Exploration of RISC-V architectures and extensions for hardware-accelerated post-quantum cryptography schemes on FPGAs

MASTER IN INDUSTRIAL ELECTRONICS

Author:

Javier Míguez Silva

Project director:

Alfonso Rodríguez

POLITÉCNICA

INDUSTRIALES ETSII | UPM

CEI UPM | Centro de Electrónica Industrial

EXECUTIVE SUMMARY

The following master thesis project aims to integrate a hardware post-quantum cryptography accelerator inside an open source RISC-V extendable hardware platform, using Verilog and SystemVerilog *Hardware Description Languages* (HDLs). The base open source platform consists of a soft-core microcontroller with RISC-V architecture, developed by OpenHWGroup, which was designed to be extended with custom hardware of any kind and implemented in some specific commercial *Field Programmable Gate Arrays* (FPGAs).

This platform provides a set of hardware interfaces and implementation features that allow the user to explore different posibilities about how to integrate external hardware modules, taking into account up-and-coming RISC-V architecture characteristics and posibilities. Hence, the main contributions of this project will reside in developing, designing, simulating and implementing custom RISC-V architecture extensions with the main microcontroller, for the particular case of a hardware post-quantum cryptography accelerator. The whole integration will result in a full embedded system for FPGAs.

INDEX

TABLES INDEX

FIGURES INDEX

CODE INDEX

GLOSSARY AND ACRONYMS

ALU Arithmetic and Logic Unit. It is the part of a CPU where arithmetic and logic operations are performed (additions, subtractions...). .

Bare-metal In software development, a bare-metal application is a program implemented directy in the physical processor, handling instructions, threads and full physical address space without any particular operating system. .

CISC Complex Intruction Set Computer. Type of computer architecture which is characterized by having an instruction set with complex and long instructions that need to be processed in smaller stages to perform the required tasks. .

CLK Clock signal. In digital design, a clock is a square wave signal that synchronizes the operation of every component (if the design is synchronous). Clocks are the base of sequential logic. Depending on how sequential modules are designed, they are activated with a rising or falling edge of the clock signal. .

CPU Central Processing Unit. In a computer, the CPU is the main component that processes instructions. It is usually divided into an ALU, a Control Unit (CU) and a memory/storage unit (registers). .

CU Control Unit. In a CPU, the control unit is the one that handles instructions reception in three stages: fetching from memory, decoding and execution. It is always coordinated with the ALU and the rest of the components. .

DMA Direct Memory Access. It is a hardware method that allows an external device or peripheral to access the main memory without any CPU intervention. This improves memory access requests performance, as less time is required. .

DMI Debug Module Interface. It is the interface that the Debug Transport Module (DTM) uses to communicate with the debug module. The DMI captures requests made by the host through its correspondant DTM register, and provides requested data through the JTAG protocol. .

DTMCS Debug Transport Module Control and Status. In the context of JTAG protocol, the Debug Transport Module is a key component that connects external JTAG interface and internal debug interface. The DTMCS is an element of the Debug Transport Module that controls and monitors its status. .

ES Embedded System. Complete device that includes electrical components or electronic hardware inside, and is made for a specific purpose. It can be very complex as it can contain multiple microcontroller chips, perpherals, communication buses, networks... .

FPGA Field Programmable Gate Array. It is an integrated circuit made by an array of logic cells, which can be programmed and reconfigured after manufacturing. FPGAs allow to implement a wide variety of logic blocks and functions without the need of designing a particular integrated circuit for each purpose. .

FSM Finite State Machine. In digital design, a finite state machine is a sequential module which is built by a set of registers, representing a finte number of internal states at any given time, driving output ports depending on the input ports or the internal state. .

GPIO General Purpose Input/Output. Usually, GPIOs are external pins of a microcontroller that have no specific purpose. These pins can be driven internally to stablish communication with the user or with external hardware devices (to perform control or data transmission tasks, for example). .

GPU Graphic Processing Unit. It is a hardware component which is specially optimized to carry out matrix and graphic calculations inside a computer. .

HDL Hardware Description Language. Specialized computer language that can be used to describe structure, behavior and timing of digital electronic circuits. .

High-level In the context of electronics and computer engineering, a high-level perspective is considered when a specific design, analysis or procedure is carried out pursuing an easy human comprehension and considering human insight. .

HPC High Performance Computing. Advanced computing approach which is able to handle large amounts of data and to perform very complex sets of calculations. HPC computers use higher speeds than personal computers, with very powerful processors, exploiting parallelism. .

HW Hardware. Physical components of an electronic system, including circuitry, boards, components, ICs... .

IC Integrated Circuit (chip or microchip). Set of electronic circuits placed on a small flat piece of semicondictor material, usually silicon. It contains a large amount of transistors and other electronic components. .

IP Intellectual Property. Complete hardware design block or integrated circuit layout design that provides a specific functionality. Usually, it is licensed and can only be used under certain conditions. .

ISA Instruction Set Architecture. Abstract layer of an electronic system that defines how central processing unit is controlled by the software. It defines which intructions will the processor be able to execute, data types, registers, memory... .

JTAG Joint Test Action Group. It is a verification and testing protocol for printed circuit boards. It implements dedicated debug ports with a serial communication interface managed by dedicated registers called Test Acess Ports (TAP). This protocol is intended to monitor the printed circuit board functionality and find potential faults. .

Kernel A kernel is the core that sustains an operating system. Its objective is to manage operations between software and hardware from a low-level perspecive. It controls the operation of each hardware component or resource. .

Low-level In the context of electronics and computer engineering, a low-level perspectve is defined when a specific design, analysis or procedure is tackled rawly, using a language that is close to hardware understanding (transistor switching, ‘1’ or ‘0’) and far from human insight. .

LUT Look-Up Table. They are the basic digital cells inside FPGAs. Look-Up Tables are the simplest logic unit capable of implementing logic functions. .

MCU Microcontroller Unit. It is a hardware module inside an integrated circuit that includes a CPU with memory, communication and peripheral interfaces. .

MOS-CMOS (Complementary) Metal Oxide Semiconductor. Most important structure technology for transistors and ICs manufacturing. It is a three-layered structure where a metallic oxide such as silicon dioxide serves as an insulating layer, with the presence of a metal and a semiconductor substrate. .

OS Operating System. Set of computer programs (software) that control the behavior of the hardware providing an user interface (with user privileges) from a high-level perspective. It usually implements a kernel that handles each hardware resource. .

RISC Reduced Instruction Set Computer. Type of computer architecture for microcontrollers which is characterized by having an instruction set with easy and short instructions that can be chained to perform more complex tasks. .

RTL Register Transfer Level. Design abstraction which models a synchronous digital circuit representing the flow of digital signals or data through hardware registers, performing logical operations on these signals. .

SoC System on Chip. Integrated circuit that cointains most of the components of a computer, such as a CPU, memory interfaces, input/output devices, storage or additional components like a GPU. .

Soft-Core Relative to a HW platform, module or IP which is capable of being implemented completely using logic synthesis, which consists of translating a particular computer code into a digital electronic circuit. .

SPI Serial Peripheral Interface. It is a synchronous interface for short-distance wired communication between integrated circuits (ICs). It supports full-duplex communication, permitting data transmition and reception simultaneously. .

SW Software. Set of non-tangible instructions and programs that an electronic system must process and execute in order to provide its required functionality. .

Sync/Async Synchronous or asynchronous. It references a digital circuit or communication protocol depending on the presence or absence of a clock signal. .

UART Universal Asynchronous Receiver Transmitter. It is a simple hardware interface that uses two wires for asynchronous serial communication. .

VLSI Very Large Scale Integration. It is a technique that allows thousands, millions or billions of transistors to be integrated onto a single silicon semiconductor chip. .

1 INTRODUCTION

Computer architectures have been evolving by leaps and bounds since early 1960s [Raza, 2023]. By that time, HW was quite expensive and limited due to the recent emergence of transistors for electronic circuits, replacing old vacuum tubes that were predominant until those years. Computing was in a really primitive stage, and new technologies were still emerging. Programming languages that have an important relevance today were being created, while engineers were developing first commercial computers and programming them using assembly language. Assembly lenguage is a set of written computer instructions that programmers used to issue orders to electronic computers. Then, all these instructions were translated into machine code, which is a burst of binary numbers ('1' or '0') that computers interpret to develop a certain behavior. Binary codes are specially useful since modern transistors work in a saturation region. When they are placed in an interconneted layered lattice inside an IC [ASML, 2023], their behavior is like a switch. They can block or let current pass through internal circuits depending on how they are programmed, so binary codes are specially suitable for this purpose. *Fig.1* shows how two main types of modern transistors in computer engineering look like [Xuesong Zhang and Zhang, 2022].

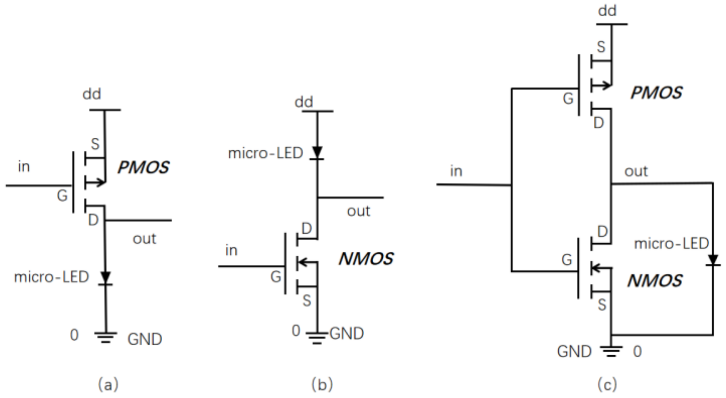

Figure 1.1: pMOS and nMOS transistors

From an architectural abstraction level, modern transistors operation is easy to understand. Basically, depending on their nature (pMOS or nMOS), they will let electrons pass from drain (D) to source (S) if they are nMOS or from source (S) to drain (D) if they are pMOS, depending on the threshold voltage received in the gate (G) terminal. These transistors are the base of logic gates and digital circuits of any kind. As an example, *Fig.1.1 (c)* shows an inverter circuit made by a pMOS and a nMOS connected in series. The micro-LED will turn on if the input is set to '0', and turn off otherwise. nMOS and pMOS transistors are combined, giving rise to CMOS structures that are the base technology for ICs manufacturing. Nowadays, these CMOS structures are grouped in silicon wafers using VLSI techniques, where a very big number of MOS transistors are placed in a small area.

Coming back to 1960s, when engineers were designing first computer architectures, they noticed that the best way to minimize HW resources usage (based on the limited number of transistors they could place in their circuits) was to make computer instructions that could carry complex tasks. This is where CISC architecture approach begins. As soon as big computer industry companies such as IBM or Intel started commercializing their processors and personal computers (*PCs*), CISC architectures became a standard in computer industry.

Nevertheless, towards the late 1970s and the early 1980s, researches at the University of California, Berkeley, proposed a new design philosophy with simpler and shorter instructions, decreasing

complexity. This new approach would provide an instruction set which is easier to implement in HW, and could get faster execution speeds while instruction decoding times are also decreased. By this time, RISC architectures start to rise until nowadays, when it is proven that they are very efficient in terms of performance and power consumption. RISC architectures are predominant in mobile and ES market. Mainly, they are focused on low-power devices market, but they are being more and more popular and effective also in HPC market, competing against CISC. A good example of a supercomputer with RISC architecture is the *Fugaku* in Japan, which has 158976 nodes and, as an example, it has a peak performance of 537 Petaflops in boost mode for double precision floating point [Fujitsu, 2024].

It is important to mention that although the rivality between CISC and RISC has been present for many years, aspects like performance or power consuption are more dependant on the microarchitecture optimizations rather than CISC or RISC classification [Hruska, 2021]. CISC vs RISC is not adequate for comparing modern architectures. In *Fig.1.2* and *Fig.1.3*, two graphs are shown, indicating main difference of performance and power consumption metrics of some commercial processors.

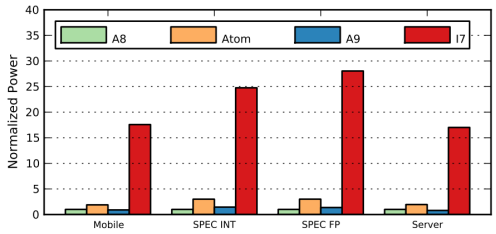

Figure 1.2: Raw average Power Consumption normalized to ARM-Cortex A8

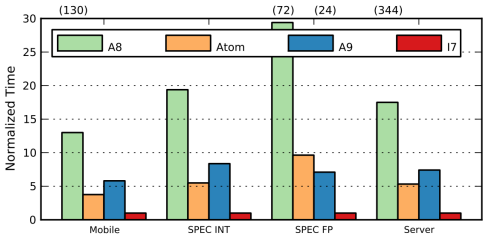

Figure 1.3: Execution Times normalized to Intel Sandy Bridge i7

Graphs above exhibit the response to certain computing benchmarks [Emily Blem and Sankaralingam, 2013], taking into account execution times and power consumption. Generally, RISC architectures will achieve less power consumption than CISC, with a reasonable throughput. However, microarchitecture design and optimization will be key points in this matter. Conclusions can not be extracted considering only ISA.

Nowadays, computers and technology have experienced a huge evolution in general, to the extent that they are key elements in our lifes. An increasing dependance on technology for finances, business, companies and governments has brought new threats for data privacy and communications security in particular. This is why cybersecurity is essential to provide protection against uninteded data leaks or cyberattacks trying to break systems and communications infraestructures, or even commit crimes.

This concern brings the necessity of exploring new possibilities on how to secure personal or confidential information, given the existing technology. Cryptography is one of the real-world solutions for cyphering and decyphering data for transportation through any communication medium securely, using mathematical problems and algorithms which are very difficult to solve by digital computers. Particularly, quantum computers are an outstanding example of how new computing capabilities are threatening conventional computers, solving in a reasonable amount of time those mathematical problems involved in conventional cryptography [Mosca, 2018].

Given the situation above, it is clear how a quantum-ready cryptography is needed to find a solution for the potential threats brought by quantum computers. Fortunately, some mathematical problems are not yet resolved by these quantum computers, and post-quantum cryptography is a key research field to provide effective security today.

As a consequence, this master thesis project will explore different possibilities on developing a full ES for FPGAs, trying to accelerate a particular quantum-resistant cryptography algorithm by using a source soft-core MCU platform with a custom accelerator, considering different implementation possibilities in order to provide a custom efficient solution to the threats brought by quantum computers despite performance limitations of FPGA implementations. FPGA target is chosen since they have multiple advantages regarding other technologies. These advantages will be discusssed in the next section.

Additionally, this master thesis project tries to explain why acceleration is necessary for this particular case, as FPGA-based soft-core MCU HW platforms usually have a very limited performance and, although the computation of many quantum-resistant algorithms is nor very resource-demanding, execution times could be reasonably high when considering low-end MCUs.

2 STATE OF THE ART

2.1 RISC-V Architecture

In May 2010, the Parallel Computing Laboratory (Par Lab) from University of California, Berkeley, developed an ISA following RISC philosophy, which was called RISC-V (*Fig.2.1*) [RISC-V-International, 2023a]. Many RISC-V processors were built using Chisel HW construction language, which was also created by Par Lab, pursuing new contributions to the field of Parallel Computing. The project was funded by Intel and Microsoft between 2008 and 2013, and since the beginning, it was thought to be developed with an Open Source approach, under the Berkeley Software Distribution (BSD) license.

Figure 2.1: RISC-V logo

RISC-V was not born as a new cutting-edge chip technology. Its purpose is to provide a global open standard to freely develop custom HW for SW applications, always under an open source perspective. In 2015, a non-profit organization called RISC-V Foundation was created with the objective of supporting an open and collaborative community for SW and HW innovators based on the RISC-V ISA. The organization became part of the Linux Foundation as soon as it was created, in order to receive technical, strategic and operational support for the ISA and make it compatible with Linux kernel.

RISC-V ISA has both an unprivileged [RISC-V-International, 2023d] and privileged [RISC-V-International, 2023c] specifications, includign instructions for user access or kernel access. Kernel instructions (privileged) have a major access to hardware resources such as memory mappings or I/O (input and output) devices, while user instructions have limited access. Specifications are both extense (found in RISC-V webpage), but focusing on RISC-V architecture main characteristics can be summarized in the following points.

- Suitable ISA for any particular microarchitecture or implementation technology, such as ASICs, FPGAs or full-custom.
- Separated into four base integer instructions sets that can be used freely, with aditional standard or custom extensions.
- Both 32-bit and 64-bit address space variants for applications, OS kernels, and hardware implementations.
- An ISA with support for highly-parallel multicore or manycore implementations, including heterogeneous multiprocessors [RISC-V-International, 2023d].

In the following tables (*Table 2.1 and Table 2.2*), base and standard extensions are presented (updated 12/21/2023) [RISC-V-International, 2023b]:

Base	Description
RV32E	Base 32-bit ISA with 16 registers
RV32I	Base 32-bit ISA
RV64I	Base 64-bit ISA
RV128I	Base 128-bit ISA

Table 2.1: RISC-V base extensions

Extension	Description
A	Atomic instructions
B	Bit manipulation
C	Compressed instructions
D	Double-precision floating point
F	Single-precision floating point
G	Shorthand for IMAFD extensions
H	Hypervisor instructions
J	Dynamically translated languages
L	Decimal floating point
M	integer multiplication and division
N	User-level interrupts
P	Packed-SIMD instructions
Q	Quad-precision floating point
S	Supervisor mode
T	Transactional mode
V	Vector operations

Table 2.2: RISC-V standard extensions

All these extensions are supported and included inside RISC-V ISA specification, but **custom extensions defined by users can also be implemented** in order to exploit hardware optimizations as much as possible. This is a strong advantage of RISC-V architecture and a key point of its potential.

2.2 Ongoing CPU projects and FPGA implementations

FPGAs have multiple advantages regarding other technologies. First, reconfiguration is the key feature since hardware microachitecture (RTL) can be modified easily to make necessary adaptations or add new features. Eventually, the development cost will be lower than for a full-custom silicon chip or ASIC. Although performance or achievable clock speeds might be higher in ASICs or full-custom implementations (SoC), FPGAs are the most cost-efficient solution providing acceptable results and reconfiguration possibilities that are essential for this project, because multiple design posibilities will be tested.

As this project will be focused on RISC-V SoC development and implementation for FPGAs using HDL, it is important to highlight the contribtions of some organizations to the field. The most relevant are OpenHWGroup, PULP (Parallel Ultra-Low Power) platform, ESL-EPFL (Embedded Systems Laboratory from École Polytechnique Fédérale de Lausanne, Switzerland) and OpenTitan.

These organizations are currently working on providing open source hardware platforms to the community, with different implementation possibilities apart from FPGAs, but this project will focus on the particular case of FPGAs. OpenHWGroup has developed a set of soft-core RISC-V CPUs (**CORE-V**) in RTL using HDL description (OpenHWGroup Projects - Official Website). These CPUs are soft-core because they can be implemented completely using logic synthesis, which is the process of transalating its described behavior in HDL into a real electronic circuit (logic gates).

The base model of soft embedded RISC-V CPU developed by PULP platform and mantained by OpenHWGroup is CV32E40P (also known as RI5CY) [OpenHWGroup, 2023a], which is a small and efficient core with a 4-stage pipeline that implements RV32IM[F,Zfinx]C ISA. This core was designed pursuing a tight power budget for ES that have intensive and short periodic workloads interleaved with long periods of inactivity, for applications such as *Internet of Things* (IoT). The objective is to maximize battery life and deploy the required performance only when necessary, which is a relevant challenge in ES research field nowadays [Pasquale Davide Schiavone et al., 2017] [Platform, 2023].

The architecture block diagram of this CPU is shown in *Fig.2.2*. As it is seen, all the expected units of a conventional CPU are included. Each unit has its own RTL file in SystemVerilog HDL.

