_TEST)
set_target_properties($NAME.elf PROPERTIES TB_TEST $ARG_TB_TEST)
else()
set_target_properties($NAME.elf PROPERTIES TB_TEST "")
endif()

add_custom_target($NAME.read)
add_custom_command(TARGET $NAME.read
POST_BUILD
COMMAND $CMAKE_OBJDUMP $CMAKE_OBJDUMP_FLAGS $<TARGET_FILE:$NAME.elf>
> $NAME.read
WORKING_DIRECTORY ./$SUBDIR
DEPENDS $NAME.elf)

add_custom_target($NAME.annotate)
add_custom_command(TARGET $NAME.annotate
COMMAND $UTILS_DIR/annotate.py $NAME.read
WORKING_DIRECTORY ./$SUBDIR
DEPENDS $NAME.read)

# add everything needed for simulation
add_sim_targets($NAME)

if(ARG_LABELS)
set_tests_properties($NAME.test PROPERTIES LABELS "$ARG_LABELS")
endif()

endmacro()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# boot code macro
# Used to generate the boot code for pulpino
# Attention: Every application name has to be unique and must have its own
# build folder
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # #
macro(add_boot_code NAME SOURCE_FILES)

# Compile boot code with RVC enabled
set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -mrvc")

# optional argument parsing
set(oneValueArgs SUBDIR TB LABELS FLAGS)
cmake_parse_arguments(ARG "$options" "$oneValueArgs" "" $ARGN)
set(SUBDIR $ARG_SUBDIR)

# OVERWRITE linker file for boot code!
set(CMAKE_EXE_LINKER_FLAGS "$BOOT_LINKER_FLAGS")

include_directories(
$CMAKE_SOURCE_DIR/libs/sys_lib/inc
$CMAKE_SOURCE_DIR/libs/new_lib/inc
)

add_executable($NAME.elf $<TARGET_OBJECTS:crt0_boot> $SOURCE_FILES)

# set subdirectory for add_executable
if(NOT "$SUBDIR" STREQUAL "")
set_target_properties($NAME.elf PROPERTIES RUNTIME_OUTPUT_DIRECTORY
$SUBDIR)
endif()

# set optional compile flags per application
if(NOT "$ARG_FLAGS" STREQUAL "")
set_target_properties($NAME.elf PROPERTIES COMPILE_FLAGS $ARG_FLAGS)
endif()

# link libraries
target_link_libraries($NAME.elf sys)
target_link_libraries($NAME.elf new)


# this specifies the testbench to use for simulation
```

```
if(ARG_TB)
set_target_properties($NAME.elf PROPERTIES TB $ARG_TB)
else()
set_target_properties($NAME.elf PROPERTIES TB run.tcl)
endif()

add_custom_target($NAME.read)
add_custom_command(TARGET $NAME.read
POST_BUILD
COMMAND $CMAKE_OBJDUMP $CMAKE_OBJDUMP_FLAGS $<TARGET_FILE:$NAME.elf>
> $NAME.read
WORKING_DIRECTORY ./$SUBDIR
DEPENDS $NAME.elf)

# generate verilog and cde files
file(MAKE_DIRECTORY $CMAKE_CURRENT_BINARY_DIR/$SUBDIR/boot/)
add_custom_command(OUTPUT $CMAKE_CURRENT_BINARY_DIR/$SUBDIR/boot/boot_code.sv
COMMAND $UTILS_DIR/s19toboot.py ../$NAME.s19
#COMMAND cp boot_code.cde $PULP_MODELSIM_DIRECTORY/boot/

WORKING_DIRECTORY ./$SUBDIR/boot/
DEPENDS $CMAKE_CURRENT_BINARY_DIR/$SUBDIR/$NAME.s19)

add_custom_target($NAME.install
COMMAND cp boot/boot_code.sv $CMAKE_SOURCE_DIR/../rtl/
DEPENDS $CMAKE_CURRENT_BINARY_DIR/$SUBDIR/boot/boot_code.sv
COMMENT STATUS "Copying boot code..."
)


