Robert Margelli

System-level Design of a Latency-insensitive RISC-V Microprocessor and Optimization via High-level Synthesis

Master's Thesis

Department of Control and Computer Engineering (DAUIN)
Politecnico di Torino

Supervision

Prof. Luciano Lavagno
Prof. Luca Carloni (Columbia University)
October 2017

Contents

Acknowledgements **vii**

Abstract **ix**

Acronyms **xi**

1 Introduction **1**

1.1 Challenges and Contribution 1

1.2 Thesis Organization 1

2 Background **3**

2.1 Microprocessors 3

2.1.1 History and Market Trends 3

2.1.2 Structure: Datapath and Control Unit 5

2.1.3 Performance Metrics 7

2.1.4 Pipelining 8

2.2 The RISC-V Instruction Set Architecture 10

2.3 System-level Design and High-level Synthesis 13

2.3.1 The SystemC Class Library 15

2.3.2 The Theory of Latency Insensitive Design 16

3 RVXRed: A System-Level Microprocessor **17**

3.1 From SystemC to Verilog RTL 17

3.2 LICs: Latency-Insensitive Channels 21

3.3 An HLS Approach to Microprocessor Design 23

3.3.1 Architecture 25

3.3.2 Fedec 26

3.3.3 Execute 30

3.3.4 Memwb 34

4 Experimental Setup **37**

4.1 Logic Simulation and Synthesis 40

4.2 FPGA Verification 44

4.3 Test Programs 50

4.4 CPU Time as a Performance Metric 52

5 Evaluation and Results **53**

5.1 FPGA Implementation 53

5.2 CMOS Implementation 54

5.3 Qualitative Results: Lines of Code 56

6 Conclusion **59**

6.1 Achievements 59

6.2 Future Works 59

A RVXRed Instruction Set **61**

List of Figures

2.1 Basic CPU Structure 6
2.2 Example of instructions flowing through a pipeline. 8
2.3 RV32I R-type and S-type instruction encodings. 12
2.4 Commonly used languages for hardware design. 13
2.5 Design Space Exploration [1]. 14
2.6 Example of Pareto curves in the performance-area space. 14

3.1 FSM resulting from the example SystemC module. 20
3.2 Signal-level LIC handshaking protocol. 22
3.3 CDFG of the put method. 23
3.4 CDFG of the get method. 23
3.5 High-level architecture of the RVXRed pipeline. 26

4.1 Experimental setup design flow. 39
4.2 RVXRed memory adapters. 45
4.3 Zero-riscy memory adapters. 45
4.4 Zero-riscy's request-grant memory protocol for a read transaction. 46
4.5 Top-level wrapper module. 46
4.6 Complete FPGA IP block system. 48

5.1 CPU Time vs Area plots 57

List of Tables

2.1 RISC-V base and extension instruction sets. 11
2.2 RISC-V general purpose registers coding conventions. 11

5.1 FPGA clock frequency and resource utilizations 54
5.2 CMOS clock frequency and area occupation 55
5.3 Division characteristics for all implementations. 55
5.4 Comparing LOC, considering the manually written SystemC code for the RVXRed versions. 56
5.5 Comparing LOC, considering the automatically generated Verilog RTL code for the RVXRed versions. 56

A.1 RV32I instruction subset 61
A.2 RV32M instruction subset 61

Acknowledgements

I would like to thank my supervisors Luca Carloni and Luciano Lavagno for the opportunity of conducting my research work on a topic I am passionate about. They have always been present and supported me throughout this journey.
I would also like to thank Paolo Mantovani for valuable discussions on design decisions and guidelines on writing this thesis, as well as Giuseppe Di Guglielmo for his lessons on high-level synthesis and Stratus HLS.
Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study.

Abstract

In recent years, the crisis of technology scaling has forced the semiconductor industry to embrace new technologies and innovative strategies in order to respect the timing of the design cycle. A first step has been the employment of multi-core architectures to exploit the parallelism inherent to computer programs. However, this approach has been proven to be insufficient to cope with a market that demands energy and power-efficient systems. This has led to the adoption of System-on-chip (SoC) devices, which are ubiquitous in electronic devices such as smart-phones or tablets. To this day, these integrated circuits are comprised of an heterogeneous set of sub-systems including more classical components such as microprocessors, memory blocks and input/output (I/O) peripherals, as well as dedicated units known as hardware accelerators in charge of performing in hardware tasks that were normally assigned to software programs. These units are for example: audio or image processors, video encoders and decoders, just to name a few. Along with SoCs, new design approaches have been introduced and put into practice to tackle with the intrinsic diversity of these devices. Traditional design processes focused on implementing and optimizing single components, for example the aforementioned accelerator units. In contrast, given the heterogeneous nature of these new devices it is necessary to shift to a higher level of abstraction capable of taking into account diverse sub-components and their inter-dependencies. In addition, this enables a faster exploration of the architecture of a SoC in order to find its optimal configuration.