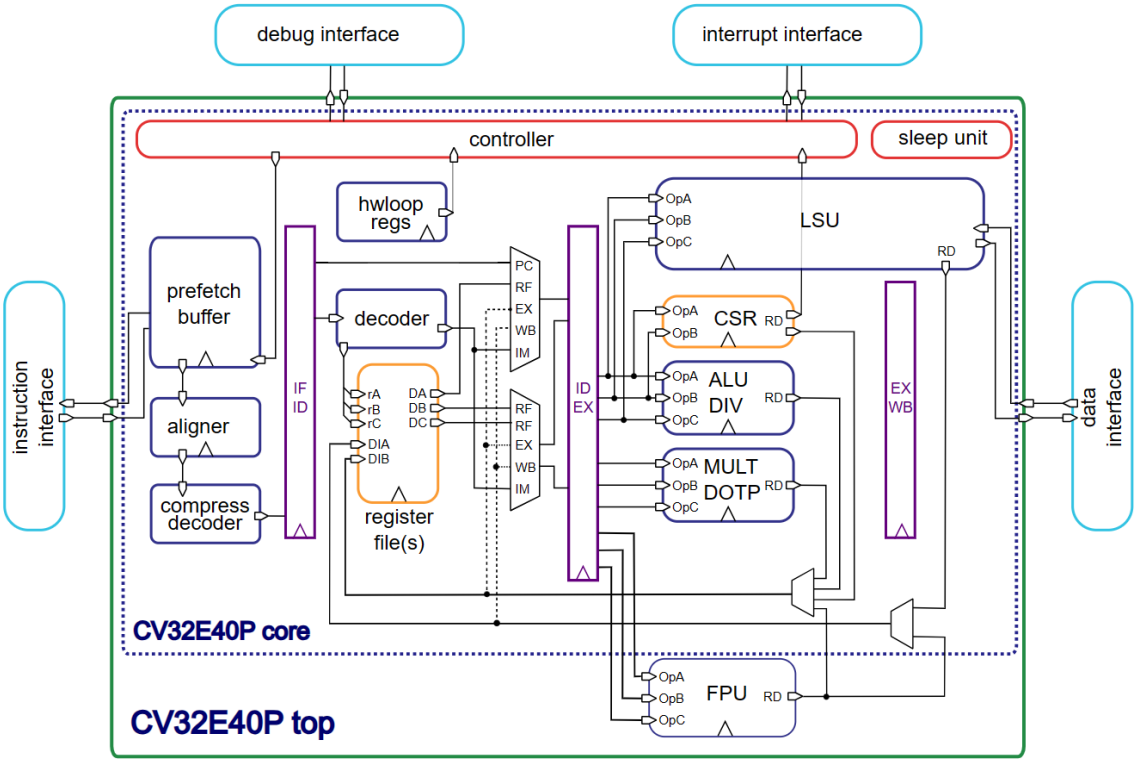

Figure 2.2: CV32E40P Block diagram

Apart from this base model, there are other CPUs (also developed by OpenHWGroup) with other purposes, such as CV32E40X. This CPU is intended for compute-intensive applications [OpenHWGroup, 2023b], and it implements a new extension interface (X-IF) which will be described later in detail. Both CV32E40P and CV32E40X are used in this project, adding external HW that will stablish communication with these CPUs in multiple ways.

2.3 Post-quantum cryptography

The two CPU models described above could be used to implement external HW-dedicated accelerators for post-quantum cryptography algorithms. This challenge araises given the context where quantum computers grow more and more through the years, and they represent a potential threat for conventional cryptography, communications security and data privacy.

However, a wide range of mathematical problems are still very difficult to solve (in a reasonable or polynomial amount of time), even for quantum computers, and they can be used as the base for post-quantum cryptography algorithms. The list below explain briefly some of these problems [Maximilina Richter, 2022]:

- **Lattice problems**: based on finding the shortest vectors in a lattice.
- **Learning With Errors**: given a system of linear equations with noise, finding a solution is compute-intensive.
- **Linear Codes**: code-based cryptography. The complexity is given when trying to decode a general linear code, which is a NP-hard problem.
- **Multivariate Equation Systems**: consists of solving multivariate polynomial equations.
- **Hash Functions**: given some properties of hash functions, this problem relies on finding an original input given the output of a hash function.

This master thesis project will focus on lattice based problems, and particularly, NTRU-based lattice cryptography, as this scheme provides a set of advantages listed below [Daniele Micciano, 2009], regarding the other quantum-resistant algorithms. Moreover, in the context of this project, these advantages become more significant for the development of a full ES based on a low-power soft-core MCU and a custom HW accelerator for FPGAs.

- **Strong Security**: considering worst-case hardness, lattice based cryptography and NTRU provide very strong security proofs.
- **Efficiency**: no excessive computational resources are required to compute the algorithm, so this is a key feature in the context of low-end ES and soft-core implementation for FPGAs.
- **Simplicity**: lattice-based cryptosystems are usually simpler than the other options. They are easier to understand, implement and analyze.
- **Quantum Resistance**: today, quantum computers are not ready for solving this cryptographic schemes in a reasonable amount of time, so they are a secure option.

Hence, NTRU and lattice-based cryptography is suitable for the implementation case commented in this section, and it is a very interesting application field for ES.

3 GROUNDWORK

3.1 X-HEEP

Given the OpenHWGroup CPU proposals (CV32E40P and CV32E40X), multiple possibilities for future embedded CPU or microcontroller projects for FPGAs are presented. This is possible because the RTL descriptions are open source, and they can be adapted, integrated or improved as desired.

Recently, researchers from ESL-EPFL decided to start the development of a whole embedded microcontroller unit (MCU) called X-HEEP [ESL-EPFL, 2023]. This really promising project aims to integrate both CV32E40P and CV32E40X CPUs with a system bus, memory subsystem, peripheral subsystem and debug subsystem, supporting board debug and communication protocols such as JTAG or SPI, and including a UART. This MCU project is also open source, and it is intended to be extended with external custom IPs, accelerators or coprocessors.

X-HEEP is also described using SystemVerilog HDL, and it uses a set of open source tools such as Verilator [Verilator, 2023] for simulation and FuseSoC [FuseSoC, 2023] for generating the core from RTL. The block diagram is included in *Fig.3.2.*

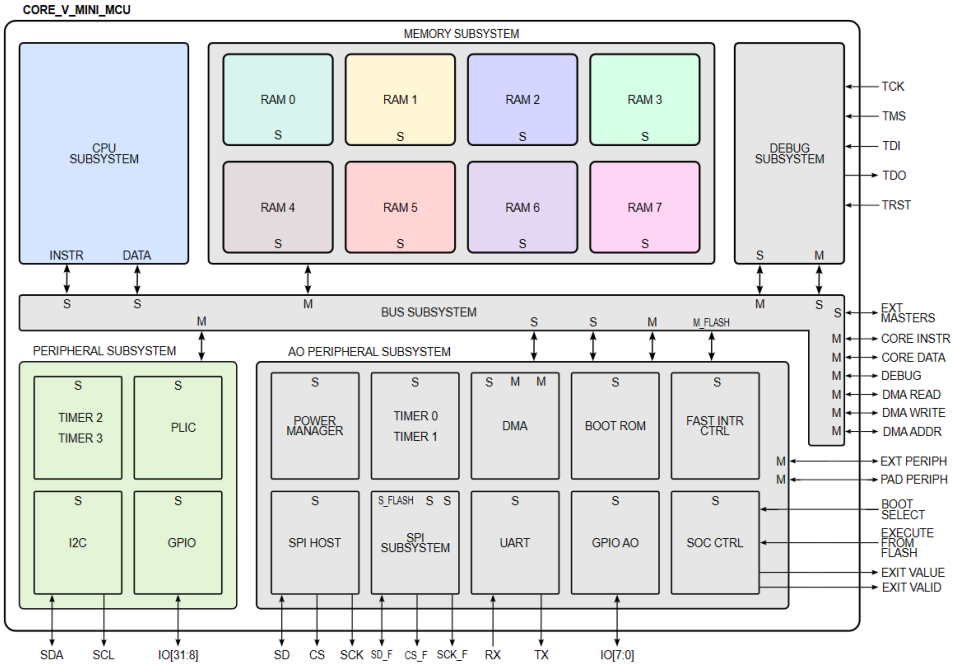

Figure 3.1: X-HEEP block diagram

The most interesting part of this MCU is the additional support for **CORE-V-XIF** (*eXtension Interface*) when CV32E40X CPU is specified and MCU is built. This interface will be described in detail in the next sections, but it is basically a group of sub-interfaces that permit RISC-V instruction set extension with custom or standarized instructions when attaching external HW, without changing the RTL of the CPU itself. This will be the key topic where the main contributions of this master thesis take place.

3.2 NTRUEncrypt

The wide range of extension possibilities offered by X-HEEP raises the challenge of developing and integrating a NTRU-based (N-th degree Truncated polynomial Ring Units) post-quantum cryptography accelerator with this hardware platform. Particularly, part of the encryption algorithm (which is called NTRUEncrypt) will be considered.

The implementation of NTRUEncrypt algorithm for this case is thought to operate with a set of polynomial coefficients, performing truncated multiplications, which is a part of the encryption process. Two cryptographic keys are needed: public and private (following an asymmetric scheme). When considering asymmetric cryptography schemes, the public key can be seen by anyone and it is used during the encryption, but the private key is only inteded to be taken by the receiver, which is the one capable of decrypting the message.

NTRU specifications have a set of parameters to define important characteristics of the algorithm, such as the polynomials length or the number of bits used to encode coefficients. These parameters are described in *Table 3.1.*

Parameter	**Description**
N	It represents the degree of the polynomial (prime number).
p	Small modulus. Number of possible values for the first input polynomial coefficients.
q	Large modulus. Number of possible values for the second set of coefficients.

Table 3.1: NTRUEncrypt parameters

The value of these parameters can be chosen among a set of numbers proposed in NTRUEncrypt specification [Cong Chen et al., 2019].

Given the following truncated polynomial ring R of degree N, where $Z_t[X]$ is a set of polynomials with integer coefficients reduced module t (as stated in [Santiago Sánchez-Solano and Brox, 2022]):

$$R_{N,t} = \frac{Z_t[X]}{X^N - 1}$$

Multiplications are carried out considering the following sets of truncated polynomials:

$$r(x) = r_0 + r_1 \cdot x + r_2 \cdot x^2 + \cdots + r_{N-1} \cdot x^{N-1}$$
$$h(x) = h_0 + h_1 \cdot x + h_2 \cdot x^2 + \cdots + h_{N-1} \cdot x^{N-1}$$

And then performing the operations as follows:

$$e_k = \sum_{i+j=k \mod N} (h_j \cdot r_i) \mod q$$

This truncated multiplication algorithm can be easily computed using any conventional programming language such as **C**, as shown in *Code 3.1*, *Code 3.2* and *Code 3.3*.

```
#ifndef POLY_RQ_MUL_H
#define POLY_RQ_MUL_H

#include <stddef.h>
#include <stdint.h>

#define NTRU_N 509

typedef struct{
  uint16_t coeffs[NTRU_N];
} poly;

void poly_Rq_mul(poly *r, const poly *a, const poly *b);

#endif
```

Code 3.1: NTRUEncrypt example C header file

```
#include "poly_rq_mul.h"

void poly_Rq_mul(poly *r, const poly *a, const poly *b)
{
  int k,i;

  for(k=0; k<NTRU_N; k++)
  {
    r->coeffs[k] = 0;
    for(i=1; i<NTRU_N-k; i++)
      r->coeffs[k] += a->coeffs[k+i] * b->coeffs[NTRU_N-i];
    for(i=0; i<k+1; i++)
      r->coeffs[k] += a->coeffs[k-i] * b->coeffs[i];
  }
}
```

Code 3.2: NTRUEncrypt example C function definition

On the one hand, in *Code 3.3*, consecutive multiplications and additions of the polynomial coefficients are carried out with a loop, depending on *NTRU_N*. For a general-purpose low-end CPU, execution times of the whole algorithm could be high, so this first approach will only be tested to compare the execution time results with the subsequent HW accelerated implementation.

```c
#include "poly_rq_mul.h"
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>

int main(){

poly x1, x2, x3, x3_ref;
poly *a = &x1, *b = &x2, *r = &x3, *r_ref = &x3_ref;
int i = 0, k = 0;

//Invoke function under test
poly_Rq_mul(r, a, b);

for(k=0; k<NTRU_N; k++)
  {
    r_ref->coeffs[k] = 0;
    for(i=1; i<NTRU_N-k; i++)
    r_ref->coeffs[k] += a->coeffs[k+i] * b->coeffs[NTRU_N-i];
    for(i=0; i<k+1; i++)
    r_ref->coeffs[k] += a->coeffs[k-i] * b->coeffs[i];
    printf("a: %u, b: %u, r: %u, r_ref: %u \n", a->coeffs[k], b->coeffs[k], r->
        ↪ coeffs[k], r_ref->coeffs[k]);

    if(r->coeffs[k] != r_ref->coeffs[k]){
      return -1;
    }
  }

return 0;
}
```

Code 3.3: NTRUEncrypt example C test program

On the other hand, it can be seen how the multiplication stage of NTRUEncrypt can be programmed with relative ease from a user perspective, remarking the discussion in the previous section, as involved operations are understandable and analyzable.

4 POLYMUL IP

The accelerator designed for NTRU algorithm is called **POLYMUL**. Its architecture came from an article made by **IMSE** (*Instituto de Microelectrónica de Sevilla, Spain*), where architectural proposals are presented for NTRU-based polynomial multipliers [Santiago Sánchez-Solano and Brox, 2022]. This article also includes implementation aspects such as resources usage for Xilinx FPGA devices, and expected performance evaluation. Base design has been replicated for this purpose, using *SystemVerilog* HDL, with the objective of exploring different implementations later.

In *Fig.4.1*, the architecture block diagram is presented. It consists of a set of three BRAM memories, called **MEM R, MEM H and MEM E** with a **Control Unit module** and an **AU** (*Arithmetic Unit*). The idea for this blocks is to be instantiated multiple times in the design, in order to achieve better acceleration results. Nevertheless, the more instantiated blocks, a higher resource utilization is presented (in LUTs).

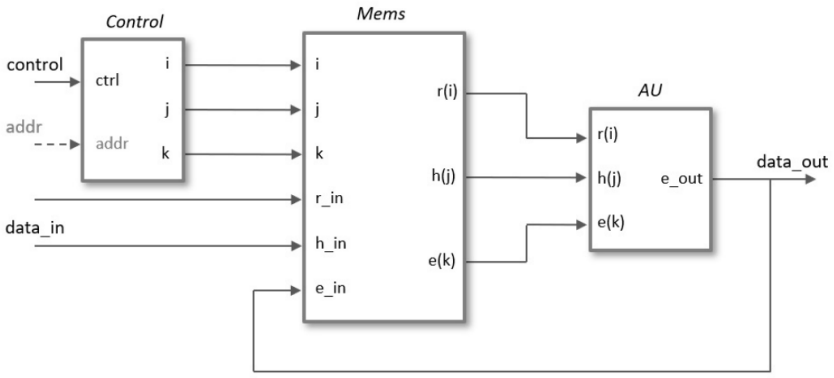

Figure 4.1: POLYMUL block diagram

For the AU, there is an architecture proposal too. It is a simple arithmetic unit because it is only expected to perform recurrent additions and subtractions. In *Fig.4.2* and *Fig.4.3*, the block diagrams for a single unit and with multiple instances are attached.

For this implementation, the NTRU standard parameters selection is **EES401EP1**, the simplest one. This standard is selected because the amount and length of the coefficients are not as relevant as the subsequent implementation of an ES for FPGAs, and if any other standard is wanted to be selected, it can be changed freely because the accelerator is designed with correspondant generics and parameters for choosing the desired configuration. Hence, EES401EP1 values are collected in *Table.4.1*. **N** parameter will represent the maximum length of **MEM R, MEM H and MEM E**, while **p** and **q** indicate the number of possible values the two sets of input coefficients could take (p=3 means 2 bits for encoding the first set of input coefficients, and q=2048 means 11 bits for encoding the second set).

Parameter	Value
N	401
p	3
q	2048

Table 4.1: NTRUEncrypt parameters

In *Code 4.1* the RTL description in SystemVerilog of the arithmetic unit (AU) is listed, as an

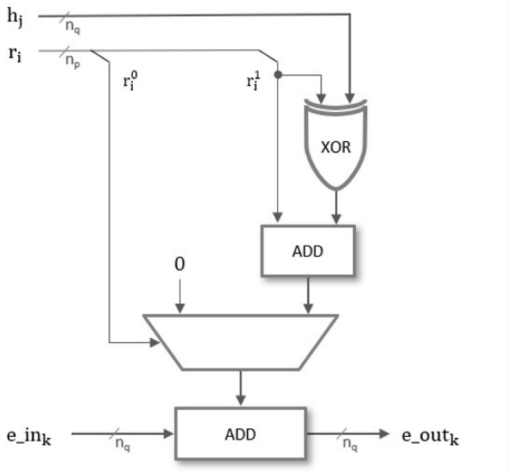

Figure 4.2: POLYMUL arithmetic unit block diagram

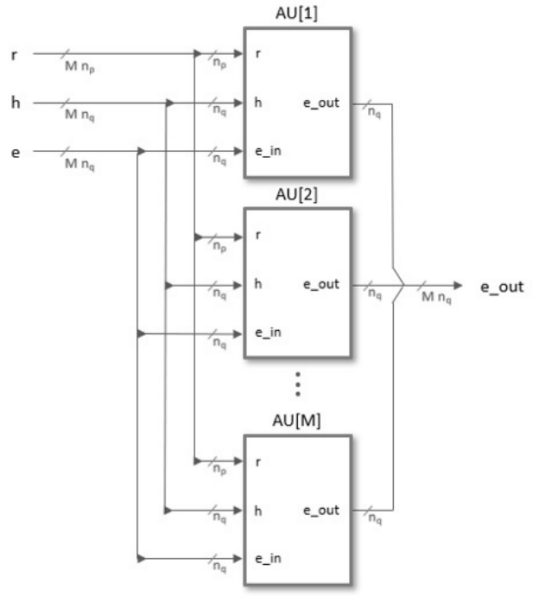

Figure 4.3: POLYMUL arithmetic units in parallel

example. Regarding the control unit, its purpose is to generate a set of indexes (**i, j, k**) for addressing **MEM R, MEM H and MEM E** respectively once all the input coefficients are loaded in **MEM R and MEM H**. This situation is indicated with **ctrl** signal (*ctrl=1*). These indexes, when addressing memories, redirect the input coefficients from **MEM R and MEM H** to the single or multiple arithmetic units, and results are stored in **MEM E**. Once all the results are loaded in **MEM E**, the accelerator will indicate that the operation phase has terminated, signaling *endop=1*.

An interesting feature of this IP, as mentioned before, is that it can be parallelized and customized as desired. When instantiated inside any design, the top-level entity has a set of parameters that can be freely configured. For example, there is a parameter called **M** which represents the number of arithmetic units placed in parallel to operate at the same time. Consequently, memory instantiations will be adapted automatically, respecting NTRU algorithm. Furthermore, if any other NTRU standard different from the one mentioned above is implemented, it is only necessary to adapt the parameters in the top-level entity accordingly, as mentioned before.

```
module ALU
    #(  parameter C_DATA_WIDTH = 32,
        parameter C_DATA_WIDTH_R = 2,
        parameter C_ADDR_WIDTH = 9,
        parameter C_MEM_DEPTH = 401
        )

    (
    input wire en, end_op, clk,
    input wire [C_DATA_WIDTH-1:0] hj, data_in,
    input wire [C_DATA_WIDTH_R-1:0] ri,
    output wire [C_DATA_WIDTH-1:0] data_out
    );

    wire [C_DATA_WIDTH-1:0] signal_OUT_MUX;
    reg signal_end_op;

    always @ (posedge clk)
    begin
        signal_end_op <= end_op;
    end

    assign data_out = (en && ~signal_end_op) ? signal_OUT_MUX + data_in
        ↪ : data_in;
    assign signal_OUT_MUX = (en && ~ri[1] && ri[0]) ? hj :
                            (en && ri[1])           ? ~hj + 1 : 0;
endmodule
```

Code 4.1: POLYMUL ALU description in Verilog

5 POLYMUL Implementation

5.1 Peripheral Integration

For the POLYMUL IP integration with X-HEEP, three different possibilities will be considered, regarding X-HEEP available resources for external IP attachments. The first possibility is to make an integration as a simple peripheral. However, it is not adequate for this master thesis project, as it is very simple and the communication through peripheral registers will not achieve the expected performance. Furthermore, during SW execution, multiple data cycles will be required to write and read from memory, so it is not an optimum implementation. Nevertheless, peripheral integration will be described roughly because it shares common points with the other two implementation possibilities, especially register-mapping.

In X-HEEP GitHub repository documentation [ESL-EPFL, 2023], there is a guide about how to integrate a simple peripheral using the dedicated REG bus from X-HEEP. This is a specific communication bus for peripherals that X-HEEP implements in its design, and it sets handshake rules between the following signals, listed in the Verilog package below (from X-HEEP repository, *Code 5.1*).

```
package reg_pkg;

  typedef struct packed {
    logic        valid;
    logic        write;
    logic [3:0]  wstrb; //4 bits to enable each byte from data lines
    logic [31:0] addr; //32-bit words for address and data lines
    logic [31:0] wdata;
  } reg_req_t;

  typedef struct packed {
    logic        error;
    logic        ready;
    logic [31:0] rdata;
  } reg_rsp_t;

endpackage
```

Code 5.1: REG bus package file

The *reg-req-t* record is a master interface that X-HEEP uses to request register-mapped data from its attached modules (and also for external devices), and *reg-rsp-t* is a slave interface. Master interface includes *addr, wdata, wstrb* lines to specify register address and data to be written, respectively, signaling which portions of *wdata* should be considered in a byte-wise manner with *wstrb*. If a register write operation is wanted to be performed, *addr, wdata, wstrb* lines should be written first, and then, *write, valid* signals must be set to '1', signaling that the data in *addr, wdata, wstrb* is valid. Otherwise, if a register read is commanded, the procedure is the same, but the *write* signal must be kept to '0'. Then, the register-mapped slave will ignore *wdata*, and will provide the data from the requested address through *rdata*, setting *ready* to '1' when the bus is released, and '0' otherwise. The *error* signal is used to point that an error occurred in the last transaction.