# add everything needed for simulation
add_sim_targets($NAME)

# if there are labels available for testing
if(ARG_LABELS)
set_tests_properties($NAME.test PROPERTIES LABELS "$ARG_LABELS")
endif()

endmacro()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# # # # # # # # # # # # # # #

add_subdirectory(helloworld)
add_subdirectory(bla)

add_subdirectory(main)
add_subdirectory(read_file)

add_subdirectory(main2)
if($ARDUINO_LIB)
add_subdirectory(Arduino_tests)
endif()

add_subdirectory(bench)

if($BEEBS_LIB)
add_subdirectory(beebs)
endif()

if($RISCY_RV32F)
add_subdirectory(ml_tests)
endif()

# RISCV only tests
if($RISCV)
#add_subdirectory(compressed)
add_subdirectory(fpga)
add_subdirectory(riscv_tests)
add_subdirectory(freertos)
add_subdirectory(boot_code)
endif()
add_subdirectory(sequential_tests)
add_subdirectory(imperio_tests)

if(IS_DIRECTORY "$CMAKE_CURRENT_SOURCE_DIR/scratch/")
add_subdirectory(scratch)
endif()
```

A.2 sw folder

```
cmake_minimum_required (VERSION 2.8)

include(CMakeParseArguments)

if ($CMAKE_VERSION VERSION_GREATER 3.1.0)
set(USES_TERMINAL "USES_TERMINAL")
else()
```

```
set(USES_TERMINAL "")
endif()

# Force object file extension to be .o
set(UNIX TRUE CACHE STRING "" FORCE)

# System name
# If we set the system to Clang/GCC we get "-rdynamic"
# however we need Linux for dynamic linking stuffs.
# We should probably create a custom system name
set(CMAKE_SYSTEM_NAME "Linux-CXX")

enable_language(C CXX ASM)
project (pulp_software C)

enable_testing()

set(RISCV 1)

if($RISCY_RV32F AND $USE_ZERO_RISCY)
message(SEND_ERROR "Floating Point Extensions are not supported on zero-riscy")
endif()

# assign default architecture flags if not defined
string(COMPARE EQUAL "$GCC_MARCH" "" GCC_MARCH_NOT_SET)

if(GCC_MARCH_NOT_SET)
message("\nUsing default architecture flags!!\n")

if($USE_RISCY)
if($RISCY_RV32F)
set(GCC_MARCH "IMFDXpulpv2")
else()
set(GCC_MARCH "IMXpulpv2")
endif()
else()
if($ZERO_RV32M)
set(GCC_MARCH "RV32IM")
else()
set(GCC_MARCH "RV32I")
endif()
endif()
endif()


message(STATUS "GCC_MARCH= $GCC_MARCH")
```

```
set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -m32 -march=$GCC_MARCH -Wa,-march=$GCC_MARC
set(CMAKE_OBJDUMP_FLAGS -Mmarch=$GCC_MARCH -d)

if($GCC_MARCH MATCHES "IMFDXpulpv2")
set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -mhard-float")
endif()

if($RVC)
message("NOTE: Using compressed instructions")
set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -mrvc")
endif()

if($ZERO_RV32E)
if ($USE_RISCY)
message(SEND_ERROR "RV32E can only be used with Zero-riscy")
endif()
message("NOTE: Using RV32E ISA")
set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -m16r")
endif()

set(LDSCRIPT "link.riscv.ld")
set(LDSCRIPT_BOOT "link.boot.ld" )


set(PULP_PC_ANALYZE "pulp-pc-analyze" CACHE PATH "path to pulp pc analyze
binary")