System-level design (SLD) methodologies adopt high-level languages (such as C or C++) to easily describe large SoCs from a higher perspective as opposed to using long-established register-transfer level (RTL) languages (Verilog and VHDL) on single components. Designers have started to utilize SLD tools such as those leveraging high-level synthesis (HLS), which generate an RTL description starting from a high-level one, drastically reducing the design cycle time and enabling engineers to meet market demands. This has yielded interesting results for data-dominated applications, however, there hasn't been significant research on the application of HLS to implement microprocessors, which are in large por-

tion control-dominated circuits.

The central processing unit (CPU) is the hardware component in charge of executing the instructions of a computer program. It decodes the instructions and performs arithmetic, logical, and input/output operations.

This work proposes a new methodology for system-level microprocessor design which has been put into practice to generate several versions of RVXRed, a 32-bit 5 stage pipeline core supporting the RV32I and RV32M instruction sub-sets of the RISC-V instruction set architecture (ISA). Starting from a single SystemC description (a C++ library apt for hardware design), several RTL descriptions were generated automatically with the use of a commercial HLS tool. In total, 4 implementations were obtained: BASIC (a reference design which does not include the application of particular HLS knobs), ASAP (a faster version of BASIC, at the expense of larger area occupation), UNDIV2 (a version where the CPU's divider has been optimized to complete in 16 clock cycles) and UNDIV4 (as in UNDIV2, but with a divider latency of 8 clock cycles).

To test each core, an experimental setup was adopted to perform: logic simulation, logic synthesis and FPGA deployment for rapid prototyping.

In addition, this report includes implementation results, which have been compared with a rival solution manually designed as Verilog RTL code by an academic research group focused on developing RISC-V chips. The processors have been compared based on static indicators such as area occupancy and achievable clock frequency, as well as program-dependent values. Three benchmark programs were devised and then executed by all implementations in order to determine the time required for executing them. The outcome of these comparisons clearly reveals that the proposed approach yields significant results and clears the path for future developments which will adopt this methodology.

Acronyms

SoC	System-on-Chip
I/O	Input/Output
SLD	System-level Design
RTL	Register-transfer level
VHDL	VHSIC Hardware Description Language
HLS	High-Level Synthesis
CPU	Central Processing Unit
ISA	Instruction Set Architecture
FPGA	Field-Programmable Gate Array
SPEC	Standard Performance Evaluation Corporation
LID	Latency-insensitive Design
ALU	Arithmetic Logic Unit
CMOS	Complementary Metal-Oxide-Semiconductor
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computing
CPI	Clock Per Instruction
CDFG	Control and Data Flow Graph
DUT	Device Under Test
CLB	Configurable Logic Block
LUT	Lookup Table
CC	Clock Cycle
LOC	Lines Of Code

Chapter 1

Introduction

1.1 Challenges and Contribution

The main goal of my research work was to implement a RISC-V compatible microprocessor in a high-level programming language, and to explore the advantages and drawbacks of resorting to an HLS tool to obtain multiple RTL implementations starting from a single reference design.

Over the years, very little research regarding the high-level synthesis of microprocessors has been produced. Some studies only concerned the use of SystemC as a means to describe and simulate these systems [2], while others introduced the possibility of resorting to HLS but did not elaborate on the quality of the developed processors [3]. Moreover, no work has ever focused on a rigorous experimental procedure to extrapolate metrics used for the evaluation of the generated circuits.

A first challenge was finding a way to describe the processor pipeline using SystemC, in such a way that the generated Verilog description behaved in accordance to the specifications. Next, the adopted tool was analyzed and exploited to understand where and how design space exploration was applicable. This led to four different implementations which, in addition to a reference processor, were employed in an experimental setup which included logic simulation, logic synthesis and FPGA verification. Finally, I was able to obtain indicators of the quality of each implementation, namely area occupation and effective latency.

1.2 Thesis Organization

This report is structured as follows.

Chapter 2 exposes the reader to background information and the motivating factors of my research activity. After an introduction to microprocessors and the RISC-V ISA, some details on the topic of system-level design are uncovered.

Chapter 3 represents the main intellectual contribution of my work. In Section 3.1, necessary information on the generation of Verilog starting from SystemC source code is covered, before delving into the proposed methodology and architecture for high-level microprocessor design.

Chapter 4 covers the design flow and experimental setup followed in order to obtain performance indicators and results. These informations are used to draw conclusions and compare my design with its rival implementation (**Chapter 5**).

Finally, **Chapter 6** summarizes the contributions related to my work and gives recommendations for future work.

Chapter 2

Background

2.1 Microprocessors

2.1.1 History and Market Trends

The history of the microprocessor is tied to some pivotal discoveries and contributions that originated in the 20th century. One of the first steps in this evolution has been the invention of the bipolar transistor by Bardeen, Brattain and Shockley at the Bell Laboratories in 1949. Nine years later, the first integrated circuit (IC), developed by Robert Noyce of Fairchild Semiconductor and Jack Kilby of Texas Instruments, was demonstrated.