To integrate POLYMUL IP with this bus, a specific file and directory tree must be created,

following the next structure and copying resulting folder in `/hw/ip` of X-HEEP directory tree.

```
data
└── polymul.hjson
rtl
├── polymul-reg-pkg.sv
├── polymul-reg-top.sv
├── polymul-reg-window.sv
└── polymul.sv
polymul.core
polymul.sh
polymul.vlt
```

Multiple file types are included in this structure. Some of the files such as `polymul-reg-pkg.sv` and `polymul-reg-top.sv` are automatically generated using OpenTitan `reggen` tool [OpenTitan, 2023], which is a Python script that takes a `.hjson` source file specifying which registers are expected to be generated, and then, it outputs the SystemVerilog descriptions to implement those specified registers. Inside `data` folder, `polymul.hjson` is included with the list of registers to which X-HEEP will get access. File is listed in *Code 5.3*.

Then, there is a `polymul.core` file, which is listed in *Code 5.2*. This is a file processed by `FuseSoC` to group all the Verilog modules (or other source files) that are going to be compiled and to set dependencies between file groups. These `.core` files are essential to chain multiple compilation queues between source modules. Aditionally, ther is a `polymul.vlt` file too, which is a Verilator-specific file capable of disabling certain warnings that the simulator outputs when unused signals appear in the design, or width mismatches take place when connecting logic arrays, for example. These warnings appear when `FuseSoC` compiles the model for Verilator sim target, and they will interrupt compilation process until the required Verilator rules are specified in `.vlt` files.

```
CAPI=2:
name: "x-heep:ip:polymul"
description: "core-v-mini-mcu pdm to pcm converter peripheral"
filesets:
  files_rtl:
    depend:
      - pulp-platform.org::common_cells
    files:
      - rtl/ALU.sv
      - rtl/Control.sv
      - rtl/MEM_R.sv
      - rtl/MEM_H.sv
      - rtl/MEM_E.sv
      - rtl/REG_BRAM_CTRL.sv
      - rtl/REG_POLY_MUL.sv
      - rtl/Poly_Mul.sv
      - rtl/polymul.sv
      - rtl/polymul_reg_pkg.sv
      - rtl/polymul_reg_top.sv
    file_type: systemVerilogSource
targets:
  default:
    filesets:
    - files_rtl
```

Code 5.2: polymul.core file listing

```
{ name: "polymul"
  clock_primary: "clk_i"
  bus_interfaces: [
    { protocol: "reg_iface", direction: "device" }
  ]
  regwidth: "32"
  registers: [

    // CTRL pin

    { name:     "CTRL"
      desc:     "Control pin"
      swaccess: "rw"
      hwaccess: "hro"
      fields: [
        { bits: "31:0", name: "CTRL", desc: "Start IP operation." }
      ]
    }

    // DATA_IN bus

    { name:     "DATA_IN"
      desc:     "Data input bus for loading coeffs in memory"
      swaccess: "rw"
      hwaccess: "hro"
      fields: [
        { bits: "10:0", name: "DATA_IN", desc: "Data input bus" }
      ]
    }

    // Window implementation

    { window: {
      name: "POLYMUL_RXDATA",
      items: "1",
      validbits: "11",
      desc: "POLYMUL BRAM read results window",
      swaccess: "ro"
      }
    }
    ...
  ]
}
```

Code 5.3: polymul.hjson file listing

Inside the `.hjson` file, `windows` can be declared. These `windows` are open regions of the registers map that are not specially intended for implementing registers, as they allow specific ranges inside the registers map thant can be used for other purposes. For example, a `window` could be implemented in POLYMUL registers map in order to access the read data buffer where operation results will be dumped. Then, X-HEEP will be able to collect results from there performing read operations.

To execute `reggen` Python script, a shell file must be created (`polymul.sh`). The content of this file is listed in *Code 5.4*.

```
cd hw/ip/polymul

./../../vendor/pulp_platform_register_interface/vendor\
/lowrisc_opentitan/util/regtool.py -r -t rtl data/polymul.hjson

./../../vendor/pulp_platform_register_interface/vendor\
/lowrisc_opentitan/util/regtool.py --cdefines \
-o ../../../sw/device/lib/drivers/polymul/polymul_regs.h \
data/polymul.hjson
```

Code 5.4: polymul.sh file listing

As it is seen, the Python script is located inside `hw/vendor` folder of X-HEEP source directory. It is executed and register files in SystemVerilog are generated in `hw/ip/polymul/rtl`.

The next step is to instantiate POLYMUL IP inside the peripheral subsystem of X-HEEP in `hw/core-v-mini-mcu/peripheral-subsystem.sv` (*Code 5.5*). Aditionally, it will also be necessary to adapt `core-v-mini-mcu.sv` module to include external signals or interrupts from the IP, as well as required parameters. Moreover, `mcu-cfg.hjson` in base directory must be modified as well, indicating the new address space for the peripheral inside X-HEEP (*Code 5.6*). In general, all these steps are described inside X-HEEP documentation in GitHub repository (`docs/source/How-to/IntegratePeripheral.md`) [ESL-EPFL, 2023].

```
polymul #(
    .M   (M),
    .C_DATA_WIDTH  (C_DATA_WIDTH),
    .C_DATA_WIDTH_R  (C_DATA_WIDTH_R),
    .C_ADDR_WIDTH  (C_ADDR_WIDTH),
    .C_MEM_DEPTH  (C_MEM_DEPTH),
    .MEM_DATA_BUS_WIDTH (MEM_DATA_BUS_WIDTH),
    .MEMORY_SELECT_POS (MEMORY_SELECT_POS),
    .reg_req_t(reg_pkg::reg_req_t),
    .reg_rsp_t(reg_pkg::reg_rsp_t)
) polymul_i (
    .clk_i,
    .rst_ni,
    .intr_endop(intr_endop),
    .rx_ready  (polymul_rx_ready),
    .reg_req_i (peripheral_slv_req[core_v_mini_mcu_pkg::POLYMUL_IDX]),
    .reg_rsp_o (peripheral_slv_rsp[core_v_mini_mcu_pkg::POLYMUL_IDX])
);
```

Code 5.5: POLYMUL IP instantiation in peripheral subsystem using REG bus

```
polymul: {
            offset:  0x00080000,
            length:  0x00010000,
            is_included: "yes",
            path:    "./hw/ip/polymul/data/polymul.hjson"
        },
        ...
```

Code 5.6: Added polymul module instance with its registers space inside peripheral-subsystem entry of mcu-cfg.hjson

Once this is done, a SW test program in `C` can be written to check if the HW platform is working as expected. In this case, included libraries from X-HEEP (`mmio.h, core-v-mini-mcu.h`) and the ones generated by `reggen` (`polymul-regs.h`) are used to perform requests through the REG bus. A pointer must be declared to write and read from the register, as follows (*Code 5.7, Code 5.8*).

```
mmio_region_t polymul_base_addr = mmio_region_from_addr((uintptr_t)
    ↪ POLYMUL_START_ADDRESS);
...
```

Code 5.7: Declaration of polymul REG pointer in main.c test file

```
...
//MEM H
memory_pos = 0x200;
mmio_region_write32(polymul_base_addr, POLYMUL_ADDR_IN_REG_OFFSET , memory_pos);
mmio_region_write32(polymul_base_addr, POLYMUL_DATA_IN_REG_OFFSET, 0x000003FF);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0xFFFFFFFF);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0x00000000);
memory_pos += 1;

mmio_region_write32(polymul_base_addr, POLYMUL_ADDR_IN_REG_OFFSET , memory_pos);
mmio_region_write32(polymul_base_addr, POLYMUL_DATA_IN_REG_OFFSET, 0x00000005);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0xFFFFFFFF);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0x00000000);
memory_pos += 1;

mmio_region_write32(polymul_base_addr, POLYMUL_ADDR_IN_REG_OFFSET , memory_pos);
mmio_region_write32(polymul_base_addr, POLYMUL_DATA_IN_REG_OFFSET, 0x000003CF);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0xFFFFFFFF);
mmio_region_write32(polymul_base_addr, POLYMUL_WSTRB_REG_OFFSET, 0x00000000);
memory_pos += 1;
...
```

Code 5.8: Write requests to POLYMUL MEM-H through dedicated registers in main.c test file

As said before, this implementation is not relevant for this master thesis project because it is not optimum, but registers implementation aspects are essential to be understood. For the discussion in the next subsection, register mapping process will be repeated for POLYMUL control signals.

5.2 Memory Mapping (OBI Interface)

X-HEEP has another integrated bus called `OBI` (*Open Bus Interface*), which has a written specification describing all the signals and characteristics [OpenHWGroup, 2023c]. OBI protocol is very similar to REG bus, and it is intended for memory mapping inside X-HEEP (*Code 5.9*). In this case, X-HEEP has dedicated external master and slave OBI interfaces to attach external IPs, and these interfaces can be used directly to communicate with the MCU through the system bus. POLYMUL IP will be integrated as a slave, using this OBI bus first and handling its correspondant signals.

For this purpose, it will be necessary to develop an OBI controller module in SystemVerilog, which is going to translate OBI signals into POLYMUL BRAM signals. This module is the one

that receives or provides data through the bus and writes into POLYMUL BRAM memories if required. The bus handshake process as a slave is listed in *Code 5.10*.

```
package obi_pkg;

  typedef struct packed {
    logic        req;
    logic        we;
    logic [3:0]  be;
    logic [31:0] addr;
    logic [31:0] wdata;
  } obi_req_t;

  typedef struct packed {
    logic        gnt;
    logic        rvalid;
    logic [31:0] rdata;
  } obi_resp_t;

endpackage
```

Code 5.9: OBI protocol interface signals

```
...
// OBI slave handshake
always_ff @ (posedge clk or posedge rst)
begin
    if (rst) begin
        gnt <= 0;
        rvalid <= 0;
        rdata <= 0;
    end
    else begin
        gnt <= 1;
        rvalid <= 0;
        if (master_req_i.req) begin
            rvalid <= 1;
        end
        if (mem_sel[1]) begin
                gnt <= 0;
            rdata <= bram_data_out_e;
        end
    end
end
...
```

Code 5.10: OBI protocol simple slave handshake

The protocol handling in this case is simple. The `gnt` signal is asserted to indicate that the slave is available to receive a request through the bus. When a request is received from the master (`req=1`), `rvalid` is asserted in the next clock cycle to indicate that the request has been processed successfully. In *Fig 6.1*, the waveform of a burst of write transactions can be visualized (from X-HEEP main memory to POLYMUL BRAMs, MEM-R or MEM-H).

Then, in *Figure 6.2*, it is visualized a burst of read transfers, from POLYMUL `MEM-E` (where

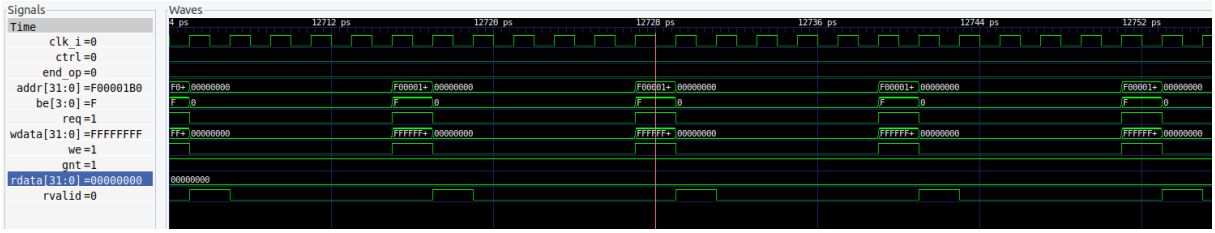

Figure 5.1: OBI write transfers burst

operation results are stored) to X-HEEP main memory. As in the previous case `gnt` signal was set to '1' during all the write transactions, now it is de-asserted when `rvalid=1` during reads. This is done to indicate the master that there is one clock cycle where the slave needs to hold the bus to provide the read result.

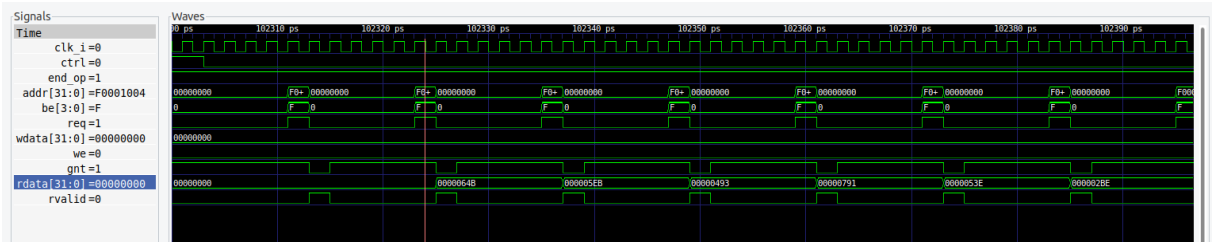

Figure 5.2: OBI read transfers burst

This is an explanation of how OBI protocol works, but now the whole implementation process of POLYMUL IP with X-HEEP through memory map will be explained. To do so, a new folder structure must be createted, but it will be different from the one created with the peripheral integration, as X-HEEP external devices integration relies on a procedure called `vendorizing` (as stated in `docs/source/How-to/eXtendingHEEP.md`) [ESL-EPFL, 2023].

Vendorizing means placing all the source files of a project (X-HEEP in this case) inside a folder called `vendor`, and then developing all the external HW/SW interfaces or modules (as well as other required filetypes) in the folders above. Regarding the context of X-HEEP, there is another Python script named `vendor.py` which takes a `.hjson` input file (declaring the repository link, version or specific commit and other relevant aspects) and performs vendorizing automatically. For the POLYMUL IP integration, it is necesarry to vendorize X-HEEP as indicated in `docs/source/How-to/eXtendingHEEP.md`. Afterwards, the following folder structure is created.

```
hw
├── polymul
├── TOP
└── vendor
    └── esl-epfl-x-heep
scripts
├── sim
└── synthesis
sw
├── applications
├── external
├── device --> (hw/vendor/esl-epfl-x-heep/sw/device)
└── linker --> (hw/vendor/esl-epfl-x-heep/sw/linker)
tb
util
```

```
├── auto-init-setup.sh
├── Makefile
├── polymul-x-heep.core
└── README.md
```

Symbolic links are indicated with (-->), and `ln -s` Linux bash command is used to create them. There is an `auto-init-setup.sh` shell script which is used to launch `Anaconda` environment (*core-v-mini-mcu*), define environment variables (with exports) and add the required binaries to the Linux `$PATH`. The file is listed in *Code 5.11*.

```
#!/bin/bash
conda activate core-v-mini-mcu
export RISCV=/home/$USER/tools/riscv
export VERILATOR_VERSION=4.210
export PATH=/home/$USER/tools/verilator/$VERILATOR_VERSION/bin:$PATH
export VERIBLE_VERSION=v0.0-1824-ga3b5bedf
export PATH=/tools/verible/$VERIBLE_VERSION/bin:$PATH
export PATH=/home/$USER/tools/openocd/bin:$PATH
export JTAG_VPI_PORT=4567
#Execute in terminal: . ./auto_init_setup.sh
```

Code 5.11: auto-init-setup.sh file listing

The `polymul-x-heep.core` file is a large file that contains all the necessary dpendencies from X-HEEP and the new implemented HW modules, for both simulation (using `Verilator`) and synthesis/implementation with `Vivado`. This file is essential to make `FuseSoC` work properly. The structure is similar to the `polymul.core` file listed in *Code 5.2*. Then, the Makefile is customized including rules for this particular implementation project after vendorizing, linking to X-HEEP main makefile as well. Now, it is time to describe the memory mapping HW implementation approach through OBI bus. The following files are included inside `hw/TOP` folder.

```
hw
└── TOP
    ├── ext-bus.sv --> (../hw/vendor/esl-epfl-x-heep/tb/ext-bus.sv)
    ├── ext-xbar.sv --> (../hw/vendor/esl-epfl-x-heep/tb/ext-xbar.sv)
    ├── polymul-x-heep-pkg.sv
    ├── polymul-x-heep-top.sv
    └── polymul-x-heep-top.vlt
```

Inside the `polymul-x-heep-pkg.sv`, POLYMUL IP is connected to X-HEEP through the external bus (`ext-bus.sv`), which is an interconnection module that gathers multiple OBI interfaces (for DMA address, DMA read, DMA write, debug master, core data and core instructions) and translates each of them to the correspondant X-HEEP OBI interface for external slaves. Then, `polymul-x-heep-pkg.sv` includes a set of parameters to set up memory mapping and define which address regions will be available for accessing POLYMUL. With this file, POLYMUL BRAMs will be accessible from X-HEEP through an address range of its whole address space definition. When POLYMUL address range is pointed, the system bus will redirect internal OBI bus through the external bus, and it will communicate directly with POLYMUL BRAMs, allowing write and read requests. The block diagram is shown in *Figure 5.3*.

When POLYMUL IP is attached with X-HEEP MCU and memory map is already configured, external interrupts by HW can be added in order to warn the MCU that POLYMUL has finished operating and has available results, once the corresponding POLYMUL control signals

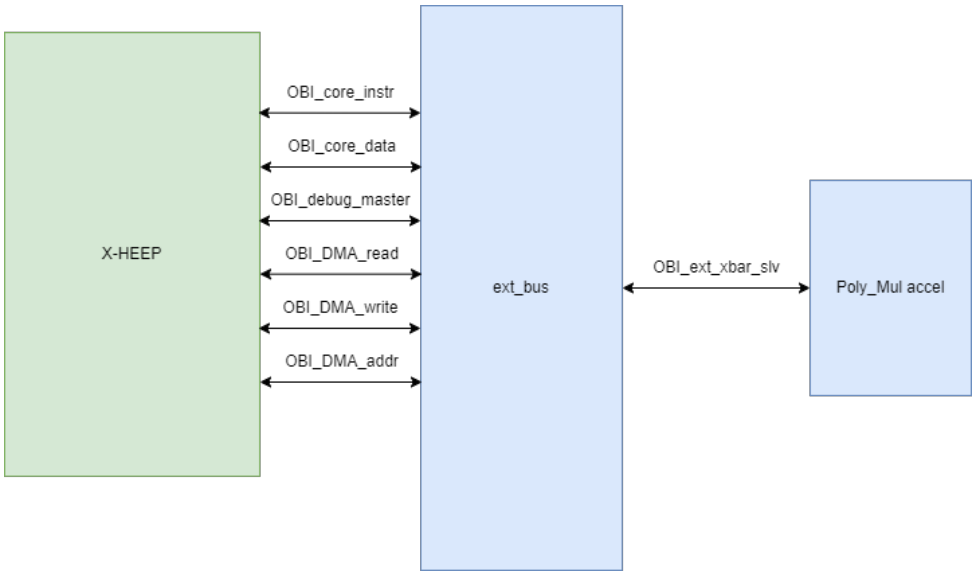

Figure 5.3: POLYMUL and X-HEEP through OBI memory mapping block diagram

are activated through the REG bus (`ctrl=1`). HW interrupts are a method that CPUs use to halt the main SW execution flow and provide service to a HW module that needs specific tasks to be done by that time. This is done through a SW function which is called *Interrupt Service Routine (ISR)*. In this case, HW interrupts will be enabled for POLYMUL, notifying the MCU when `ctrl=1` and operation results are available in `MEM-E` (signaling `end-op=1`). Consequently, when `end-op=1`, a HW intterrupt will be raised and the MCU will have to serve the interrupt request, executing a particular ISR.

In `polymul-x-heep-top.sv`, `end-op` signal can be connected to an external interrupt vector that X-HEEP owns to handle interrupts from external devices. Once the corresponding interrupt is attached, the MCU will receive the interrupt when requested (*Code 5.12*).

```
always_comb begin
  // All interrupt lines set to zero by default
  for (int i = 0; i < core_v_mini_mcu_pkg::NEXT_INT; i++) begin
    ext_intr_vector[i] = 1'b0;
  end
  // Re-assign the interrupt lines used here
  ext_intr_vector[0] = polymul_int;
end

...

x_heep_system #(
      .COREV_PULP(COREV_PULP),
      .FPU(FPU),
      .ZFINX(ZFINX),
      .EXT_XBAR_NMASTER(POLYMUL_XBAR_NMASTER)
  ) x_heep_system_i (
      .clk_i,
      .rst_ni,
      .jtag_tck_i,

...

      .intr_vector_ext_i(ext_intr_vector)

...
```

Code 5.12: External interrupt vector signal attachment in polymul-x-heep-top.sv

In X-HEEP, the module that is capable of handling interruptions is the `rv-plic` (*RISC-V Platform-Level Interrupt Controller*). It is a module which is part of RISC-V specification, and RTL can be found in *hw/vendor/esl-epfl-xheep/hw/vendor/lowrisc-opentitan/hw/ip/rvplic*. The chosen implementation is the one created by OpenTitan. Furthermore, the rv-plic implements a SW driver that can be found in *sw/device/lib/drivers/rv-plic*. These source files are required to be checked in order to understant how rv-plic works.