set(CMAKE_C_FLAGS "$CMAKE_C_FLAGS -Wextra -Wall -Wno-unused-parameter -
Wno-unused-variable -Wno-unused-function -fdata-sections -ffunction-sections -fdiagnostics-
color=always")
set(CMAKE_EXE_LINKER_FLAGS "$CMAKE_EXE_LINKER_FLAGS -L$CMAKE_CURRENT_SOU
-T$LDSCRIPT -nostartfiles -Wl,–gc-sections")
set(BOOT_LINKER_FLAGS "-L$CMAKE_CURRENT_SOURCE_DIR/ref -T$LDSCRIPT_BOOT
-nostartfiles -Wl,–gc-sections")

set(CMAKE_CXX_COMPILER "$CMAKE_C_COMPILER")
set(CMAKE_CXX_FLAGS "$CMAKE_C_FLAGS")

set(CMAKE_ASM_COMPILER "$CMAKE_C_COMPILER")
set(CMAKE_ASM_FLAGS "$CMAKE_C_FLAGS")

set(PULP_MODELSIM_DIRECTORY "" CACHE PATH "Path to the ModelSim PULPino
build")
set(VSIM "vsim" CACHE FILEPATH "Path to the vsim executable")

##########################################################
```

```
if($ZERO_RV32E)
set(crt0 "ref/crt0.riscv_E.S")
set(crt0_boot "ref/crt0.boot_E.S")
else()
set(crt0 "ref/crt0.riscv.S")
set(crt0_boot "ref/crt0.boot.S")
endif()

include_directories(libs/sys_lib/inc)

include_directories(libs/new_lib/inc)

if($ARDUINO_LIB)
include_directories(libs/Arduino_lib/core_libs/inc)
include_directories(libs/Arduino_lib/separate_libs/inc)
endif()

if($RISCY_RV32F)
include_directories(libs/math_fns_lib/inc)
endif()

set_source_files_properties($crt0 PROPERTIES LANGUAGE C)
add_library(crt0 OBJECT $crt0)

add_library(crt0_boot OBJECT $crt0_boot)
set_target_properties(crt0_boot PROPERTIES COMPILE_FLAGS "-DBOOT")


#########################################################
# Other targets
#########################################################


add_custom_target(vcompile
COMMAND tcsh -c "./vcompile/build_rtl_sim.csh"
WORKING_DIRECTORY $PULP_MODELSIM_DIRECTORY
$USES_TERMINAL)

add_custom_target(vcompile.ps
COMMAND tcsh -c "env PL_NETLIST=$PL_NETLIST ./vcompile/vcompile_ps.csh"
WORKING_DIRECTORY $PULP_MODELSIM_DIRECTORY
$USES_TERMINAL)

add_custom_target(vcompile.pl
COMMAND tcsh -c "env PL_NETLIST=$PL_NETLIST ./vcompile/vcompile_pl.csh"
WORKING_DIRECTORY $PULP_MODELSIM_DIRECTORY
$USES_TERMINAL)

add_custom_target(vcompile.core.riscy
COMMAND tcsh ./vcompile/vcompile_riscv.csh
WORKING_DIRECTORY $PULP_MODELSIM_DIRECTORY
$USES_TERMINAL)

add_custom_target(vcompile.core.zero
COMMAND tcsh ./vcompile/vcompile_zero-riscy.csh
WORKING_DIRECTORY $PULP_MODELSIM_DIRECTORY
$USES_TERMINAL)

##########################################################

add_subdirectory(libs/string_lib)
add_subdirectory(libs/sys_lib)

if($RISCY_RV32F)
add_subdirectory(libs/math_fns_lib)
endif()

if($ARDUINO_LIB)
add_subdirectory(libs/Arduino_lib)
endif()

add_subdirectory(libs/new_lib)

add_subdirectory(libs/bench_lib)

set(BEEBS_LIB 0)

if($BEEBS_LIB)
add_subdirectory(libs/beebs_lib)
endif()