Many families of circuits were then introduced but one of the turning points in the microelectronic revolution is due to the Metal Oxide Semicondutor (MOS), which replaced the use of Bipolar Junction Transistor (BJT) in microelectronic devices and led to the Complementary Metal-Oxide-Semiconductor (CMOS) technology for silicon-based devices. The problem with the BJT fabrication process was that the metal gate implied a slow switching and an unreliable metal-oxide-semiconductor contact.

In 1968, Federico Faggin introduced the silicon gate technology: now the gate was made of polycrystalline silicon, much more conductive than silicon, and thus enabled a faster switching of the transistors. Soon after, while at Intel, Faggin exploited this new technology and defined a methodology for integrating in a unique circuit the first microprocessor architecture. This was the 4004 chip produced in 1971. With a die area of 3 x 4 mm, including 2300 transistors with a 10 μm technology and a 4-bit architecture, it was able to run at 100 KHz.

In the following decades processor architectures grew rapidly alongside technology evolution, as predicted by the renowned *Moore's law*. This observation made in 1965 by Gordon Moore stated that the integration of transistors in integrated circuits would double every

year [4] [5] [6]. Ten years later he revised the law, re-formulating that the integration complexity would grow every to two years. The trend has confirmed such predictions. Today, processors run at about 3GHz, include more that 100 million transistors and are fabricated with technologies below 20 nm.

The key point that allowed this performance improvement was raising the level of abstraction of processor design. When first microprocessors were designed, the whole circuit was drafted directly at the layout level. Specialist knew everything about their processor, from the ISA to the final transistor layout. At this point, processors were described as hierarchical blocks using RTL languages, which abstract many low level details and the engineering teams were much larger, with personnel dedicated to jobs such as verification and testing.

In the early 2000's Moore's law still seemed to be valid, however, today we find ourselves at a possible stall and two directions can be followed. The first, known as *more Moore*, suggests a change in the design process, requiring deep sub-micron considerations even when designing at the system and register transfer levels. Vice-versa, at the physical design step, system-level implications must be evaluated. The second, *more than Moore*, implies the adoption of new advancements in the production process while leaving CMOS behind. Examples are devices that rely on carbon nanotubes, quantum dot cellular automata or molecular electronics.

Not concerning directly the evolution of the microprocessor from a technology standpoint but still part of its history, it is worth spending a few words on Reduced Instruction Set Computers (RISC). This architecture is the groundwork of the ISA upon which my processor is founded on.

To this day, RISC has been widespread with the advent of smartphones and tablets which leverage the ARM ISA, a family of proprietary architecture based on the RISC concepts. Among others, notable examples of ISAs that are derived from RISC are: MIPS, Blackfin, SPARC and PowerPC. As opposed to Complex Instruction Set Computing (CISC), the first RISC designers aimed at reducing the clock cycles per instruction (CPI) to the value of 1 by significantly simplifying the ISA's characteristics, such as:

- Utilizing a small and simple set of instructions;
- Pipelining: an implementation technique that allows various instructions to be executed in parallel (Section 2.1.4);
- Introducing a large number of fixed-length general purpose registers which prevent costly interactions with memory;

- Load/Store Instructions: memory accesses occur only when explicitly requested by special instructions and not by any instructions;
- Encoding all instructions on a fixed number of bits.

This entailed shifting the complexity from hardware to software, specifically onto the compilers.
From an analytical viewpoint consider:

$$CPU\ Time = CP * avg\_CPI * NCI \tag{2.1}$$

Where:

- CP (clock period): function of the technology and depth of the pipeline;
- avg_CPI (average clock cycles per instruction): related to the micro-architecture and ISA;
- NCI (number of committed instructions): depends on the program and input data, this represents the number of instructions that are executed by the processor.

By lowering the avg_CPI, and allowing the NCI to slightly increase, the program execution time was successfully reduced. The increase in the NCI is due to utilizing simple instructions and thus requiring many to describe a functionality that would otherwise be coded in just a few lines using a CISC ISA.
Today, it is hard to identify an ISA as purely RISC or CISC. The line between the two has blurred over the years and each school of thought has embraced concepts from one another. Most commonly they have become labels used for marketing purposes. However, the designers of the RISC-V ISA have maintained most, if not all, of the RISC features.