At SW level, POLYMUL memory map implementation needs a driver to make HW dedicated address regions accessible via SW. In `sw/external/extensions/polymul-x-heep.h`, POLYMUL size and address range is declared.

Then, in `sw/external/drivers/polymul`, registers for control signals are defined at SW level (generated by `reggen`), and *polymul* data type is defined as a memory region with its base address in `polymul.h`).

Finally, the main SW application program is going to be described. It can be found in `sw/applications/polymul-test` path. At the beginning, the `ISR` (*Interrupt Service Routine*) is defined, and when the main starts, interruptions configuration and activation functions are called (*Code 5.13*). In addition, the CSR (*Control and Status registers*) are written to allow interrupt handling. The `CSR` registers of X-HEEP are meant to enable or disable certain HW features of the MCU, and check internal status.

```c
#include <stdio.h>
#include <stdlib.h>

#include "core_v_mini_mcu.h"
#include "polymul.h"
#include "polymul_regs.h"
#include "polymul_x_heep.h"
#include "mmio.h"
#include "hart.h"
#include "rv_plic.h"
#include "csr.h"

//DEFINES

//Global variables
int8_t buff_read;

void handler_irq_ext(uint32_t id){
  if (id == EXT_INTR_0) {
    buff_read = 1;
  }
}

int main (int argc, char *argv[])
{

//Interrupts
plic_Init();
plic_irq_set_priority(EXT_INTR_0, 1);
plic_irq_set_trigger(EXT_INTR_0, kPlicIrqTriggerEdge);
plic_irq_set_enabled(EXT_INTR_0, kPlicToggleEnabled);

plic_assign_external_irq_handler(EXT_INTR_0, *handler_irq_ext);

CSR_SET_BITS(CSR_REG_MSTATUS, 0x8);
const uint32_t mask = 1 << 11;
CSR_SET_BITS(CSR_REG_MIE, mask);

polymul_mem_init();

//VARIABLES

mmio_region_t polymul_reg_base_addr = mmio_region_from_addr((uintptr_t)
        ↪ POLYMUL_PERIPH_START_ADDRESS);
int32_t buff_read_mem = 0;

//POLYMUL pointer
volatile int32_t *polymul_mem_ptr = (int32_t*) (POLYMUL_START_ADDRESS);

...
```

Code 5.13: Initialization and interrupt activaction in main.c

As it is seen, the main pointer `*polymul-mem-ptr` is also declared in order to address POLYMUL memory region. Then, `MEM-R` and `MEM-H` from polymul can be written as follows (*Code 5.14*).

```
//Memory load

//MEM R

*polymul_mem_ptr = 0x00000001;
polymul_mem_ptr++;

*polymul_mem_ptr = 0x00000001;
polymul_mem_ptr++;

*polymul_mem_ptr = 0xFFFFFFFF;
polymul_mem_ptr++;

*polymul_mem_ptr = 0x00000001;
polymul_mem_ptr++;

...
```

Code 5.14: Memory writes in POLYMUL BRAMs, from main.c

When POLYMUL `MEM-R and MEM-H` are written, the control signals can be driven to enable operation (`ctrl=1`), and the MCU could go to low-power mode while waiting for the interrupt request (*Code 5.15*).

```
//Control signals

mmio_region_write32(polymul_reg_base_addr, POLYMUL_CTRL_REG_OFFSET , 0xFFFFFFFF);
buff_read = 0;

while (buff_read == 0) {
  wait_for_interrupt();
}

mmio_region_write32(polymul_reg_base_addr, POLYMUL_CTRL_REG_OFFSET , 0x00000000);
```

Code 5.15: MCU control signals activation and interrupt wait, from main.c

At the end of the program, operation results can be read from `MEM-E` (*Code 5.16*). The program will finish successfully and the user will be able to read results directly from X-HEEP memory using `GDB`. ALternatively, the results could be printed through the UART, but this option is avoided because printing through the UART is very slow (and there are many values to read).

```
...

//Memory read

polymul_mem_ptr = POLYMUL_START_ADDRESS + 0x400*4; //MEM E ACCESS
for (i=0; i<401; i+=1){
    buff_read_mem = *polymul_mem_ptr;
    polymul_mem_ptr++;
}

return EXIT_SUCCESS;
}
```

Code 5.16: MCU memory read from MEM-E to get operation results, in main.c

5.3 eXtension Interface (X-IF)

The last implementation approach (and the most interesting one) for POLYMUL IP and X-HEEP is through the **CORE-V-XIF** (*eXtension Interface*), whose specification can be extracted from [OpenHWGroup, 2023d]. This is a special bus interface which is intended for extending RISC-V MCU instruction set with other custom or standarized instructions without changing the RTL of the CPU. This brings another implementation perspective, allowing the integration of a decentralized CPU among external HW IPs through a single communication interface. In this case, no interrupts or memory mapping are needed, as a decentralized module of the CPU will take control of some custom intructions, potentially achieving a better performance. X-IF allows register file access of the MCU from the external device, and it also permits DMA to the main MCU RAM memory.

Register files are used by the CPU to store ALU operands or results temporarilly, achieving high speed accesses (few clock cycles required) as these register files are very close to the ALU and access to storages with a higher hierarchy level is not required. Hence, having access to these register files is a key feature to get a high performance. Sometimes, a dedicated register bank could be required to be implemented on the custom coprocessor or external device accessing x-if, but it is not necessary for this case.

RISC-V architecture is characterized by having a load-store approach. This means that, usually, RISC-V instructions can be divided into three stages: loading operands in register files, performing ALU operations and storing results back in register files. Additionally, RV32I also includes dedicated instructions for accessing CSR registers, but this type of instructions is not relevant for the present implementation. X-IF is formed by a set of sub-interfaces to separate instructions processing into each required stage, but not every sub-interface is required for implementation (it depends on the application case). Specification from [OpenHWGroup, 2023d] must be read in order to understand every signal from every sub-interface, as well as the handshake and operation. The X-IF sub-interfaces that are going to be used for this particular implementation are listed below.

- **Issue Interface**: it is used to offload instructions from the main CPU to the external device or coprocessor. It has separated signals for request and response. Request signals are used for instructions offload and identification, while response lines are used for accepting or rejecting current offload attempt, providing more information about the instruction.
- **Commit Interface**: it is used by the CPU to notify the external device or coprocessor that the instruction offloading (issue transaction) has finished correctly, even if the instruction has been rejected. Every issue transaction needs of a commit transaction afterwards.
- **Memory Request/response Interface**: it is the specific interface for performing DMA (if necesarry for the current offloaded instruction), providing direct access to main CPU RAM memory.
- **Memory Result Interface**: it is the one for providing memory read results from the request/response interface.
- **Result Interface**: it is used to provide register file operation results when the instruction has finished. Exactly one transaction through the result interface is required when an instruction is allowed to be offloaded (through the commit interface) and the instruction is accepted through an issue response transaction, even if the instruction itself does not have any operation result to be stored back in the main CPU register files.

To make POLYMUL IP accelerator work with the X-IF, it will be necessary to change the implementation perspective from the previous subsection, as POLYMUL is not an external slave now. POLYMUL must work as a coprocessor capable of decoding instructions by itself, and then, performing DMA to the main CPU (without needing a request from the main CPU master). Consequently, the first step is to define which kind of tasks will the coprocessor (POLYMUL) do and which instructions type is going to be considered.

POLYMUL must carry out the following tasks in order to operate correctly.

1. Request input polynomial coefficients from the main CPU memory and load requested data into `MEM-R and MEM-H`.
2. Activate control signals (`ctrl=1`) to start operation once all the coefficients are loaded in `MEM-R and MEM-H`, and wait until operation has finished (`end-op=1`).
3. Perform DMA to main CPU memory in order to store all the operation results from `MEM-E`

As a result, three different custom instructions could be implemented in this case to perform these three steps. Two of them will be memory-to-memory instructions (one for loading from X-HEEP RAM to POLYMUL dedicated BRAMs and the other one for storing from POLYMUL BRAMs to X-HEEP RAM), and the last one will be a custom instruction for driving control signals, without any particular result. The three proposed instructions are 32-bit non-compressed (as well as the majority of RISC-V base instructions) (*Table 5.1, Table 5.2, Table 5.3*).

When defining custom instructions, it must be taken into account that X-HEEP has a bank of 32 registers (with 32 bits each) to store operands and results to/from the ALU. Then, as a naming convention, register source files are named as **rs1, rs2, rs3**... On the other hand, register destination file is named as **rd**. To address the registers bank, 5 bits are required (specified inside the correspondant instruction bit fields).

- **Mnemonic**: `xcld` (*eXternal Coefficient Load*).
- **Format**: xcld rs2, offset(rs1)
- **Description**: Adds value rs1 and the offset. The result is the starting address in X-HEEP memory where 32-bit coefficients will be read and loaded to POLYMUL memory. The specific polymul memory is defined in rs2.

31-27	**26-25**	**24-20**	**19-15**	**14-12**	**11-7**	**6-2**	**1-0**
offset[11:7]	offset[6:5]	rs2	rs1	101	offset[4:0]	11001	11

Table 5.1: XCLD instruction bit fields

- **Mnemonic**: `xmul` (*eXternal Multiplication Start*).
- **Format**: xmul
- **Description**: Launches polymul start operation.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
Undef	Undef	Undef	Undef	Undef	Undef	11101	11

Table 5.2: XMUL instruction bit fields

- **Mnemonic**: `xstr` (*eXternal Store Result*).
- **Format**: xstr offset(rs1)
- **Description**: Stores back the multiplication results, starting by X-HEEP memory address located in rs1. Offset is added to the the value in rs1 to calculate effective address.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:7]	offset[6:5]	Undef	rs1	101	offset[4:0]	11110	11

Table 5.3: XSTR instruction bit fields

Once these three required instructions are presented, it will be necessary to develop new HW RTL modules in SystemVerilog in order to make POLYMUL IP work as a coprocessor through the X-IF. The new device will be called `polymul-ss` (*Polymul subsystem*) and its architecture is presented in *Figure 5.4*. In addition, the new top-level entity instantiation template is presented in *Code 5.17*.

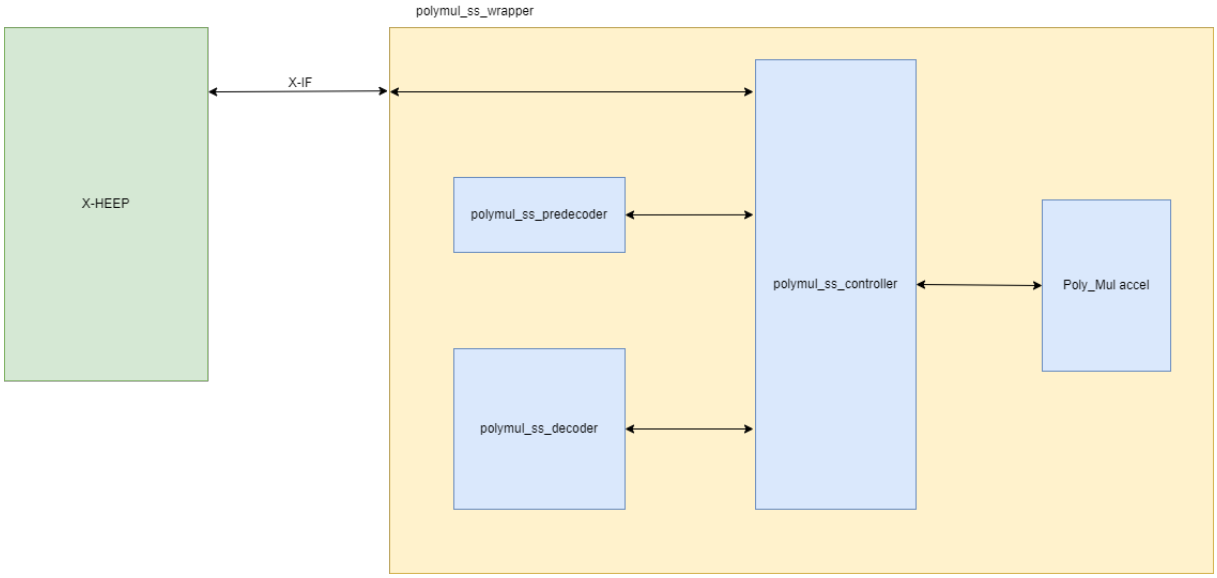

Figure 5.4: polymul-ss block diagram

To develop this architecture proposal, it will be necessary to create two new projects. One of them will be for developing and including all the RTL source files of `polymul-ss` (decoder, predecoder, controller and accelerator), while the other one will include `X-HEEP and polymul-ss` vendorizing, as well as all the simulation, SW and FPGA implementation files.

At the beginning, the source RTL submodules of `polymul-ss` will be explained *Table 6.4*, and *Table 6.5* includes the explanation of the involved Verilog packages. Then, RTL files will be listed in *Code 5.18, Code 5.19*.

```
polymul_ss #(
) polymul_ss_i (
    // clock and reset
    .clk_i                  (),
    .rst_ni                 (),

    // Issue Interface
    .x_issue_valid_i        (),
    .x_issue_ready_o        (),
    .x_issue_req_i          (),
    .x_issue_resp_o         (),

    // Commit Interface
    .x_commit_valid_i       (),
    .x_commit_i             (),

    // Memory Request/Response Interface
    .x_mem_valid_o          (),
    .x_mem_ready_i          (),
    .x_mem_req_o            (),
    .x_mem_resp_i           (),

    // Memory Result Interface
    .x_mem_result_valid_i   (),
    .x_mem_result_i         (),

    // Result Interface
    .x_result_valid_o       (),
    .x_result_ready_i       (),
    .x_result_o             ()
);
```

Code 5.17: polymul-ss instantiation template

Name	Description	SystemVerilog File
`polymul-ss`	Top level module	polymul-ss.sv
`polymul-ss-predecoder`	Decides when and if an offload attempt from the core is accepted or not	polymul-ss-predecoder.sv
`polymul-ss-decoder`	Decodes instructions	polymul-ss-decoder.sv
`Poly-Mul`	Main processing unit	polymul.sv
`polymul-ss-controller`	Control unit for the whole subsystem and the CV-X-IF	polymul-ss-controller.sv

Table 5.4: Submodules of polymul-ss

Name	Description	SystemVerilog File
polymul-ss-pkg	Defines coprocessor specific structs, cv-x-if structs and the Poly-Mul implementation	polymul-ss-pkg.sv
polymul-ss-instr-pkg	Instruction masks for the polymul-ss-decoder	polymul-ss-predecoder.sv
polymul-ss-prd-pkg	Identifies instructions for the predecoder according to the mask. Activates acceptance signals	polymul-ss-decoder.sv

Table 5.5: Submodules of polymul-ss

```
module polymul_ss_decoder (
    input  logic                              [31:0] instr_i,
    output polymul_ss_pkg::decoded_op_t              decoded_op_o
);
    always_comb begin
        unique casez (instr_i)
            polymul_ss_instr_pkg::X_MUL: begin
                decoded_op_o.reg_sel      = 3'b000;
                decoded_op_o.is_launch_op = 1'b1;
                decoded_op_o.is_ldst      = 2'b00;
            end
            polymul_ss_instr_pkg::X_STR: begin
                decoded_op_o.reg_sel      = 3'b001;
                decoded_op_o.is_launch_op = 1'b0;
                decoded_op_o.is_ldst      = 2'b10;
            end
            polymul_ss_instr_pkg::X_CLD: begin
                decoded_op_o.reg_sel      = 3'b011;
                decoded_op_o.is_launch_op = 1'b0;
                decoded_op_o.is_ldst      = 2'b01;
            end
        endcase
    end
endmodule // polymul_ss_decoder
```

Code 5.18: polymul-ss decoder file listing

```
package polymul_ss_instr_pkg;
  localparam logic [31:0] X_MUL              = 32
        ↪ 'b??????????????????????????1110111;
  localparam logic [31:0] X_STR              = 32
        ↪ 'b???????????????????101?????1111011;
  localparam logic [31:0] X_CLD              = 32
        ↪ 'b???????????????????101?????1100111;
endpackage // polymul_ss_instr_pkg
```

Code 5.19: polymul-ss-instr-pkg file listing

In *Code 5.19*, the custom instructions are defined in the package, and then, the decoder (*Code 5.18*) is capable of identifying each 32-bit defined instruction word. The decoder will tell the

`polymul-ss-controller` which source registers (`rs1 or rs2`) should be selected and which operation type (load or store) is invloved (with the 2-bit signal `is-ldst`), as well as the identification of the launch operation (`xmul`, if it is received). Then, the predecoder in *Code 6.26* decides if a defined instruction inside the package must be accepted or not, and it also extracts more information (for example, if the instruction itself is a memory instruction or not).

In addition, there is a `polymul-ss-pkg` file where all the necessary parameters and data types (custom data types and x-if data types) are defined.

Then, the remaining module is the `polymul-ss-controller`, which is the most complex and largest one. It is the one capable of coordinating the whole subsystem, transferring data from the `x-if` to POLYMUL BRAMs, and providing requested results through this interface. Furthermore, the `polymul-ss-controller` coordinates the handshake from issue, commit, memory and response interfaces, communicating properly with the CPU and performing DMA when necessary.

X-IF signals description and explanaition must be read from [OpenHWGroup, 2023d], and there are some of them, such as the ones for writing or specifying *Extension Context Status (ECS)* data or the ones for exceptions reporting, which will not be used in this case and are set to `'0`. These lines are present in the *Result Interface* and are not required for the correct behavior of POLYMUL.