add_subdirectory(apps)
```

Bibliography

[1] Qing Zhang, Xueqiu Yu, Zhiyi Yu, Xiaoyang Zeng
A Turbo Decoder Implementation for LTE Downlink Mapped on a Multi-Core Processor Platform

[2] Suiping Zhang, Rongrong Qian, Tao Peng , Ran Duan, Kuilin Chen
High Throughput Turbo Decoder Design for GPP Platform

[3] Chengjun Liu, Zhisong Bie, Canfeng Chen, Xianjun Jiao
A Parallel LTE Turbo Decoder On GPU

[4] Andrew Wareman, Krste Asanovic, SiFive Inc, CS Division, EECS Department, University of California, Berkeley. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*
Document Version 2.2.

[5] *https://github.com/pulp-platform/pulpino.*

[6] Andreas Traber, Michael Gautschi, Pasquale Davide Schiavone. *RI5CY: User Manual.*
June 2018 Revision2.2
`https://github.com/pulp-platform/riscv/blob/master/doc/user_manual.doc`

[7] Andreas Traber, Michael Gautschi. *PULPino: Datasheet.*
`https://github.com/pulp-platform/pulpino/blob/master/doc/datasheet/datasheet.pdf`

[8] *https://github.com/pulp-platform/ri5cy_gnu_toolchain.*

[9] *https://github.com/pulp-platform/pulp.*

[10] *https://github.com/pulp-platform/pulp-riscv-gnu-toolchain.*

[11] *https://github.com/pulp-platform/pulp-sdk.*

[12] Antonio Pullini, Davide Rossiy, Igor Loiy, Giuseppe Tagliaviniy, and Luca Benini.
Integrated Systems Laboratory, ETH Zürich, Gloriastr. 35, 8092 Zurich, Switzerland
DEI, University of Bologna, Via Risorgimento 2, 40136 Bologna, Italy
Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing

[13] Antonio Pullini, Davide Rossi, Igor Loi, Alfio Di Mauro, and Luca Benini
Integrated Systems Laboratory, ETH Z¨urich, Gloriastr. 35, 8092 Zurich, Switzerland
DEI, University of Bologna, Via Risorgimento 2, 40136 Bologna, Italy
Mr.Wolf: A 1 GFLOP/s Energy-Proportional Parallel Ultra Low Power SoC for IoT Edge Processing

[14] *https://github.com/pulp-platform/pulp-rt-examples.*

[15] Emmanuel Boutillon, Catherine Douillard, and Guido Montorsi
Iterative Decoding of Concatenated Convolutional Codes: Implementation Issues.

[16] Lajos Hanzo, Jason P. Woodard, and Patrick Robertson
Turbo Decoding and Detection for Wireless Applications.

[17] S. VikramaNarasimhaReddy, Charan Kumar .K, Neelima Koppala
Design of Convolutional Codes for Varying Constraint Lengths.

[18] Cheng-Chi Wong, Hsie-Chia Chang
Turbo Decoder Architecture for Beyond-4G Applications

[19] Anum Imran
SOFTWARE IMPLEMENTATION AND PERFORMANCE OF UMTS TURBO CODE

[20] *https://greenwaves-technologies.com/manuals/BUILD/PULP-OS/html/index.html*

[21] Himanshu Bhatnagar
Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler™ Physical Compiler™ and PrimeTime®

[22] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K. Gürkaynak, Luca Benini
PULPino: A small single-core RISC-V SoC

[23] Michael Wu, Yang Sun, and Joseph R. Cavallaro
Electrical and Computer Engineering
Rice University, Houston, Texas 77005
Implementation of a 3GPP LTE Turbo Decoder Accelerator on GPU

[24] Heungseop Ahn, Yong Jin, Sangwook Han and Seungwon Choi
Design and Implementation of GPU-based Turbo Decoder with a Minimal Latency

[25] *https://www.nvidia.it/object/geforce-gtx-titan-it.html#pdpContent=2*

[26] *https://en.wikipedia.org/wiki/RISC-V*