2.1.2 Structure: Datapath and Control Unit

To familiarize with the structure of a CPU, Figure 2.1 illustrates a block diagram resembling its basic units.
The control unit and the datapath are the main components. The former is in charge of controlling the behavior of the latter based on which instruction must be executed. In this scenario, the data and instruction memories are not part of the core itself and can be accessed through dedicated busses. In some units instead, the processor includes caches for both memories to increase performance by lowering the latency of their accesses.
The data path contains, among others:

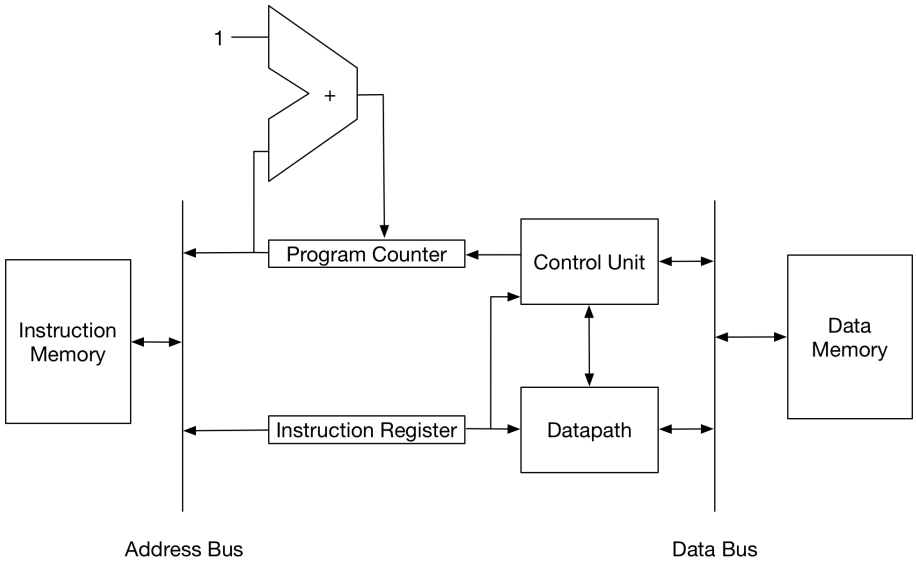


Figure 2.1: Basic CPU Structure

- Register file: a bank of registers which are used to store information used during program execution;
- Arithmetic and Logic Unit (ALU): circuitry in charge of performing arithmetic and logic operations on the operands of an instruction.

The flow of information is from left to right and starts with the fetching of the data contained in the instruction memory, addressed with the contents of the program counter (PC). The value contained in this register is incremented at each clock cycle, this enables stepping through the program to be executed. In a real scenario, programs are not fully sequential as jumps and branches to other code sections commonly take place. Here, this feature is omitted for simplicity but in practical terms this means multiplexing the input of the PC with either the incremented value or the target address of the jump or branch instruction. The obtained data is written into the instruction register and can now be decoded. Generally an instruction encodes: the operation to be performed, the input operands and the destination location (either in register file or in data memory). Such information is used by the control unit to generate the *control word*, which manages the components of the data path following the decoding logic, in particular for instructing the ALU which operation must be performed on the operands. Once the ALU is finished its computations, there may be an access to data memory, either for reading or writing, and finally a *writeback* to the register file, in the case that the destination of the operation was a register.

2.1.3 Performance Metrics

The performance of a computing system is a function of all of its components and their interdependencies. Key metrics may be reported for the system as a whole or on specific components such as the CPU alone. Traditionally, the two main measures of interest are execution time and throughput, which are reciprocal values. The former is simply the elapsed time from the start to the end of the execution of a generic instruction, while the latter is the rate at which results can be processed by the system. Increasing performance is not an immediate task. Today, computer based systems have found application in a variety of fields and for such reason new metrics should be taken into consideration. For example, in mobile embedded systems, those that are running within a limited power budget, power consumption is an important factor for evaluating their quality and must be kept to a minimum. In these cases, increasing the computational resources is not always the best viable option for enhancing performance.

In multiprogramming, a CPU that is waiting for an I/O operation to be performed switches to execute another program. The factor here reported as CPU time (or effective latency) acknowledges this distinction by definition as it represents the time since the CPU has started executing a program, excluding the intervals in which it is waiting for I/O or while running other programs. Clearly the response time seen by the user is the elapsed time of the program, not the CPU time. CPU time can be further divided into the time spent executing the program, called user CPU time, and that dedicated to the operating system performing tasks requested by the program, known as system CPU time. In Section 4.4, an analytical explanation of CPU time is given. This measure is relevant when drawing results and comparing different implementations (Chapter 5).

Ideally, to measure performance the computer should be let running programs by a differentiated set of users over a long period of time. This is not the real case. Instead, companies and researchers resort to benchmark suites. These are collections of programs which aim at stressing specific units and features of the system. The Standard Performance Evaluation Corporation (SPEC) is a non-profit organization which has been producing, maintaining and releasing collections of benchmarks, which have become the most widely adopted programs for evaluating new designs and come in a variety of programming languages (C, C++, Java, and others). The SPEC CPU set is used for testing CPU performance by measuring the effective latency of running several programs such as the Perl interpreter, video compression, route planning, just to name a few. Instead of resorting to the SPEC suite, three commonly used programs (Section 4.3) have been written and used to benchmark the considered CPU cores.

2.1.4 Pipelining

Probably the most common technique for improving CPU performance, pipelining enables multiple instructions to be executed concurrently across the pipeline stages. The datapath is split into separate units, divided by pipeline registers and, at each clock cycle, the stages work on different instructions in a parallel fashion (Figure 2.2).