The `polymul-ss-controller` module is mostly sequential, as the majority of the signals and data from the x-if must be stored in registers while a specific operation is carried out. Nevertheless, the controller also has combinational processes and assignments, such as for the issue response signals assignment. While *polymul-ss* is carrying out an operation, some x-if signals might be hold, but not all of them, and this is why registers are very important in this case. The specific handshake an rules for each sub-interface (issue, commit, memory and result) must always be considered.

```
1  ...
2  //Instruction data assignment to decoder
3  assign instr_data = x_issue_req_i.instr;
4
5  always_ff @(posedge clk_i, negedge rst_ni) begin //decoded load_store
       ↪ assignments
6      if (!rst_ni) begin
7          dec_loadstore <= '0;
8          dec_launch    <= '0;
9      end
10     else begin
11         if (x_issue_valid_i) begin
12             dec_loadstore <= decoded_op.is_ldst;
13             dec_launch    <= decoded_op.is_launch_op;
14         end
15         else if (end_op) begin
16             dec_launch <= 1'b0;
17         end
18     end
19 end
20
21 always_ff @(posedge clk_i, negedge rst_ni) begin
22     if (!rst_ni) begin
23         accept_q <= 1'b0;
24         x_mem_req_o_mode <= '0;
25     end
26     else begin
27         accept_q <= 1'b0;
28         if (x_issue_valid_i && prd_rsp.p_accept && x_commit_i.id ==
           ↪ x_issue_req_i.id) begin
29             accept_q <= 1'b1;
30             x_mem_req_o_mode <= x_issue_req_i.mode;
31         end
32     end
33 end
34 ...
```

Code 5.20: Decode and acceptance registers assignments in polymul-ss-controller

In *Code 5.20*, it is seen how signals from load/store or launch decoding are stored in registers through sequential processes. Then, the `accept` signal is also registered in order to keep the required decoded information for the combination of the `issue` and `commit` transactions. Then, in *Code 5.21*, the corresponding register source operands (`rs1 and rs2`) are extracted. In this case, the register source `rs1` will be addressed from the corresponding instruction bits and read from X-HEEP register bank. It will contain the source address of the first memory position where POLYMUL input coefficients are located in X-HEEP main memory. On the other hand, source register `rs2` will be used to select the memory position in X-HEEP memory where the correspondant `MEM-R or MEM-H` is selected, performing an OR logic function reduction (*line 26, Code 5.21*).

```
...
always_ff @(posedge clk_i, negedge rst_ni) begin //rs1 operand
        ↪ assignment
if (!rst_ni) begin
    rs1_valid   <= 1'b0;
    rs1_operand <= '0;
end
else begin
    rs1_valid <= 1'b0;
    if (decoded_op.reg_sel[0] && x_issue_valid_i && prd_rsp.p_accept &&
        ↪ x_commit_i.id == x_issue_req_i.id &&
        ↪ !decoded_op.is_launch_op) begin
        rs1_operand <= x_issue_req_i.rs[0] + offset;
        rs1_valid   <= 1'b1;
    end
end
end
always_ff @(posedge clk_i, negedge rst_ni) begin //rs2 operand
        ↪ assignment
if (!rst_ni) begin
    rs2_valid   <= 1'b0;
    rs2_operand <= '0;
    mem_sel     <= '0;
end
else begin
    rs2_valid <= 1'b0;
    if (decoded_op.reg_sel[1] && x_issue_valid_i && prd_rsp.p_accept &&
        ↪ x_commit_i.id == x_issue_req_i.id &&
        ↪ !decoded_op.is_launch_op) begin
        rs2_operand <= x_issue_req_i.rs[1];
        rs2_valid   <= 1'b1;
        mem_sel     <= |x_issue_req_i.rs[1];
    end
end
end
...
```

Code 5.21: rs1 and rs2 operands assignments in polymul-ss-controller

In *Code 5.21*, it is also checked at the beginning that the corresponding `id's` from `commit` and `issue` interfaces are the same, in order to ensure that the decoded instruction is always the same. When an instruction is already offloaded in the coprocessor, these `id` signals will be auto-incremented by the CPU in order to signal the next instruction.

```
...
always_ff @(posedge clk_i, negedge rst_ni) begin : p_mem_req_id
if (!rst_ni) begin
    x_mem_req_o_id <= '0;
end
else begin
    if (finish_polymul || end_op) begin
        x_mem_req_o_id   <= x_mem_req_o_id + 1;//x_issue_req_i.id;
    end
end
end

always_ff @(posedge clk_i, negedge rst_ni) begin : p_mem_result_valid_q
if (!rst_ni) begin
    x_mem_result_valid_q <= '0;
end
else begin
    x_mem_result_valid_q <= x_mem_result_valid_i;
end
end

always_ff @(posedge clk_i, negedge rst_ni) begin //Memory loading and
        ↪ storing control flags assignments
if (!rst_ni) begin
    mem_loading    <= 1'b0;
    mem_storing    <= 1'b0;
    x_mem_req_o_be <= '0;
    x_mem_valid_o  <= 1'b0;
end
else begin
    if (x_commit_valid_i && accept_q && !x_commit_i.commit_kill) begin
        if (dec_loadstore == 2'b01) begin
            mem_loading      <= 1'b1;
            x_mem_req_o_be   <= '1;
            x_mem_valid_o    <= 1'b1;
        end
        else if (dec_loadstore == 2'b10 && end_op) begin
            mem_storing      <= 1'b1;
            x_mem_req_o_be   <= '1;
            x_mem_valid_o    <= 1'b1;
        end
    end
    if (finish_polymul) begin
        mem_loading    <= 1'b0;
        mem_storing    <= 1'b0;
        x_mem_req_o_be <= '0;
    end
    if (x_mem_req_o_last) begin
        x_mem_valid_o  <= 1'b0;
    end
end
end
...
```

Code 5.22: Memory and internal control signals assignments in polymul-ss-controller

```
...
//Memory interface assignments
assign x_mem_req_o_size = 3'b100;
assign x_mem_req_o_attr = '0;
assign x_mem_req_o_spec = '0;
assign x_mem_req_o_addr = rs1_operand + addr_cnt - 4 ;
//Request memory process
always_ff @( posedge clk_i, negedge rst_ni ) begin : p_req_mem
    if (!rst_ni) begin
        addr_cnt <= '0;
        finish   <= '0;
    end
    else begin
        finish           <= 1'b0;
        if ( mem_loading || mem_storing ) begin
            if (addr_cnt < NUM_LD_COEFS * 4) begin
                addr_cnt <= addr_cnt + 4;
            end else begin
                finish   <= 1'b1;
            end
        end
        else begin
            addr_cnt <= '0;
        end
    end
end
...
```

Code 5.23: MCU memory addressing process in polymul-ss-controller

In *Code 5.22*, some internal control signals like `mem-loading` and `mem-storing` are included to indicate that the coprocessor is still handling an instruction and needs to halt the x-if in order to make the CPU wait until finished. When the coprocessor finishes handling the instruction, it must de-assert `mem-loading` (if the current instruction is `xcld`) or `mem-storing` (if the current instruction is `xstr`) and start a result transaction, in order to notify the CPU that the instruction has finished.

Then, in *Code 5.23*, the process for addressing X-HEEP memory can be visualized. In this case, while handling a load-store instruction, the coporcessor, will auto-increment address by 4 (pointing each 4-byte word, 32 bits), and each required polynomial coefficient will appear through the memory result interface.

```
...
//Polymul BRAM address assignments
always_comb begin
    if (mem_loading && x_mem_result_valid_i) begin
        if (!mem_sel) begin //MEM_R selected
            bram_addr_r = addr_cnt_polymul[BRAM_ADDR_BUS_WIDTH-1:0];
        ↪ //- 2;
            bram_addr_h = '0;
            bram_addr_e = '0;
        end
        else if (mem_sel) begin //MEM_H selected
            bram_addr_h = addr_cnt_polymul[BRAM_ADDR_BUS_WIDTH-1:0];
        ↪ //- 2;
            bram_addr_r = '0;
            bram_addr_e = '0;
        end
        else begin
            bram_addr_r = '0;
            bram_addr_h = '0;
            bram_addr_e = '0;
        end
    end
    else if (mem_storing) begin //MEM_E selected
        bram_addr_e = addr_cnt_polymul[BRAM_ADDR_BUS_WIDTH-1:0]; //- 2;
        bram_addr_r = '0;
        bram_addr_h = '0;
    end
    else begin
        bram_addr_e = '0;
        bram_addr_r = '0;
        bram_addr_h = '0;
    end
end
...
```

Code 5.24: POLYMUL BRAM address assignments in polymul-ss-controller

Code 5.24 listing shows how BRAM memories from POLYMUL accelerator are addressed, depending on the memory selection (`mem-sel`) and depending on the offloaded instruction (`xcld` with `mem-loading=1` or `xstr` with `mem-storing=1`).

On the other hand, *Code 5.25* includes the process that is used to auto-increment POLYMUL BRAM memory addresses through every memory position. It must start two clock cycles after X-HEEP memory starts being address, as memory result interface has a latency of two clock cycles at the beginning to provide the first memory read result. Consequently, the coprocessor will have enough time to take the read value from memory result interface and load it back in the corresponding POLYMUL BRAM (`MEM-R or MEM-H`).

```
...
//Load polymul memory process
always_ff @( posedge clk_i, negedge rst_ni ) begin : p_load_mem
    if (!rst_ni) begin
        addr_cnt_polymul <= '0;
        finish_polymul   <= '0;
        x_mem_req_o_last <= '0;
    end
    else begin
        x_mem_req_o_last <= '0;
        finish_polymul   <= '0;
        if ((mem_loading && x_mem_result_valid_i) || mem_storing ) begin
            if (addr_cnt_polymul < NUM_LD_COEFS) begin
                if (addr_cnt_polymul == NUM_LD_COEFS - 2) begin
                    x_mem_req_o_last <= 1'b1;
                end
                addr_cnt_polymul <= addr_cnt_polymul + 1;
            end
            else begin
                finish_polymul   <= 1'b1;
            end
        end
        else begin
            addr_cnt_polymul <= '0;
        end
    end
end
assign bram_en_r = (!mem_sel) ? x_mem_result_valid_i : 1'b0;
assign bram_en_h = (mem_sel) ? x_mem_result_valid_i : 1'b0;
...
```

Code 5.25: Load polymul memory process in polymul-ss-controller

Aditionally, *Code 5.26* shows how POLYMUL BRAM write enable (`we`) signals are driven, as well as how data buses are assigned depending on the instruction that is currently being processed. Assignments in this case are intended to be combinational, as data buses and write enable signals must be assigned immediately.

```
...
always_comb begin //Polymul en an we BRAMs assignments
bram_we_r           = '0;
bram_data_in_r      = '0;
bram_we_h           = '0;
bram_data_in_h      = '0;
bram_en_e           = '0;
bram_we_e           = '0;
bram_data_in_e      = '0;
x_mem_req_o_we      = '0;
x_mem_req_o_wdata   = '0;
if (mem_loading && x_mem_result_valid_i) begin //x_mem_result_i.id ==
        ↪ x_issue_req_i.id
    if (!mem_sel) begin //MEM_R selected
        bram_we_r      = 1'b1;
        bram_data_in_r =
        ↪ x_mem_result_i.rdata[BRAM_DATA_BUS_WIDTH_R-1:0];
    end
    else begin //MEM_H selected
        bram_we_h      = 1'b1;
        bram_data_in_h = x_mem_result_i.rdata[BRAM_DATA_BUS_WIDTH-1:0];
    end
end
else if (mem_storing) begin //MEM_E selected
    bram_en_e                                   = 1'b1;
    x_mem_req_o_we                              = 1'b1;
    x_mem_req_o_wdata[BRAM_DATA_BUS_WIDTH-1:0] = bram_data_out_e;
end
if (x_commit_valid_i && dec_loadstore == 2'b10) begin
    bram_en_e = 1'b1;
end
end
...
```

Code 5.26: POLYMUL BRAM control signals and data buses assignments

Then, *Code 5.27* points how POLYMUL `ctrl` signal is handled. In this case, it is only driven when `xmul` instruction is offloaded. This means that a launch operation has started after all the polynomial coeffiecients are already loaded, and `ctrl` signal must be set to '1'. This `xmul` instruction is different from the other two, as only one control signal must be driven, and after the corresponding `issue` and `commit` transactions, the coprocessor halts the x-if waiting for `end-op` signal, which is asserted when POLYMUL ALUs have finished performing multiplications. When `end-op` is asserted, the coprocessor must initiate a `result` transaction, to allow the next instruction offloading through the x-if.

```
...
//Clock and reset assignment to polymul
assign bram_clk_r = clk_i;
assign bram_clk_h = clk_i;
assign bram_clk_e = clk_i;
assign bram_rst_r = !rst_ni;
assign bram_rst_h = !rst_ni;
assign bram_rst_e = !rst_ni;

assign clk_polymul = clk_i;
assign rst_polymul = !rst_ni;

//Polymul ctrl signal
always_ff @( posedge clk_i, negedge rst_ni ) begin : p_ctrl_valid
    if (!rst_ni) begin
        ctrl_valid <= '0;
    end
    else begin
        if (dec_launch && x_commit_valid_i && !x_commit_i.commit_kill)
        ↪ begin
            ctrl_valid <= 1'b1;
        end
        else if (end_op) begin
            ctrl_valid <= 1'b0;
        end
    end
end

always_ff @( posedge clk_i, negedge rst_ni ) begin : p_ctrl
    if (!rst_ni) begin
        ctrl <= '0;
    end
    else begin
        if (ctrl_valid && !finish_polymul && finish_polymul_q) begin
            ctrl <= 1'b1;
        end
        else if (end_op) begin
            ctrl <= 1'b0;
        end
    end
end
...
```

Code 5.27: POLYMUL ctrl signal, clock and reset assignments

Finally, *Code 5.28* includes the `result interface` signals and `x-issue-ready` handling. Result transactions must start when coprocessor operations during instruction execution have already finished. Consequently, a result transaction is launched to let X-HEEP know that the current instruction has finished. Afterwards, when X-HEEP receives the result transaction and signals `x-result-ready` signal accordingly, the signal `x-issue-ready` is asserted by the coporcessor to indicate X-HEEP that the next instruction can be offloaded. The signal `x-issue-ready` must be de-asserted when an instruction is offloaded until the corresponding result transaction has been released. Then, X-HEEP notifies the recipt through `x-result-ready` signal.

```
...
//Result interface assignments
assign x_result_o.id      = x_result_o_id;
assign x_result_o.data    = '0;
assign x_result_o.rd      = '0;
assign x_result_o.we      = '0;
assign x_result_o.ecsdata = '0;
assign x_result_o.ecswe   = '0;
assign x_result_o.exc     = '0;
assign x_result_o.exccode = '0;
assign x_result_o.err     = '0;
assign x_result_o.dbg     = '0;
//Result id assignment
always_ff @( posedge clk_i, negedge rst_ni ) begin : p_x_result_id
    if (!rst_ni) begin
        x_result_o_id <= '0;
    end
    else begin
        if (x_result_valid_o) begin
            x_result_o_id <= x_result_o_id + 1;
        end
    end
end
always_ff @(posedge clk_i, negedge rst_ni) begin: p_issue_ready
    if (!rst_ni) begin
        x_issue_ready_o <= 1'b0;
    end
    else begin
        x_issue_ready_o <= 1'b1;
        if ((!x_mem_result_valid_i && x_mem_result_valid_q) ||
        ↪ x_mem_result_valid_i || (x_commit_valid_i &&
        ↪ x_issue_ready_o) || mem_loading || mem_storing) begin
            x_issue_ready_o <= 1'b0;
        end
        else if (dec_launch && !end_op) begin
            x_issue_ready_o <= 1'b0;
        end
    end
end
...
always_ff @( posedge clk_i, negedge rst_ni) begin : p_result
if (!rst_ni) begin
    x_result_valid_o <= 1'b0;
end
else begin
    x_result_valid_o <= 1'b0;
    if ((!x_mem_result_valid_i && x_mem_result_valid_q) ||
        ↪ x_result_ready_i) begin
        x_result_valid_o <= 1'b1;
        if (dec_launch && !end_op) begin
            x_result_valid_o <= 1'b0;
        end
    end
end
end
...
```

Code 5.28: Result interface assignments in polymul-ss-controller

To understand properly how `x-if` operates, some simulation waveforms will be included below, visualizing all the corresponding signals involved through the sub-interfaces (issue, commit, memory and result).

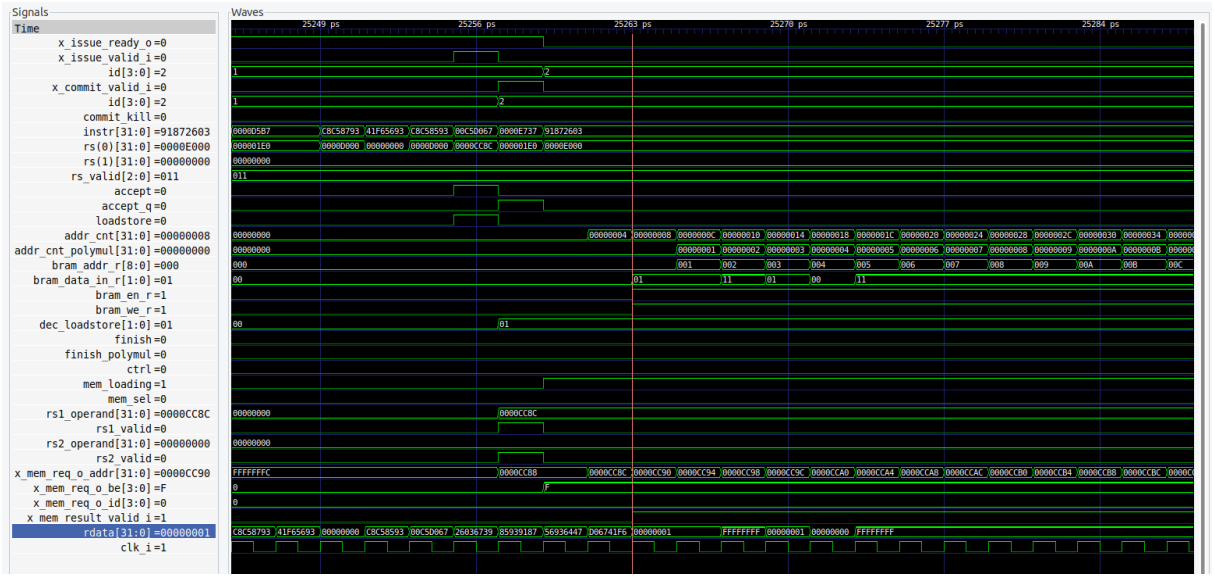

Figure 5.5: xcld instruction offload during simulation

In *Figure 5.5*, it is seen when an `xcld` instruction is offloaded from X-HEEP to POLYMUL coprocessor. The coprocessor accepts immediately the instruction through the predecoder and reads `rs1 and rs2` through the issue request interface, and the CPU answers with a commit transaction, signaling `commit-valid`. Then, `x-issue-ready` is de-asserted by the coprocessor and multiple memory transactions through the memory interface start, in order to request polynomial coefficients from X-HEEP main memory.

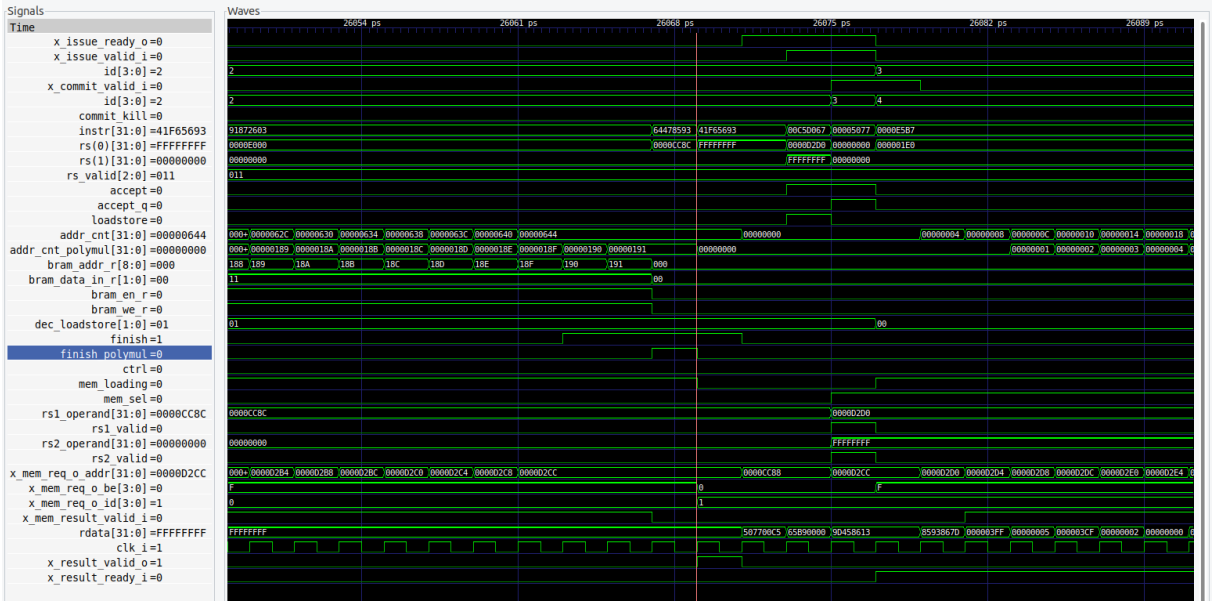

Figure 5.6: xcld instruction finished during simulation

Then, in *Figure 5.6*, the instruction finishes when all the corresponding X-HEEP memory positions are read, and a result transaction is released signaling `x-result-valid-o` and activating `finish` and `finish-polymul` internal control signals. For these instructions, result transaction

data is set to `'0` because it is not necessary to write anything back to any register file from X-HEEP. However, as it was said before, a result transaction is required anyway. When X-HEEP receives the result transaction, it asserts `x-result-ready` back.

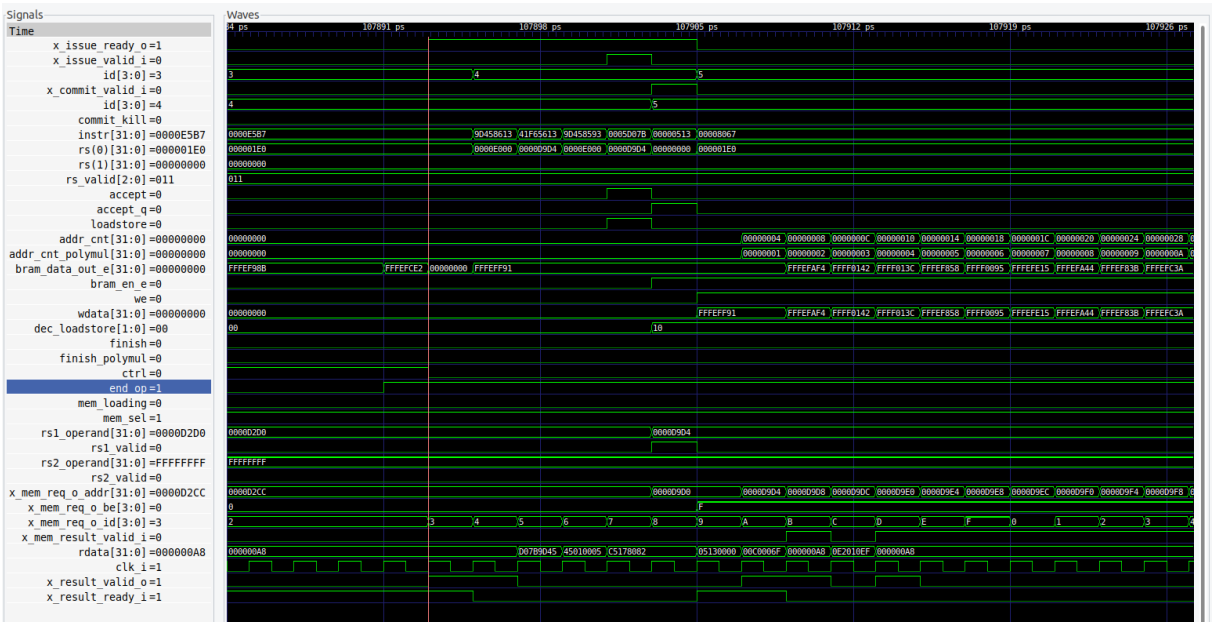

Figure 5.7: xstr instruction offload during simulation

Finally, in *Figure 5.7*, an `xstr` instruction is offloaded when multiplication results from POLYMUL are required to be provided back to X-HEEP, and `end-op=1`. It is seen how `MEM-E` is enabled, and the correspondig read data is assigned to `wdata` bus of `x-mem-req-o` interface, asserting the correspondig `we` signal. Then all the memory positions are addressed as expected until the instruction finishes.