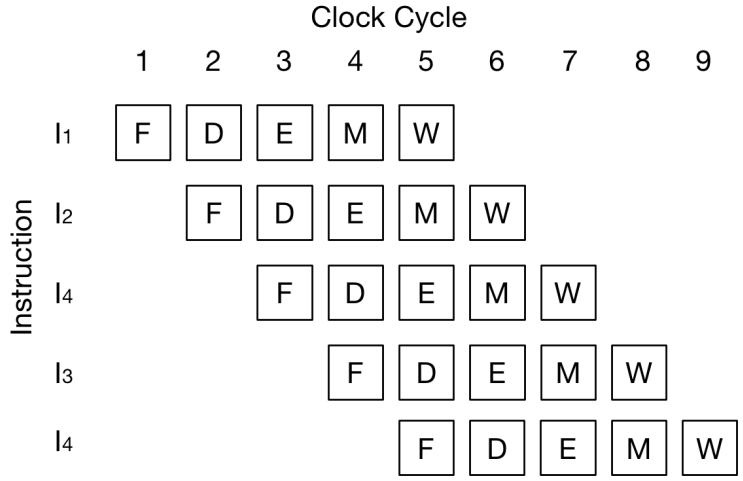


Figure 2.2: Example of instructions flowing through a pipeline.

Once the pipeline has been *filled* with instructions, it is able to complete the execution of an instruction at each clock cycle. The CPU throughput benefits enormously from this technique at the cost of a very low area overhead. In fact, the pipeline is implemented by inserting registers among the stages. This placement gives optimal results when the stages' critical paths are balanced, in fact the slowest stage determines the clock period. This value dictates the time for executing one step in the pipeline. Generally, clock cycle per instruction (CPI) is a common metric to evaluate pipelines.
The throughput of an ideal pipeline (that is, one with perfectly balanced stages) is:

$$throughput(pipelined) = throughput(un - pipelined) * n \tag{2.2}$$

where n is the number of stages. It should be clear that pipelining increases throughput but does not decrease instruction execution time. In fact, the processing of an instruction is slower due to the overhead introduced by the pipeline implementation. Still, it is negligible with respect to the increase in throughput. The execution of an instruction depends on the architecture but in simple solutions can be decomposed into five cycles [7] .
Traditionally, the instruction is processed through the following steps:

- Instruction fetch cycle (IF): the instruction is read from the instruction memory;
- Instruction decode/register fetch cycle (ID): decodes the instruction and accesses the register file;
- Execution/effective address cycle (EX): computations on the supplied operands occur here;
- Memory access/branch completion cycle (MEM): access to data memory, if required;
- Write-back cycle (WB): sends the values to be written in the register file.

To have an idea of the advantage in throughput of a pipelined architecture, consider the following. Let's first assume that:

1. We have an un-pipelined datapath with clock period of 4 ns;
2. Operations that read from data memory require 4 cycles to terminate;
3. Operations that write to data memory require 3 cycles;
4. All other operations require 5 cycles;
5. A normal program consists in 20%, 20% and 60% of the operations 2,3 and 4 respectively.

On average the execution time of an instruction is given by:

$$avg\_exe\_t = 4 * (0.2 * 4 + 0.2 * 3 + 0.6 * 5) = 4 * 4.4 = 17.6ns \tag{2.3}$$

Supposing that the pipelined architecture slows down the clock frequency by 20% (i.e. it increases the period by 0.8 ns). Once the pipeline is full, the average instruction execution time coincides with the clock period, that is 4.8 ns. Thus, the speedup introduced by pipelining is:

$$speedup = \frac{17.6}{4.8} = 3.7 \tag{2.4}$$

The pipelined architecture is 3.7 times faster than the initial one.

2.2 The RISC-V Instruction Set Architecture

The ISA represents the software-hardware interface for any microprocessor core and is the starting point for its design. Among the many available ISAs, I chose RISC-V (pronounced *risc five*), a modern general-purpose RISC architecture that has been recently introduced by researchers at the University of California, Berkeley [8] [9]. Since its initial inception in 2010, it has become a popular alternative in academia and has gained particular traction in industry. Semiconductor companies such as IBM, Google and Oracle among many others, have joined the RISC-V foundation. This non-profit corporation founded in 2015 is controlled by its members, who direct the advancements of the RISC-V ISA by maintaining and releasing the official ISA specifications, as well as periodically organizing events and workshops.
Among many reasons, RISC-V is an appealing solution because:

- It is an universal instruction set with the goal of serving all market sectors, from ultra-low power microcontrollers to data intensive processors, such as those found in large servers;
- For smaller architectures (i.e. not VLIW, superscalar, etc.), the footprint is drastically reduced with respect to typical ARM or x86 solutions;
- Provides a base instruction set for 32, 64 or 128-bit architectures which can be extended with official or custom subsets;
- It is BSD licensed, so anyone can access and tailor the ISA to its particular needs.