To conclude this implementation case, it is necessary to describe how this HW is controlled from a SW perspective, as these custom instructions are not supported by RISC-V compiler. Fortunately, there are available methods that can be integrated for this purpose in order to inject machine code directly in some specific fragments of a SW application. This is done by `ROCC` instruction macros (*Rocket Custom Core*), which are extracted from [California Berkeley, 2023]. These `ROCC` instruction macros can be written in `C` programming language, for example, and they are used to define a specific instruction word in machine code. Afterwards, macros can be called from any specific point of a SW application, and then executed using `asm volatile` directive.

In *Code 5.29*, there is a `C` header file which includes the definition of the macros that are going to be used in the SW to call the defined custom instructions `xcld, xmul and xstr`. By default, if the `x-if` is enabled by HW, X-HEEP will receive these non-recognized instruction words and will always offload them through the `x-if`, waiting for a response from any external device. In this case, if POLYMUL coprocessor answers, an illegal instruction exception is not raised and the SW execution flow becomes dependant on the `x-if` handshake until the instruction has finished. This means that the SW will not continue to the next instruction until the corresponding transaction through the `result interface` is performed, as explained before from the HW perspective.

```
#ifndef SRC_MAIN_C_ROCC_H
#define SRC_MAIN_C_ROCC_H
#include <stdint.h>
#define STR1(x) #x
#define STR(x) STR1(x)
#define EXTRACT(a, size, offset) (((~(~0 << size) << offset) & a) >> offset)
#define CUSTOMX_OPCODE(x) CUSTOM_ ## x
#define CUSTOM_0 0b1110111
#define CUSTOM_1 0b1111011
#define CUSTOM_2 0b1100111
#define CUSTOMX(X, xd, xs1, xs2, rd, rs1, rs2, funct)   \
    CUSTOMX_OPCODE(X)                                   |              \
    (rd                 << (7))                         |              \
    (xs2                << (7+5))                       |              \
    (xs1                << (7+5+1))                     |              \
    (xd                 << (7+5+2))                     |              \
    (rs1                << (7+5+3))                     |              \
    (rs2                << (7+5+3+5))                   |              \
    (EXTRACT(funct, 7, 0) << (7+5+3+5+5))
#define ROCC_INSTRUCTION_SS(X, rs1, rs2, funct) \
    ROCC_INSTRUCTION_I_R_R(X, 0, rs1, rs2, funct, 11, 12)
#define ROCC_INSTRUCTION_S(X, rs1, funct) \
  ROCC_INSTRUCTION_I_R_I(X, 0, rs1, 0, funct, 11)
#define ROCC_INSTRUCTION(X, funct) \
  ROCC_INSTRUCTION_I_I_I(X, 0, 0, 0, funct)
#define ROCC_INSTRUCTION_I_R_R(X, rd, rs1, rs2, funct, rs1_n, rs2_n) {     \
    register uint64_t rs1_ asm ("x" # rs1_n) = (uint64_t) rs1;             \
    register uint64_t rs2_ asm ("x" # rs2_n) = (uint64_t) rs2;             \
    asm volatile (                                                         \
        ".word " STR(CUSTOMX(X, 1, 0, 1, rd, rs1_n, rs2_n, funct)) "\n\t" \
        :: [_rs1] "r" (rs1_), [_rs2] "r" (rs2_));                          \
}
#define ROCC_INSTRUCTION_I_R_I(X, rd, rs1, rs2, funct, rs1_n) {          \
    register uint64_t rs1_ asm ("x" # rs1_n) = (uint64_t) rs1;           \
    asm volatile (                                                       \
        ".word " STR(CUSTOMX(X, 1, 0, 1, rd, rs1_n, rs2, funct)) "\n\t" \
        :: [_rs1] "r" (rs1_));                                           \
}
#define ROCC_INSTRUCTION_I_I_I(X, rd, rs1, rs2, funct) {                 \
    asm volatile (                                                       \
        ".word " STR(CUSTOMX(X, 1, 0, 1, rd, rs1, rs2, funct)) "\n\t" ); \
}
#endif  // SRC_MAIN_C_POLYMUL_TEST
```

Code 5.29: Rocket Custom Core C header file

It can be visualized how in *Code 5.29*, custom `OPCODEs` are defined in order to identify the correspondant custom instruction, and then, input parameters are located in the corresponding `rs` (*register source*) or `offset` bit fields of the instruction, identifying the desired values or memory positions.

Finally, the main SW application turns out to be very simple. It just defines three functions for calling each custom instructions with its input parameters and allocates three memory regions for storing input coefficients and results. Then, functions are called until they finish and the program terminates. Template is listed in *Code 5.30*.

```c
#include <stdio.h>
#include <stdlib.h>
#include "core_v_mini_mcu.h"
#include "x-heep.h"
#include "csr.h"
#include "rocc.h"
#define POLY_COEFS_DEPTH 401
static inline void load_coefs(uint32_t *mem_start, uint32_t *mem_select) {
    //asm volatile ("xcld");
    ROCC_INSTRUCTION_SS(2, mem_start, mem_select, 0);
}
static inline void store_coefs(uint32_t *e_start) {
    //asm volatile ("xstr");
    ROCC_INSTRUCTION_S(1, e_start, 0);
}
static inline void launch_polymul(void) {
    //asm volatile ("xmul");
    ROCC_INSTRUCTION(0, 0);
}
int main (int argc, char *argv[])
{
    static uint32_t r_coefs_array_4B[POLY_COEFS_DEPTH] __attribute__ ((aligned (4
        ↪ ))) = { ... };
    static uint32_t h_coefs_array_4B[POLY_COEFS_DEPTH] __attribute__ ((aligned (4
        ↪ ))) = { ... };
    static uint32_t e_coefs_array_4B[POLY_COEFS_DEPTH] __attribute__ ((aligned (4
        ↪ ))) = { 0 };
    static uint32_t mem_r_sel[1] __attribute__ ((aligned (4))) = { 0x00000000 };
    static uint32_t mem_h_sel[1] __attribute__ ((aligned (4))) = { 0xFFFFFFFF };
    volatile uint32_t *p_mem_r_sel = (uint32_t*) mem_r_sel;
    volatile uint32_t *p_mem_h_sel = (uint32_t*) mem_h_sel;
    load_coefs(r_coefs_array_4B, *p_mem_r_sel); //Load r-coefs in polymul R-MEM
    load_coefs(h_coefs_array_4B, *p_mem_h_sel); //Load h-coefs in polymul H-MEM
    launch_polymul(); //This will take several clock cycles 'till polymul finishes
        ↪ ...
    store_coefs(e_coefs_array_4B); //Store results back to X-HEEP
    return 0;
}
```

Code 5.30: Rocket Custom Core C header file

6 SETUP

6.1 FPGA Device

The device selected is Xilinx PYNQ-Z1 [Digilent, 2023], which includes a Zynq-7000 SoC with a dual-core ARM Cortex A9 processor and a xc7z020clg400-1 FPGA (*Fig.6.1*). It can be controlled, configured and programmed using Vivado Design Suite. Most significant features and characteristics are included in *Table 6.1, Table 6.2*.

This board is adequate for academic projects like this one. Moreover, the balance between price and hardware resources is acceptable regarding other available solutions in the market. The PYNQ-Z1 will be capable of handling MCUs such as X-HEEP and additional HW extensions or custom HW accelerators.

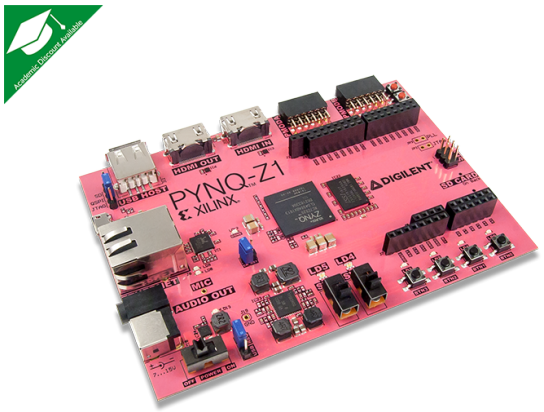

Figure 6.1: PYNQ-Z1 Board

Logic slices	13300
6-input LUTs	53200
Flip-Flops	106400
Block RAM	630 KB
DSP Slices	220
Clock	Zynq PLL (4 outputs), 4 PLLs, 4 MMCMs, 125 MHz external clock
ADC	Dual-channel, 1 MSPS

Table 6.1: Key FPGA specifications

USB	USB-UART, USB-JTAG, Programmer USB Host
Ethernet	Gigabit Ethernet PHY
HDMI	Sink (Input) and Source (Output)
Microphone	Microphone with PDM interface
Audio	PWM driven mono audio output with 3.5 mm jack
PMOD connectors	2
Other connectors	Arduino ChipKit shield connector

Table 6.2: Connectivity and On-board I/O FPGA specifications

6.2 X-HEEP implementation

The first step is to prepare a new Linux machine with Ubuntu 20 OS, which is the one tested for compiling and implementing X-HEEP project. Then, it is necessary to install all the required SW and tools, such as Vivado, Python Anaconda, Verilator, Verible and RISC-V GNU toolchain, setting up all the necessary Linux environment variables as indicated in X-HEEP documentation from its GitHub repository [ESL-EPFL, 2023]. Then, the following bash command in Linux terminal is executed:

```
$ make mcu-gen
```

X-HEEP compilation is managed using Makefiles, which are specific files that include compilation rules, calling programs or scripts to start the building process. In this case, *make mcu-gen* points to a rule inside the main Makefile that calls the script *mcu-gen.py*. This script generates Verilog design files from templates (*.tpl* files on the repository) depending on the specified options when calling *make mcu-gen*.

Then, a *hello world* SW example application program can be compiled. This a simple reference application written in C programming language, which is specially compiled for RISC-V architecture (using *riscv gnu toolchain*) and prints a *Hello World!* message through the UART.

```
$ make app PROJECT=hello_word TARGET=sim
```

This command will compile *hello world* application for Verilator simulation target. Then, it is time to call FuseSoC tool in order to make all the source files and then start simulation.

```
$ make verilator-sim
$ cd ./build/openhwgroup.org_systems_core-v-mini-mcu_0/sim-verilator
$ ./Vtestharness +firmware=../../../sw/build/main.hex
```

The *make verilator-sim* rule will call FuseSoC inside the Makefile and then, launching *Vtestharness* executable inside the build folder, the Verilator simulation of X-HEEP platform will start. The output terminal messages of the simulation are presented in *Fig.5.5*.

Program Finished with value 0 means that an *EXIT SUCCESS* value has been returned, so the program is executed successfully. Then, inside the build folder, there is a log file *uart.log* that can be opened, and all the UART output messages will be collected there. A *Hello World!* message will be seen in this file.

Simulation launching when using Verilator is comfortable, as many different SW programs can be developed and tested using C programming language. X-HEEP has a set of SW examples inside its repository, testing all the different HW resources available, such as external peripherals, DMA, GPIOs or SPI.

Once the simulation is tested, the target can be changed to *pynq-z2* in order to perform all the building tasks for the PYNQ board. X-HEEP is intended to be built for PYNQ-Z2 board, which is different from PYNQ-Z1, but the difference between these two boards is not relevant for this implementation case, as the differences take place in some PMOD, audio interfaces or external attachment ports, but the SoC Zynq-7000 is the same in both cases. The following command must be executed.

```
$ make vivado-fpga FPGA_BOARD=pynq-z2 FUSESOC_FLAGS=--flag=use_bscane_xilinx
```

It is important to notice that the *use bscan xilinx* flag must be enabled in order to activate Xilinx boundary scan interface for the PYNQ board. The Xilinx Boundary Scan interface is an

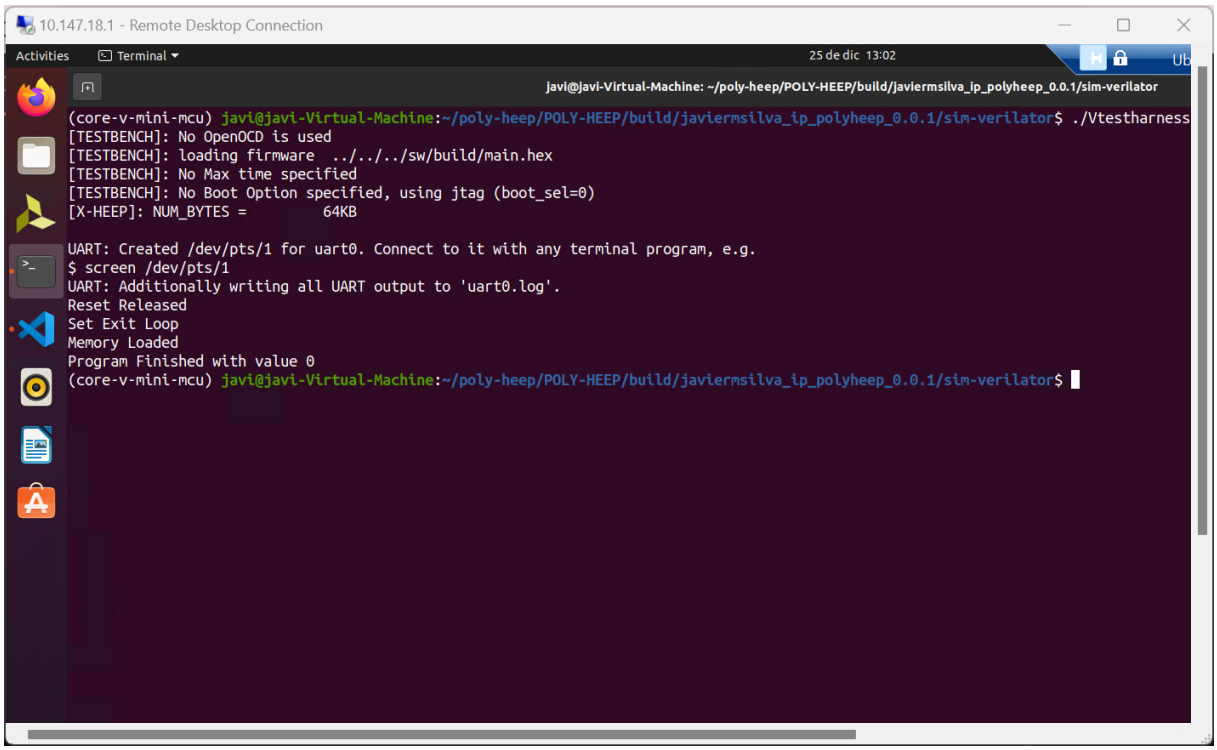

Figure 6.2: Verilator sim Linux terminal output

IP developed by Xilinx which is capable of providing JTAG TAP (*Test Access Port*) Controller access through the external USB *FTDI* (*Future Technology Devices International Ltd.*) port. Once executed the command above, Vivado is launched in command line triggering synthesis and implementation. Synthesis is the process of translating every HDL description into a RTL model (logic circuit) which will be finally burned in the FPGA. When running implementation, logical and path optimizations, timing constraints analysis, pin assign and place and route will be carried out in order to ensure correct operation of the circuit when the FPGA is programmed. Finally, Vivado will provide the bitstream, which is a *.bit* file that can be burned directly with Vivado Hardware Manager.

6.3 Debugging environment

Now, as the FPGA is already programmed, the debugging environment should be set up. The supported tool to stablish connection with X-HEEP is OpenOCD. OpenOCD is a tool that creates a remote connection through a TCP (*Transmission Control Protocol*) port in localhost (Linux machine). Then, it is possible to request a connection with the microcontroller using GDB (*GNU Debugger*), being capable of programming, seeing internal registers and memory, setting breakpoints and using other debugging features.

By default, OpenOCD will not be able to stablish connection with Xilinx BSCAN through USB port because it needs an input configuration file. This configuration file (*.cfg*) is necessary to initialize and address JTAG registers through FTDI USB driver (*Fig.5.6*), as well as other configuration parameters, and it is indicated when calling openocd command in Linux bash.

```
$ openocd -f <path to cfg file>
```

For the particular case of the Xilinx BSCAN in PYNQ-Z1 board, the required configuration file

Figure 6.3: Future Technology Devices Internation Ltd logo

is listed in *Code 5.1*.

```
adapter driver ftdi
adapter speed 1000
transport select jtag

ftdi_vid_pid 0x0403 0x6010

ftdi_channel 0
ftdi_layout_init 0x0088 0x008b
ftdi_layout_signal nTRST -data 0x0080 -oe 0x0080

set _CHIPNAME riscv

jtag newtap $_CHIPNAME cpu -irlen 6 -expected-id 0x0x23727093
jtag newtap arm_dap_0 tap -irlen 4 -expected-id 0x4ba00477

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME -coreid 0x00
riscv set_ir idcode 0x09
riscv set_ir dtmcs 0x22
riscv set_ir dmi 0x23

scan_chain
```

Code 6.1: core-v-mini-mcu for PYNQ-Z1 OpenOCD cfg file

In this file, a connection stablishment is indicated, specifying the communication speed in kHz and the transport protocol (JTAG). In this case, FTDI driver is also specified, as the USB chip model is the FT2232H. This chip is used to convert UART signals from the Linux host through the USB to JTAG signals.

The next step consists of specifying the PYNQ-Z1 device PID (*Product Identification*). In Linux bash, *lsusb* command is used to find out the identification number, which is *0x0403 0x6010* in this case.

Then, initialization and configuration registers addressing for the FTDI chip is carried out. To understand which are the adequate initialization parameters, *Table 5.4* must be visualized. It was extracted from FT2232H datasheet [FTDI, 2023].

When configuring JTAG registers, output pins are selected with a '1', and input pins are selected with a '0'. When using *ftdi channel* in *Code 5.1*, two possible channels can be selected. Channel 0 must be selected, as the channel 1 has no functionality. Then, with *ftdi layout init*, the first hexadecimal parameter is used to specify the initial value of each bit inside the register. To enable JTAG operation, TMS (*Test Mode Select*) must be set to '1' as well as nTRST reset signal (*negated Test Reset*). The signals corresponding to the second hexadecimal parameter

		FT2232H	JTAG	SWD
Name	**Pin**	**Name**	**Func**	**Func**
D0	J1-3	ADBUS0	TCK	SWDCLK
D1	J1-4	ADBUS1	TDO/DI	SWDIO
D2	J1-5	ADBUS2	TDI/DO	SWDIO
D3	J1-6	ADBUS3	TMS	N/A
D4	J1-7	ADBUS4	(GPIOL0)	
D5	J1-8	ADBUS5	(GPIOL1)	
D6	J1-8	ADBUS6	(GPIOL2)	
D7	J1-9	ADBUS7	(GPIOL3)	
C0	J2-1	ACBUS0	(GPIOH0)	
C1	J2-2	ACBUS1	(GPIOH1)	
C2	J2-3	ACBUS2	(GPIOH2)	
C3	J2-4	ACBUS3	(GPIOH3)	
C4	J2-5	ACBUS4	(GPIOH4)	
C5	J2-6	ACBUS5	(GPIOH5)	
C6	J2-7	ACBUS6	(GPIOH6)	
C0	J2-8	ACBUS7	(GPIOH7)	
C8	J2-9	ACBUS8		
C9	J2-10	ACBUS9		

Table 6.3: JTAG configuration registers of FT2232H

must be configured too, following JTAG specification protocol. Finally, *ftdi layout signal* declares nTRST pin in GPIOL3.

Once basic connection parameters are configured, two TAP (*Test Access Ports*) must be created in order to distinguish between X-HEEP microcontroller (RISC-V) and the on-chip ARM Cortex A9. The ARM Coretx A9 microcontroller will not be used, but it must be declared in order to avoid OpenOCD error messages. TAPs are used to control JTAG protocol through a FSM that shifts serial data bits through the control signals (*TMS, TCK, TRST*). TAP instruction registers length (*-irlen*) is defined inside Vivado Hardware Manager, while the *-expected-id* is defined when OpenOCD performs *scan-chain*. OpenOCD uses the *scan-chain* command to detect information about the TAPs, and this information must usually be included in the configuration (*.cfg*) file.