To this day, a diverse set of RISC-V compatible chips and architectures have been produced. For instance, researchers at ETH Zurich and University of Bologna have created and have been regularly contributing to the PULP Platform [10], a parallel platform for ultra-low power computing which leverages RISC-V cores. PULP is released under the Solderpad Hardware License, and the source RTL code can be freely accessed on-line. Among the various cores PULP provides, I have chosen to compare my implementations with Zero-riscy [11], a simple RISC-V processor supporting the RV32I and RV32M instruction subsets.
To introduce some of RISC-V's features, some general details are covered in the following as well as information concerning the RV32I and RV32M subsets. These instructions are fully supported by my design (Chapter 3). The reader can refer to [12] for a more in-depth analysis and description of all subsets. Table 2.1 lists the three base instruction sets and the 6 official extensions. All have a clean and fixed length encoding, while variable length encodings are only permitted in custom extensions. There are 32 general purpose registers

(x0-x31), to which the programming conventions listed in Table 2.2 are applied. Additionally, each implementation can define an arbitrary collection of Control and Status Registers (CSRs) to manage and provide system policies such as multi-threading or privilege levels [13].

Subset	**N. Instructions**	**Description**
RV32I	47	32-bit address space and integer instructions
RV64I	59	64-bit address space and integer instructions, in addition to RV32I
RV128I	71	128-bit address space and integer instructions, in addition to RV32I and RV64I
Extension Subsets	**N. Instructions**	**Description**
M	8	Integer multiplication and division
A	11	Atomic memory operations
F	26	Single-precision (32-bit) floating point operations
D	26	Double-precision (64-bit) floating point operations, in addition to the F extension
Q	26	Quad-precision (128-bit) floating point operations, in addition to the F and D extensions
C	46	Compressed (16-bit encoding) integer instructions

Table 2.1: RISC-V base and extension instruction sets.

Register	**Description**
x0	Hard wired to zero
x1	Return address
x2	Stack pointer
x3-x31	Temporary, function arguments/return values

Table 2.2: RISC-V general purpose registers coding conventions.

In RV32I, the 47 instructions are encoded on 32 bits and can be functionally classified as follows:

- Integer Register-Register: perform arithmetic and logical operations with both operands as general purpose registers;
- Integer Register-Immediate: perform arithmetic and logical operations with one operand as a general purpose register and the other represented as the value of the immediate field;
- Control Transfer: used to alter the program flow (branches and jump instructions);
- Load and Store: for accessing data memory for reading (load) or writing (store);
- System: for operating on the CSRs and managing tasks related to the operating system.

Figure 2.3 reports the encoding for the integer register-register (R-type) and the store (S-type) types of instructions. The regularity in the encoding among classes of instructions greatly simplifies the decoding logic. Although this may results in complex encoding schemes, such as the immediate field in S-types being split, this approach aims at solving some implementations aspects at the ISA level. As a matter of example, the operands, which are usually on the critical path of a processor core, are in fixed positions.

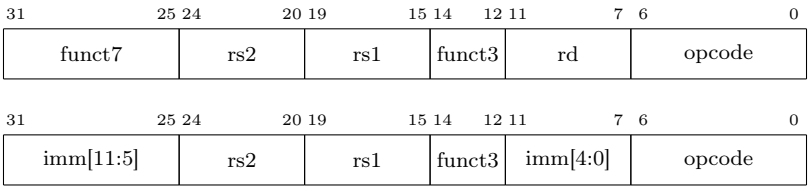


Figure 2.3: RV32I R-type and S-type instruction encodings.

RV32M is an 8 instruction subset dedicated to multiplication and division operations. To obtain the full result of a multiplication, a sequence of two instructions is needed. In fact, given two 32-bit operands, `mul` returns the lower 32 bits of the result while `mulh`, `mulhsu`, `mulhu`, return the upper 32 bits considering the input operands as signed, signed-unsigned and unsigned pairs respectively. For division, the quotient and remainder are obtained with `div` and `rem` when considering signed operands, while `divu` and `remu` are used with unsigned operands.

2.3 System-level Design and High-level Synthesis

In recent years, struggles in productivity of the semiconductor industry have led to the investigations of new design methodologies. Traditional bottom-up approaches have been demonstrated to be inefficient in the light of modern heterogenous SoCs, mainly because local optimizations do not necessarily entail global ones. This is a crucial aspect considering the heterogeneity of these devices and a higher perspective of the system is necessary. For such reason, recent system-level design (SLD) methodologies have gained the attention and interest of several companies in the industry. These methods have been successfully applied to components that deal with large sets of data and perform computationally intensive tasks, such as computer vision or signal processing applications.
In this dissertation, I propose an SLD methodology for microprocessor design which leverages high-level synthesis (HLS). HLS tools have become more popular and are increasingly evolving, and supporting many high-level languages, such as C/C++, SystemC, BlueSpec and MATLAB [14] [15].

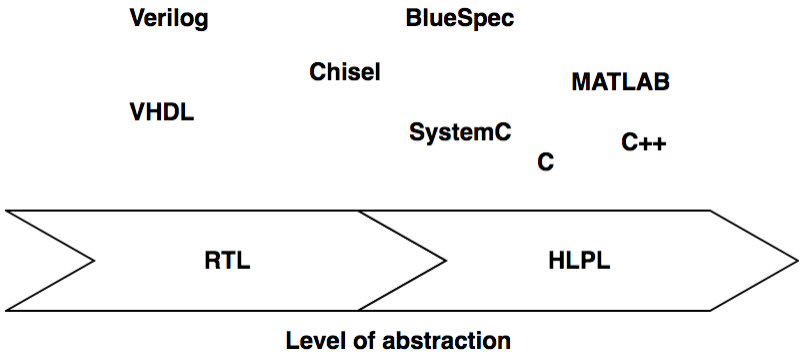


Figure 2.4: Commonly used languages for hardware design.

Many companies which have strictly been working on software oriented products are now producing and using custom hardware to gain a competitive advantage that is unsurmountable by commonly used software solution [16] [17]. For these applications HLS is the right option as engineers can focus on the data structures and the characteristics of the algorithm to implement. These components can be simulated on virtual platforms [18] which are faster than RTL simulators and easily integrate the software stack that is meant to be run on the final product. Although with its limitations, mainly the reduced set of high-level programming language features that it supports, HLS provides designers with a vast collection of configuration knobs that enables the automatic synthesis of different microarchitectures, starting from a single system specification. This process is known as design space exploration (DSE, Figure 2.5) and can be exploited by designers to perform

multi-objective optimizations. The result is a Pareto set, i.e. a collection of optimal implementations in the considered design space (area, latency, power, etc.). Figure 2.6 gives a qualitative idea, implementations on curves 1 and 3 take part of the Pareto set, while the ones on curve 2 are fully dominated, and so they can be discarded.

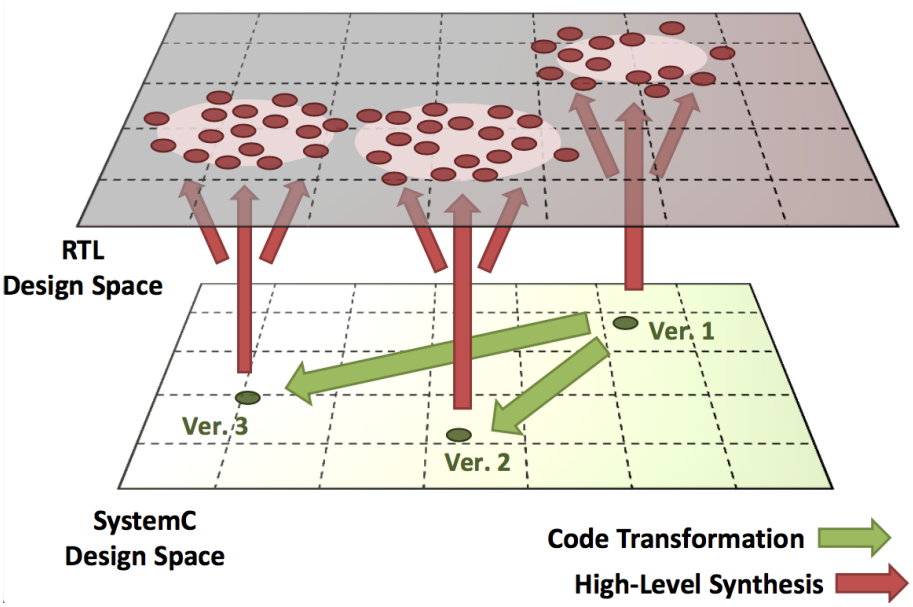


Figure 2.5: Design Space Exploration [1].

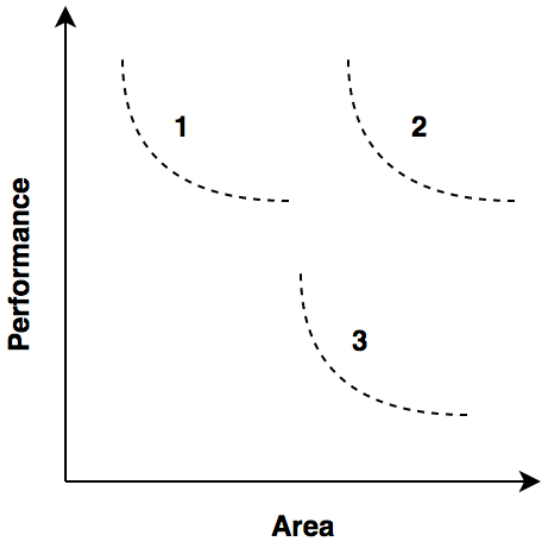


Figure 2.6: Example of Pareto curves in the performance-area space.