At last, a *target* must be created to enable connection with X-HEEP. The parameter *-coreid* must be set to *0x00* in this case. Then, it is necessary to specify location codes for DTMCS and DMI, which are essential components of the JTAG protocol. Identification codes described in RISC-V specification should also be set. For the PYNQ-Z1 board, DTMCS and DMI modules are placed inside debug chains JTAG 3 and JTAG 4 (*JTAG-CHAIN*), as JTAG 1 and JTAG 2 are used for other purposes. JTAG chains are dedicated series of devices connected through JTAG ports, being controlled with a single JTAG connector. They are mainly used in boards where multiple JTAG devices are connected. In this case, considering a daisy chain of multiple JTAG devices, Test Data Input (TDI) and Test Data Output (TDO) pins are connected in series, while Test Reset (TRST) and Test Clock (TCLK) are connected in parallel [Hajimowlana, 2024].

Once the configuration file is finished, OpenOCD is executed. TCP port 3333 will be enabled automatically in localhost to accept connection requests using GDB. GDB will be the main debugging environment used to control program execution flow in X-HEEP through the Linux terminal. Programs must be compiled first, and then loaded when calling GDB. Later, break-

points are allowed to be placed during program execution flow, stopping when reached and continuing when indicated. X-HEEP only supports one breakpoint, but when it is reached, it can be removed and then placed again in another line, continuing normal execution.

7 COMPARATIVE AND RESULTS

In this section, a comparative between the two implementations described above (memory mapping and x-if implementation) will be done, considering relevant aspects such as resource utilization in the FPGA and acceleration of the NTRU algorithm. Afterwards, the extracted results from HW, in simulation, will be discussed in order to conclude which performance and resource utilization milestones have been achieved for this application.

Previously, it will be explained how the system should be mounted after RTL code development is terminated and the required file system (all the source files and directory structures) is set up. First, the bitstream must be generated, depending on the case, with the corresponding flag for enabling `Xilinx BSCAN`.

For the project of the memory map implementation with `OBI`:

```
$ make vivado-fpga FUSESOC_FLAGS=--flag=use_bscane_xilinx
```

For the project of the implementation through `X-IF`:

```
$ make synth-pynq-z2 FUSESOC_FLAGS=--flag=use_bscane_xilinx
```

These commands should be run from the `BASE` directory, which is the one where all the file system and directories structure is mounted, and where the main `Makefile` is located. Vivado will be called and the synthesis, implementation and place and route will be launched. It will take some time. When finished, the bitstream file is obtained and it can be used to program the PYNQ-Z1 through `Vivado HW Manager`. After the FPGA is already programmed, an `OpenOCD` server must be started in the Linux host machine.

The `OpenOCD` binary directory must be added to the Linux `$PATH`. When executing *auto-init-setup.sh*, this is already added.

```
$ export PATH=/home/$USER/tools/openocd/bin:$PATH
```

Then, `OpenOCD` is called, pointing to the configuration file (*Code 5.1*). The server must be left running in the background.

```
$ openocd -f /home/$USER/BASE/tb/core-v-mini-mcu-pynq-z1-cei.cfg
```

The corresponding SW application must be compiled.

```
$ make app PROJECT=polyheep_test ARCH=rv32imfc TARGET=pynq-z2
```

Afterwards, the corresponding `GDB` version for `RISC-V` shall be called. This `GDB` version is automatically installed with RISC-V GNU toolchain. It must be added to Linux `$PATH`.

```
$ export PATH=/home/$USER/tools/riscv/bin:$PATH
```

```
$ riscv32-unknown-elf-gdb sw/build/main.elf
```

The debugging interface is now executed, and it is possible to stablish connection with the `OpenOCD` target.

```
<gdb> target remote localhost:3333
```

The SW application is loaded as follows.

```
<gdb> load
```

And debugging tasks can be carried out using standard `OpenOCD` commands. It is recommended to check the user manual, which can be found in [OpenOCD-Project, 2024].

To check results and performance, it will be necessary to define a dataset first, which will be sourced as the set of input polynomial coefficients. These coefficients could be random or user-defined. The `NTRUEncrypt` reference algorithm must be computed first using the CPU only. This is done looking for the initial performance without acceleration of any kind.

The `NTRUEncrypt` algorithm is executed in X-HEEP CPU, and available HW resources will be used in order to measure execution times. In this case, X-HEEP has a `rv-timer`, which is a dedicated HW timer for `RISC-V` architecture. It can be programmed and configured by SW, using the defined functions available from the drivers. An example of the source NTRUEncrypt algorithm with the configuration and initialization of the timer is listed in *Code 7.1*.

```
...
#include "rv_timer.h"
#include "soc_ctrl.h"
...
static rv_timer_t timer_0;
static const uint64_t kTickFreqHz = 1000 * 1000; // 1 MHz
...
uint64_t cpu_ticks;
// Get current Frequency
soc_ctrl_t soc_ctrl;
soc_ctrl.base_addr = mmio_region_from_addr((uintptr_t)SOC_CTRL_START_ADDRESS);
uint32_t freq_hz = soc_ctrl_get_frequency(&soc_ctrl);
//Setup rv timer
mmio_region_t timer_0_reg = mmio_region_from_addr(RV_TIMER_AO_START_ADDRESS);
rv_timer_init(timer_0_reg, (rv_timer_config_t){.hart_count = 1, .comparator_count
    ↪ = 1}, &timer_0);
rv_timer_tick_params_t tick_params;
rv_timer_approximate_tick_params(freq_hz, kTickFreqHz, &tick_params);
rv_timer_set_tick_params(&timer_0, 0, tick_params);
...
rv_timer_counter_set_enabled(&timer_0, 0, kRvTimerEnabled);
for (k = 0; k < 401; k++) {
    mem_e_sw[k] = 0;
for (i=1; i<401-k; i++)
  mem_e_sw[k] += mem_r_sw[k+i] * mem_h_sw[401-i];
for (i=0; i<k+1; i++)
  mem_e_sw[k] += mem_r_sw[k-i] * mem_h_sw[i];
}
rv_timer_counter_set_enabled(&timer_0, 0, kRvTimerDisabled);
rv_timer_counter_read(&timer_0, 0, &cpu_ticks);
...
printf("CPU clock count : %d", cpu_ticks);
```

Code 7.1: NTRUEncrypt reference algorithm, executed by the CPU, by SW

Additionally, the `soc-ctrl` library is used to get the correspondant clock frequency of the system, depending on the case, using `soc-ctrl-get-frequency`. For the `rv-timer`, the object type is created first (`rv-timer-t`), and then the tick frequency is defined. The tick frequency is used to set up the time interval that the timer takes to increment by one the internal counter. In this case, the timer will be used as a single counter, without any interruption enabled. Consequently, when calling `rv-timer-init`, only one HW resource (*hart timer*) and only one comparator are

specified. Then, clock and tick frequency are configured using `rv-timer-set-tick-params`, and the timer is started (or stopped) using `rv-timer-set-enabled`.

Once the timer is started, the algorithm is executed until finished, and then stopped. The tick count can be read using `rv-timer-counter-read`. This value can be printed through the UART in order to measure the execution time. In *Table 7.1*, the execution time of the **EES401EP1** reference algorithm by SW (without any acceleration) is collected, with *target sim* specified.

	Tick Count	**Clock frequency**	**Tick frequency**	**Execution Time**
CPU only	1860000000	15 MHz	1 MHz	1860 s

Table 7.1: NTRUEncrypt execution time for CPU only (-Os)

It is stated that the algorithm computation is very slow when using the CPU only, as X-HEEP has a slow clock too. Due to the soft-core nature of `X-HEEP`, when embedded for a FPGA, clock speed is limited in order to meet the required timing constraints (by default, clock set to 15 MHz). Consequently, the computation power is also limited in this case.

However, the situation changes when using `POLYMUL`, even in its simplest case with only 1 ALU (M=1). Execution times depending on the selected implementation case (memory map or x-if) are collected in *Table 7.2*. These times include coefficient load, operation and memory storing in simulation.

M=1	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	55000000	15 MHz	1 MHz	55.00 s
X-IF	54000000	15 MHz	1 MHz	54.00 s
M=2	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	28500000	15 MHz	1 MHz	28.50 s
X-IF	27266666	15 MHz	1 MHz	27.27 s
M=4	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	15133333	15 MHz	1 MHz	15.13 s
X-IF	13900000	15 MHz	1 MHz	13.90 s
M=8	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	8466666	15 MHz	1 MHz	8.47 s
X-IF	7233333	15 MHz	1 MHz	7.23 s
M=16	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	5100000	15 MHz	1 MHz	5.10 s
X-IF	3866666	15 MHz	1 MHz	3.87 s
M=32	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
Memory map	3366666	15 MHz	1 MHz	3.36 s
X-IF	2133333	15 MHz	1 MHz	2.13 s
M=64	**Tick Count**	**Clock frequency**	**Tick frequency**	**Execution Time**
X-IF	1333333	15 MHz	1 MHz	1.33 s

Table 7.2: NTRUEncrypt execution times using POLYMUL, depending on M

Computation times are much more reasonable when using HW acceleration, considering that `X-HEEP` is a slow microcontroller. The accelerator seems to perform correctly, achieving a **133x boost** with **M=4** as an example. The `X-IF` implementation seems to be a little more optimized, achieving around one second less than memory map implementation, in average. However, both implementations are similar in terms of performance. In *Figure 7.1*, a comparative graph is also

provided, with the test execution times for each implementation case depending on M.

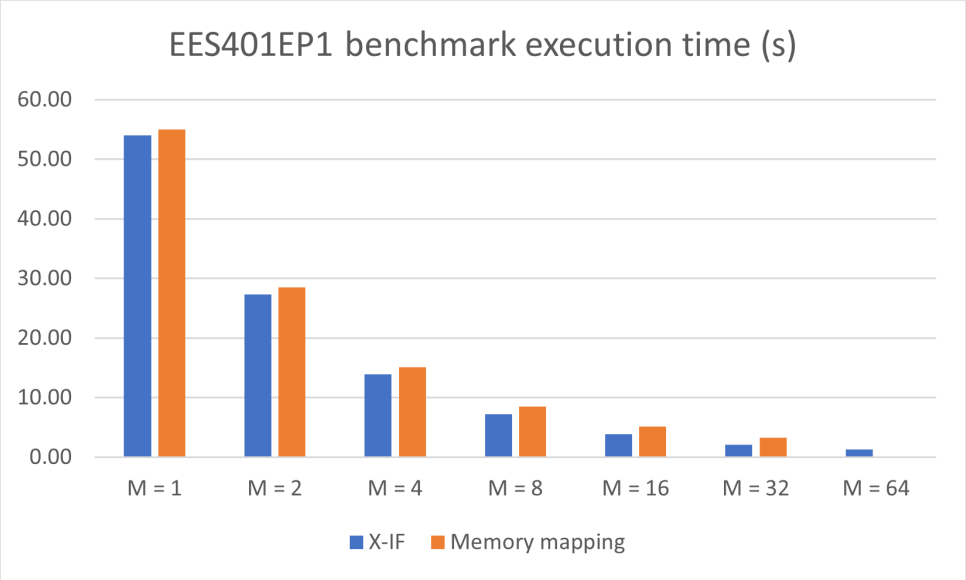

Figure 7.1: SW test benchmark execution times for each implementation case

Now, resource utilization will be discussed, considering the reports provided by `Vivado` once the correspondant bitstream is generated. Results are collected in *Table 7.3*, *Table 7.4*, depending on the number of instantiated ALUs (`M` parameter).

Site Type	**M=2**	**M=4**	**M=8**	**M=16**	**Available**
LUTs as Logic	40.26%	40.36%	40.58%	41.04%	53200
LUTs as Memory	0%	0%	0%	0%	17400
Registers as Flip-Flop	24.89%	24.89%	24.89%	24.89%	106400
Registers as Latch	<0.01%	<0.01%	<0.01%	<0.01%	106400
F7 Muxes	8.17%	8.17%	8.21%	8.17%	26600
F8 Muxes	6.51%	6.51%	6.51%	6.51%	13300
RAMB36/FIFO	0%	0%	0%	0%	140
RAMB18E1 only	1.79%	3.21%	6.07%	11.79%	280
Bonded IOB	44.00%	44.00%	44.00%	44.00%	125
BUFGCTRL	28.13%	28.13%	28.13%	28.13%	32

Table 7.3: Slice Logic, Memory, IO/GT and Clocking utilization for memory map (OBI) implementation

Generally, less than the 50% of available resources in the FPGA are used with these implementation cases. Depending on the number of ALUs instantiated in parallel, slice logic utilization incresases slightly. This analysis was registered until `M=16` because greater values of `M` are not expected to increase performance significantly, as it was stated in [Santiago Sánchez-Solano and Brox, 2022] and proven also here. When `M` parameter is greater than 16, resource utilization starts being significant, and performance boost is not very high.

Site Type	M=2	M=4	M=8	M=16	Available
LUTs as Logic	41.32%	41.62%	42.32%	43.57%	53200
LUTs as Memory	0%	0%	0%	0%	17400
Registers as Flip-Flop	25.39%	25.39%	25.39%	25.39%	106400
Registers as Latch	<0.01%	<0.01%	<0.01%	<0.01%	106400
F7 Muxes	8.02%	8.02%	8.02%	8.14%	26600
F8 Muxes	6.50%	6.50%	6.50%	6.50%	13300
RAMB36E1 only	2.86%	5.71%	11.43%	22.86%	140
RAMB18E1 only	0.36%	0.36%	0.36%	0.36%	280
Bonded IOB	47.20%	47.20%	47.20%	47.20%	125
BUFGCTRL	31.25%	31.25%	31.25%	31.25%	32

Table 7.4: Slice Logic, Memory, IO/GT and Clocking utilization for X-IF implementation

8 CONCLUSIONS

To conclude this Master thesis project, it is confirmed that open and extendable hardware platforms like `X-HEEP` have a strong potential that goes beyond its implementation possibilities, based on the powerful and up-and-coming `RISC-V ISA`. Hardware extensions and customizations for embedded microcontrollers, designed with RISC-V architecture, have multiple possibilities and applications in the real world, such as post-quantum cryptography and many other fields. HW acceleration is a significant challenge nowadys, and all these new emerging perspectives for HW architectures are providing a wide range of features and possibilities that were not considered in the past. For example, interfaces such as `X-IF`, permit extending RISC-V CPUs without changing their RTL and providing a new promising approach that could achieve an excellent performance when attaching external accelerators or coprocessors.

Moreover, it is also stated that FPGAs are a solid implementation solution nowadays for purposes like the ones pursued with this Master thesis project. Reconfiguration capabilities are essential to test and design different architecture possibilities for academic or even industrial scopes. Although FPGAs have their own limitations, such as achievable clock speeds, power consumption or HDL development complexity, they have a solid structure which is capable of providing real HW operation with a reasonable performance and with an overall cost which is eventually lower than ASICs' cost.

Finally, the obtained results for the POLYMUL IP operation with X-HEEP, especially when POLYMUL is working as a coprocessor and using `X-IF`, are actually remarkable in simulation. Acceleration boost is consistent although the clock speed is not very high due to design limitations, and execution times are potentially improvable when using HDL `parameters` configured by the users to adapt the design size or structure for the desired purpose.

BIBLIOGRAPHY

ASML (2023). *The basics of microchips - (12/09/2023)*. URL: https://www.asml.com/en/technology/all-about-microchips/microchip-basics.

California Berkeley, U. of (2023). *Chipyard GitHub Repository - (12/30/2023)*. URL: https://github.com/ucb-bar/chipyard/blob/main/tests/rocc.h.

Cong Chen Oussama Danba, J. H. et al. (2019). *NTRU Algorithm specification and supporting documentation*. URL: https://ntru.org/f/ntru-20190330.pdf.

Daniele Micciano, O. R. (2009). "Lattice-based Cryptography". In: *CSE University of California in San Diego and Tel-Aviv University*.

Digilent (2023). *Digilent Reference - PYNQ-Z1*. URL: https://digilent.com/reference/programmable-logic/pynq-z1/start.

Emily Blem, J. M. and Sankaralingam, K. (2013). "A Detailed Analysis of Contemporary ARM and x86 Architectures". In: *University of Wisconsin*.

ESL-EPFL (2023). *X-HEEP GitHub repository- (12/23/2023)*. URL: https://github.com/esl-epfl/x-heep.

FTDI (2023). *FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC - (12/25/2023)*. URL: https://ftdichip.com/wp-content/uploads/2020/07/DS_FT2232H.pdf.

Fujitsu (2024). *Fugaku specifications - Official Websitte - (01/25/2024)*. URL: https://www.fujitsu.com/global/about/innovation/fugaku/specifications/.

FuseSoC (2023). *Official Website- (12/23/2023)*. URL: https://fusesoc.net/.

Hajimowlana, H. (2024). *Architecting a Multi-Voltage JTAG Chain - (01/26/2024)*. URL: https://www.analog.com/en/analog-dialogue/articles/architecting-multi-voltage-jtag-chain.html.

Hruska, J. (2021). *RISC vs. CISC Is the Wrong Lens for Comparing Modern x86, ARM CPUs - (12/09/2023)*. URL: https://www.extremetech.com/extreme/323245-risc-vs-cisc-why-its-the-wrong-lens-to-compare-modern-x86-arm-cpus.

Maximilina Richter Magdalena Bertram, J. S. (2022). "A Mathematical Perspective on Post-Quantum Cryptography". In: *Mathematics*.

Mosca, M. (2018). "Cybersecurity in an era with quantum computers: Will we be ready?" In: *IEEE Security and Privacy* 16.5.

OpenHWGroup (2023a). *CV32E40P GitHub repository- (12/22/2023)*. URL: https://github.com/openhwgroup/cv32e40p.

– (2023b). *CV32E40X GitHub repository- (12/22/2023)*. URL: https://github.com/openhwgroup/cv32e40x.

OpenHWGroup (2023c). *OBI (Open Bus Interface) - (12/27/2023)*. URL: https://github.com/openhwgroup/obi/tree/072d9173c1f2d79471d6f2a10eae59ee387d4c6f.

– (2023d). *X-IF (eXtension Interface) - (12/28/2023)*. URL: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/x_ext.html.

OpenOCD-Project (2024). *Open On-Chip Debugger: OpenOCD User's Guide - (01/03/2024)*. URL: https://openocd.org/doc/pdf/openocd.pdf.

OpenTitan (2023). *Register Tool - (12/26/2023)*. URL: https://opentitan.org/book/util/reggen/index.html.

Pasquale Davide Schiavone Francesco Conti, D. R. et al. (2017). "Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications". In: *Conference Paper*.

Platform, P. (2023). *Official Website- (12/22/2023)*. URL: https://www.pulp-platform.org/.

Raza (2023). *Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) - (12/09/2023)*. URL: https://medium.com/@adilrk/reduced-instruction-set-computing-risc-and-complex-instruction-set-computing-cisc-f7c0cd6e7f37.

RISC-V-International (2023a). *History of RISC-V - (12/09/2023)*. URL: https://riscv.org/about/history/.

– (2023b). *RISC-V base and standard extensions - (12/21/2023)*. URL: https://gist.github.com/dominiksalvet/2a982235957012c51453139668e21fce.

– (2023c). *RISC-V ISA privileged specification - (12/21/2023)*. URL: https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view.

– (2023d). *RISC-V ISA unprivileged specification - (12/21/2023)*. URL: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

Santiago Sánchez-Solano Eros Camacho-Ruiz, M. C.-R. and Brox, P. (2022). "Multi-Unit Serial Polynomial Multiplier to Accelerate NTRU-Based Cryptographic Schemes in IoT Embedded Systems". In: *Sensors 2022*.

Verilator (2023). *Official Website- (12/23/2023)*. URL: https://www.veripool.org/verilator/.

Xuesong Zhang Luqiao Yin, K. R. and Zhang, J. (2022). "Research on Simulation Design of MOS Driver for Micro-LED". In: *Electronics*.

ANNEX

A Complementary memory mapping code listing

```
.PHONY: clean help

TARGET          ?= sim
FPGA_BOARD      ?= pynq-z2
PORT            ?= /dev/ttyUSB2

# 1 external domain for the CGRA
EXTERNAL_DOMAINS = 1

# Project options are based on the app to be build (default - hello_world)
PROJECT  ?= hello_world

export HEEP_DIR = hw/vendor/esl_epfl_x_heep/
include $(HEEP_DIR)Makefile.venv

...

vivado-fpga: |venv
  fusesoc --cores-root . run --no-export --target=$(FPGA_BOARD)
          ↪ $(FUSESOC_FLAGS) --setup --build polymul_x_heep 2>&1 | tee
          ↪ buildvivado.log


# Runs verible formating
verible:
  util/format-verible;

# Simulation
verilator-sim:
  fusesoc --cores-root . run --no-export --target=sim --tool=verilator
          ↪ $(FUSESOC_FLAGS) --setup --build polymul_x_heep 2>&1 | tee
          ↪ buildsim.log

...

XHEEP_MAKE = $(HEEP_DIR)/external.mk
include $(XHEEP_MAKE)

# Add a dependency on the existing app target of XHEEP to create a link to the
          ↪ build folder
app: link_build

clean-app: link_rm

link_build:
  ln -sf ../hw/vendor/esl_epfl_x_heep/sw/build sw/build

link_rm:
  rm sw/build

clean:
  rm -rf build buildsim.log
```

Code A.1: POLYMUL-X-HEEP (through memory mapping) Makefile

```
1
2   ...
3
4   //POLYMUL instance
5   polymul #(
6       .reg_req_t        (reg_pkg::reg_req_t),
7       .reg_rsp_t        (reg_pkg::reg_rsp_t),
8       .obi_req_t  (obi_pkg::obi_req_t),
9       .obi_resp_t (obi_pkg::obi_resp_t)
10  ) polymul_i (
11      .clk_i,
12      .rst_ni,
13      .intr_endop   (polymul_int),
14      .reg_req_i    (ext_periph_slave_req),
15      .reg_rsp_o    (ext_periph_slave_resp),
16      .master_req_i (ext_xbar_slave_req),
17      .slave_resp_o (ext_xbar_slave_resp)
18  );
19
20  ext_bus #(
21      .EXT_XBAR_NMASTER(POLYMUL_XBAR_NMASTER),
22      .EXT_XBAR_NSLAVE (POLYMUL_XBAR_NSLAVE)
23  ) ext_bus_i (
24      .clk_i                    (clk_i),
25      .rst_ni                   (rst_ni),
26      .addr_map_i               (EXT_XBAR_ADDR_RULES),
27      .default_idx_i            (POLYMUL_IDX[LOG_EXT_XBAR_NSLAVE-1:0]),
28      .heep_core_instr_req_i    (heep_core_instr_req),
29      .heep_core_instr_resp_o   (heep_core_instr_resp),
30      .heep_core_data_req_i     (heep_core_data_req),
31      .heep_core_data_resp_o    (heep_core_data_resp),
32      .heep_debug_master_req_i  (heep_debug_master_req),
33      .heep_debug_master_resp_o (heep_debug_master_resp),
34      .heep_dma_read_ch0_req_i  (heep_dma_read_ch0_req),
35      .heep_dma_read_ch0_resp_o (heep_dma_read_ch0_resp),
36      .heep_dma_write_ch0_req_i (heep_dma_write_ch0_req),
37      .heep_dma_write_ch0_resp_o(heep_dma_write_ch0_resp),
38      .heep_dma_addr_ch0_req_i  (heep_dma_addr_ch0_req),
39      .heep_dma_addr_ch0_resp_o (heep_dma_addr_ch0_resp),
40      .ext_slave_req_o          (ext_xbar_slave_req),
41      .ext_slave_resp_i         (ext_xbar_slave_resp)
42  );
43
44  ...
45
46  x_heep_system #(
47      .COREV_PULP(COREV_PULP),
48
49  ...
50
51      .ext_dma_write_ch0_req_o(heep_dma_write_ch0_req),
52      .ext_dma_write_ch0_resp_i(heep_dma_write_ch0_resp),
53      .ext_dma_read_ch0_req_o(heep_dma_read_ch0_req),
54      .ext_dma_read_ch0_resp_i(heep_dma_read_ch0_resp),
55
56  ...
```

Code A.2: OBI connections through ext-bus in polymul-x-heep-top.sv

```
package polymul_x_heep_pkg;

  import addr_map_rule_pkg::*;
  import core_v_mini_mcu_pkg::*;

  // POLYMUL master ports
  localparam POLYMUL_XBAR_NMASTER = 0;
  // One POLYMUL slave port
  localparam POLYMUL_XBAR_NSLAVE = 1;
  localparam int unsigned LOG_EXT_XBAR_NMASTER = POLYMUL_XBAR_NMASTER > 1 ? $clog2(
      POLYMUL_XBAR_NMASTER
  ) : 32'd1;
  localparam int unsigned LOG_EXT_XBAR_NSLAVE = POLYMUL_XBAR_NSLAVE > 1 ? $clog2(
      POLYMUL_XBAR_NSLAVE
  ) : 32'd1;

  //slave mmap and idx
  localparam logic [31:0] POLYMUL_START_ADDRESS = core_v_mini_mcu_pkg::EXT_SLAVE_START_ADDRESS + 32'h000000;
  localparam logic [31:0] POLYMUL_SIZE = 32'h100000;
  localparam logic [31:0] POLYMUL_END_ADDRESS = POLYMUL_START_ADDRESS + POLYMUL_SIZE;
  localparam logic [31:0] POLYMUL_IDX = 32'd0;
  localparam addr_map_rule_t [POLYMUL_XBAR_NSLAVE-1:0] EXT_XBAR_ADDR_RULES = '{
      '{idx: POLYMUL_IDX, start_addr: POLYMUL_START_ADDRESS, end_addr: POLYMUL_END_ADDRESS}
  };

  //slave encoder
  localparam EXT_SYSTEM_NPERIPHERALS = 1;
  localparam logic [31:0] POLYMUL_PERIPH_START_ADDRESS = core_v_mini_mcu_pkg::EXT_PERIPHERAL_START_ADDRESS + 32'h0000000;
  localparam logic [31:0] POLYMUL_PERIPH_SIZE = 32'h0001000;
  localparam logic [31:0] POLYMUL_PERIPH_END_ADDRESS = POLYMUL_PERIPH_START_ADDRESS + POLYMUL_PERIPH_SIZE;
  localparam logic [31:0] POLYMUL_PERIPH_IDX = 32'd0;
  localparam addr_map_rule_t [EXT_SYSTEM_NPERIPHERALS-1:0] EXT_PERIPHERALS_ADDR_RULES = '{
      '{
          idx: POLYMUL_PERIPH_IDX,
          start_addr: POLYMUL_PERIPH_START_ADDRESS,
          end_addr: POLYMUL_PERIPH_END_ADDRESS
      }
  };
  localparam int unsigned EXT_PERIPHERALS_PORT_SEL_WIDTH = EXT_SYSTEM_NPERIPHERALS > 1 ? $clog2(
      EXT_SYSTEM_NPERIPHERALS
  ) : 32'd1;

endpackage  // polymul_x_heep_pkg
```

Code A.3: polymul-x-heep-pkg.sv file listing (Memory mapping)

```
#ifndef POLYMUL_X_HEEP_H_
#define POLYMUL_X_HEEP_H_

#ifdef __cplusplus
extern "C" {
#endif  // __cplusplus

#include "core_v_mini_mcu.h"

#define EXT_XBAR_NMASTER 4
#define EXT_XBAR_NSLAVE  1

#define POLYMUL_START_ADDRESS (EXT_SLAVE_START_ADDRESS + 0x000000)
#define POLYMUL_SIZE 0x100000
#define POLYMUL_END_ADDRESS (POLYMUL_START_ADDRESS + POLYMUL_SIZE)

#define POLYMUL_PERIPH_START_ADDRESS (EXT_PERIPHERAL_START_ADDRESS + 0x0000000)
#define POLYMUL_PERIPH_SIZE 0x0001000
#define POLYMUL_PERIPH_END_ADDRESS (POLYMUL_PERIPH_START_ADDRESS +
    ↪ POLYMUL_PERIPH_SIZE)

#ifdef __cplusplus
}  // extern "C"
#endif  // __cplusplus

#endif  // POLYMUL_X_HEEP_H_
```

Code A.4: polymul-x-heep.h file listing (Memory mapping)

```
#ifndef _POLYMUL_REG_DEFS_
#define _POLYMUL_REG_DEFS_
#ifdef __cplusplus
extern "C" {
#endif
// Register width
#define POLYMUL_PARAM_REG_WIDTH 32
// Control pin
#define POLYMUL_CTRL_REG_OFFSET 0x0
// Reset pin
#define POLYMUL_RST_REG_OFFSET 0x4
// End operation pin
#define POLYMUL_END_OP_REG_OFFSET 0x8
#define POLYMUL_END_OP_END_OP_BIT 0
#ifdef __cplusplus
}  // extern "C"
#endif
#endif  // _POLYMUL_REG_DEFS_
```

Code A.5: polymul-regs.h file listing (Memory mapping)

```c
#ifndef _POLYMUL_H_
#define _POLYMUL_H_

#include <stdint.h>

#include "mmio.h"
#include "core_v_mini_mcu.h"

#define POLYMUL_INTR EXT_INTR_0

#ifdef __cplusplus
extern "C" {
#endif

/**
 * POLYMUL memory initialization
 */
void polymul_mem_init(void);

/**
 * Initialization parameters for POLYMUL peripheral control registers..
 *
 */
typedef struct polymul {
  /**
   * The base address for hardware registers.
   */
  mmio_region_t base_addr;
} polymul_t;

#ifdef __cplusplus
}
#endif

#endif // _POLYMUL_H_
```

Code A.6: polymul.h file listing (Memory mapping)

B X-IF implementation complementary files listing

```
module polymul_ss_predecoder #(
    parameter int NumInstr = 1,
    parameter polymul_ss_pkg::offload_instr_t OffloadInstr[NumInstr] =
        ↪ {0}
) (
    input  polymul_ss_pkg::acc_prd_req_t prd_req_i,
    output polymul_ss_pkg::acc_prd_rsp_t prd_rsp_o
);
  import polymul_ss_pkg::*;
  acc_prd_rsp_t [NumInstr-1:0] instr_rsp;
  logic         [NumInstr-1:0] instr_sel;
  for (genvar i = 0; i < NumInstr; i++) begin : gen_predecoder_selector
    assign instr_sel[i] =
      ((OffloadInstr[i].instr_mask & prd_req_i.q_instr_data) ==
        ↪ OffloadInstr[i].instr_data);
  end
  for (genvar i = 0; i < NumInstr; i++) begin : gen_predecoder_mux
    assign instr_rsp[i].p_accept    = instr_sel[i] ?
        ↪ OffloadInstr[i].prd_rsp.p_accept : 1'b0;
    assign instr_rsp[i].p_writeback = instr_sel[i] ?
        ↪ OffloadInstr[i].prd_rsp.p_writeback : 1'b0;
    assign instr_rsp[i].p_is_mem_op = instr_sel[i] ?
        ↪ OffloadInstr[i].prd_rsp.p_is_mem_op : '0;
  end
  always_comb begin
    prd_rsp_o.p_accept    = 1'b0;
    prd_rsp_o.p_writeback = 1'b0;
    prd_rsp_o.p_is_mem_op = '0;
    for (int unsigned i = 0; i < NumInstr; i++) begin
      prd_rsp_o.p_accept    |= instr_rsp[i].p_accept;
      prd_rsp_o.p_writeback |= instr_rsp[i].p_writeback;
      prd_rsp_o.p_is_mem_op |= instr_rsp[i].p_is_mem_op;
    end
  end
endmodule // polymul_ss_predecoder
```

Code B.7: polymul-ss predecoder file listing (X-IF implementation)

```
package polymul_ss_prd_pkg;
parameter int unsigned NumInstr = 3;
parameter polymul_ss_pkg::offload_instr_t OffloadInstr[NumInstr] = '{
    '{
        instr_data: 32'b 0000000_00000_00000_000_00000_1110111, // X_MUL
        instr_mask: 32'b 0000000_00000_00000_000_00000_1111111,
        prd_rsp : '{
            p_accept : 1'b1,
            p_writeback : 1'b0,
            p_is_mem_op : 1'b0
        }
    },
    '{
        instr_data : 32'b 0000000_00000_00000_101_00000_1111011, //
        ↪ X_STR
        instr_mask : 32'b 0000000_00000_00000_111_00000_1111111,
        prd_rsp : '{
            p_accept : 1'b1,
            p_writeback : 1'b0,
            p_is_mem_op : 1'b1
        }
    },
    '{
        instr_data : 32'b 0000000_00000_00000_101_00000_1100111, //
        ↪ X_CLD
        instr_mask : 32'b 0000000_00000_00000_111_00000_1111111,
        prd_rsp : '{
            p_accept : 1'b1,
            p_writeback : 1'b0,
            p_is_mem_op : 1'b1
        }
    }
};
endpackage // polymul_ss_prd_pkg
```

Code B.8: polymul-ss-prd-pkg file listing (X-IF implementation)

```
package polymul_ss_pkg;
  parameter int          X_NUM_RS    = 3;
  parameter int          X_ID_WIDTH  = 4;
  parameter int          X_MEM_WIDTH = 32;
  parameter int          X_RFR_WIDTH = 32;
  parameter int          X_RFW_WIDTH = 32;
  parameter logic [31:0] X_MISA      = '0;
  parameter logic [ 1:0] X_ECS_XS    = '0;
  localparam int XLEN = 32;
  localparam int NUM_LD_COEFS = 401 ; //Number of loaded coefficients
        ↪ in polymul per instruction
  ...
```

Code B.9: polymul-ss-pkg parameter definition (X-IF Implementation)

```
...
typedef struct packed {
  logic [31:0] q_instr_data;
} acc_prd_req_t;

// Predecoder response type
typedef struct packed {
  logic       p_accept;
  logic       p_is_mem_op;
  logic       p_writeback;
} acc_prd_rsp_t;

// Predecoder internal instruction metadata
typedef struct packed {
  logic [31:0]  instr_data;
  logic [31:0]  instr_mask;
  acc_prd_rsp_t prd_rsp;
} offload_instr_t;

typedef struct packed {
  logic [31:0]      instr_data;
  logic [3:0]       id;
  logic [1:0]       mode;
} offloaded_data_t;

typedef struct packed {
  logic [2:0] reg_sel;
  logic       is_launch_op;
  logic [1:0] is_ldst; // 00 : neither load nor store | 01 : load |
      ↪ 10: store
} decoded_op_t;
...
```

Code B.10: polymul-ss-pkg custom data types definition (X-IF Implementation)

```
...
typedef struct packed {
    logic [          31:0]                    instr;
    logic [           1:0]                    mode;
    logic [X_ID_WIDTH-1:0]                    id;
    logic [X_NUM_RS  -1:0][X_RFR_WIDTH-1:0]   rs;
    logic [X_NUM_RS  -1:0]                    rs_valid;
    logic [           5:0]                    ecs;
    logic                                     ecs_valid;
  } x_issue_req_t;
  typedef struct packed {
    logic       accept;
    logic       writeback;
    logic       dualwrite;
    logic [2:0] dualread;
    logic       loadstore;
    logic       ecswrite ;
    logic       exc;
  } x_issue_resp_t;
  typedef struct packed {
    logic [X_ID_WIDTH-1:0] id;
    logic                  commit_kill;
  } x_commit_t;
  typedef struct packed {
    logic [X_ID_WIDTH   -1:0] id;
    logic [             31:0] addr;
    logic [              1:0] mode;
    logic                     we;
    logic [              2:0] size;
    logic [X_MEM_WIDTH/8-1:0] be;
    logic [              1:0] attr;
    logic [X_MEM_WIDTH  -1:0] wdata;
    logic                     last;
    logic                     spec;
  } x_mem_req_t;
  typedef struct packed {
    logic       exc;
    logic [5:0] exccode;
    logic       dbg;
  } x_mem_resp_t;
...
```

Code B.11: polymul-ss-pkg x-if data types definition (1)

```
...
  typedef struct packed {
    logic [X_ID_WIDTH -1:0] id;
    logic [X_MEM_WIDTH-1:0] rdata;
    logic                   err;
    logic                   dbg;
  } x_mem_result_t;

  typedef struct packed {
    logic [X_ID_WIDTH      -1:0] id;
    logic [X_RFW_WIDTH     -1:0] data;
    logic [                 4:0] rd;
    logic [X_RFW_WIDTH/XLEN-1:0] we;
    logic [                 5:0] ecsdata;
    logic [                 2:0] ecswe;
    logic                        exc;
    logic [                 5:0] exccode;
    logic                        err;
    logic                        dbg;
  } x_result_t;
...
```

Code B.12: polymul-ss-pkg x-if data types definition (2)

```
...
logic [X_RFR_WIDTH-1:0] rs1_operand, rs2_operand;
logic rs1_valid, rs2_valid;
logic mem_sel;
logic mem_loading, mem_storing; //For signaling if polymul memory is
        ↪ being loaded
logic [X_RFR_WIDTH-1:0] addr_cnt, addr_cnt_polymul;
logic finish, finish_polymul, dec_launch;
logic [X_RFR_WIDTH-1:0] offset; //Address offset
logic ctrl_valid, finish_polymul_q, x_result_valid_q,
        ↪ x_mem_result_valid_q;
logic [X_ID_WIDTH-1:0] x_result_o_id;
logic [1:0] dec_loadstore;
logic accept_q;
...
//Offset assignment
assign offset [X_RFR_WIDTH-1:12] = '0;
assign offset [11:5]  = x_issue_req_i.instr[31:25];
assign offset [4:0]   = x_issue_req_i.instr[11:7];
//x_issue response signals assignment
assign prd_req.q_instr_data      = x_issue_req_i.instr;
assign x_issue_resp_o.accept     = prd_rsp.p_accept;
assign x_issue_resp_o.writeback  = prd_rsp.p_writeback;
assign x_issue_resp_o.dualwrite  = 1'b0;
assign x_issue_resp_o.dualread   = '0;
assign x_issue_resp_o.loadstore  = prd_rsp.p_is_mem_op;
assign x_issue_resp_o.ecswrite   = 1'b0;
assign x_issue_resp_o.exc        = 1'b0;
...
```

Code B.13: polymul-ss-controller offset and issue response interface assignments (X-IF implementation)

```
module polymul_ss_controller
    import polymul_ss_pkg::*;
#(
    parameter M = 4,
    parameter BRAM_DATA_BUS_WIDTH = 32,
    parameter BRAM_DATA_BUS_WIDTH_R = 2,
    parameter BRAM_ADDR_BUS_WIDTH = 9,
    parameter C_MEM_DEPTH = 401,
    parameter MEMORY_SELECT_POS = 10
) (
    //================ Clock and Reset ====================
    input logic clk_i,
    input logic rst_ni,
    //===================== X-IF ==========================
    // Compressed Interface
    input  logic x_compressed_valid_i,
    output logic x_compressed_ready_o,
    input  x_compressed_req_t x_compressed_req_i,
    output x_compressed_resp_t x_compressed_resp_o,
    // Issue Interface
    input  logic x_issue_valid_i,
    output logic x_issue_ready_o,
    input  x_issue_req_t x_issue_req_i,
    output x_issue_resp_t x_issue_resp_o,
    // Commit Interface
    input  logic x_commit_valid_i,
    input  x_commit_t x_commit_i,
    // Memory Request/Response Interface
    output logic x_mem_valid_o,
    input  logic x_mem_ready_i,
    output x_mem_req_t x_mem_req_o,
    input  x_mem_resp_t x_mem_resp_i,
    // Memory Result Interface
    input  logic x_mem_result_valid_i,
    input  x_mem_result_t x_mem_result_i,
    // Result Interface
    output logic x_result_valid_o,
    input  logic x_result_ready_i,
    output x_result_t x_result_o,
    //=========== polymul_ss_predecoder ports =============
    output polymul_ss_pkg::acc_prd_req_t prd_req,
    input  polymul_ss_pkg::acc_prd_rsp_t prd_rsp,
    //=========== polymul_ss_decoder ports ================
    output logic [X_RFR_WIDTH-1:0]     instr_data,
    input polymul_ss_pkg::decoded_op_t decoded_op,
...
```

Code B.14: polymul-ss-controller ports definition (1) (X-IF implementation)

```
...
//============== Poly_Mul ports ======================
//Control ports
output logic clk_polymul, rst_polymul, ctrl,
input  end_op,
//MEM-R ports
output logic bram_clk_r, bram_en_r, bram_we_r, bram_rst_r,
output logic [BRAM_ADDR_BUS_WIDTH-1:0] bram_addr_r,
output logic [BRAM_DATA_BUS_WIDTH_R-1:0] bram_data_in_r,
input  [BRAM_DATA_BUS_WIDTH_R-1:0] bram_data_out_r,
//MEM-H ports
output logic bram_clk_h, bram_en_h, bram_we_h, bram_rst_h,
output logic [BRAM_ADDR_BUS_WIDTH-1:0] bram_addr_h,
output logic [BRAM_DATA_BUS_WIDTH-1:0] bram_data_in_h,
input  [BRAM_DATA_BUS_WIDTH-1:0] bram_data_out_h,
//MEM-E ports
output logic bram_clk_e, bram_en_e, bram_we_e, bram_rst_e,
output logic [BRAM_ADDR_BUS_WIDTH-1:0] bram_addr_e,
output logic [BRAM_DATA_BUS_WIDTH-1:0] bram_data_in_e,
input  [BRAM_DATA_BUS_WIDTH-1:0] bram_data_out_e
);
...
```

Code B.15: polymul-ss-controller ports definition (2) (X-IF implementation)