2.3.1 The SystemC Class Library

Since its initial inception in 1999, SystemC has grown to become an IEEE-standard (2005) under the guidance of the Open SystemC Initiative (OSCI), with its last revision released in 2011.
SystemC is a C++ library which has been developed to support system-level design and verification. Although still evolving, it incorporates hardware and software concepts which are generally treated separately by other languages, and thus can be used for system-level modeling, architectural exploration, verification and high-level synthesis [19].
The main features which are added to the C++ language are reported in the following.

- Time model: at its core, the library provides an event-driven simulation kernel which manages the timing of each existing process;
- Hardware data types: these support user-defined bit widths for integer and fixed-point data types, as well as non-binary values such as *high-impedance* and *unknown* commonly used in digital systems;
- Hierarchy and structure: designs can be broken down into sub-modules which are integrated to form a larger block. Hierarchy enables an easier comprehension and re-usability by the engineering team;
- Communications management: communication between modules can be modeled as simple wires or as more complex communication infrastructures such as industrial-grade bus-architectures. Modules are interconnected via ports and exchange information through channels. Moreover, it is possible to have different versions of a channel and use them interchangeably;
- Concurrency: the simulation kernel provides the illusion of executing processes concurrently, as if they were real hardware units.

In order to understand implementation details found in later chapters, it is worth spending a few words on the building blocks of a SystemC design: modules and threads.
Modules are used to encapsulate functionalities and are created using the `SC_MODULE` base class. They may incorporate other modules, processes, channels and ports.
Processes, which are scheduled by the simulator, are defined as member functions of `SC_MODULE` classes. They are C++ functions which return a `void` value and have an empty argument list. From a software viewpoint processes are threads of execution, while in hardware terms they model independently timed circuits.
There are three kinds of processes:

- `SC_METHOD`: its execution will not cause the simulation time to advance and is invoked only once, thus it is usually used to model combinational logic;
- `SC_THREAD`: it can be called multiple times and can suspend itself by calling the `wait()` function, allowing time to pass before continuing execution;
- `SC_CTHREAD`: it merges the features of an `SC_THREAD` with the needs of synthesis, in fact, when employed, one must assign clock and reset signals to it. This is the only kind of process that is used in my design;

2.3.2 The Theory of Latency Insensitive Design

Latency-insensitive design (LID) is a correct-by-construction design methodology that meets very well the challenges of designing modern SoC.
At its core, LID is comprised of the *protocols and shells* paradigm [20] which is the backbone of obtaining a physical design starting from a system-level description. The protocols separate the communication and computation portions of a system by defining it as a collection of computational processes exchanging data through interfaces and channels, thus enabling the *latency-insensitiveness* of the communication with respect to the delay of the channels themselves. In addition, once the protocol has been defined, the interfaces (shells) can be automatically generated. To designers this is a very attractive feature, not having to deal with data synchronization issues typical of digital hardware design, and being able to focus solely on the computational units and explore the design space.
We can say that by its nature, LID supports the concept of re-usability typical of SLD: once the interfaces have been defined to respect a latency-insensitive protocol, the units can be seamlessly replaced by other implementations. Ultimately, this enables a scalable communication and computation infrastructure.
Among its advantages, we can point out that LID is efficient from a design viewpoint (it enables the reuse of components) and scalable (the automatic generation of interfaces renders a correct-by-construction system).
In my work LID has been adopted to handle communication among pipeline stages. Section 3.2 covers the implementation aspects of including such feature in my design.

Chapter 3

RVXRed: A System-Level Microprocessor

This chapter first introduces details regarding the generation of an RTL description starting from SystemC source code, then it covers latency-insensitive channels (LICs), the means through which information flows in the proposed pipeline. These concepts are necessary in order to understand the design.

The proposed architecture is named RVXRed and supports the RV32I and RV32M subsets of the RISC-V ISA specification for a total of 54 instructions (Appendix A).

The name 'RVXRed' originated when first developing a very basic version of the core, which only supported 9 instructions. The RISC-V Instruction Set Manual dictates a naming convention for custom subsets, which consists in appending an alphabetic identifier to the letter *X*. Given the initially limited amount of supported instructions, *Red*, abbreviating the word *reduced*, was adopted and the label *RVXRed* was extended to the processor core itself. The name continued to be used even after fully extending the instruction set to the RV32I and RV32M subsets and its meaning today does not represent what it initially stood for.

3.1 From SystemC to Verilog RTL

One of the first steps performed by behavioral synthesis tools is separating the design into two portions: control and datapath [21] [22]. The input source code is scheduled and in part transformed into a Finite State Machine (FSM) representation. Based on the data dependencies of the algorithm to synthesize and the latency of the units in the technology library, the operations of the algorithm are assigned to specific clock cycles and monitored

by the FSM. Given an algorithm, there may be more than one possible schedule. This is where HLS directives can come into play, shaping the resulting implementation.

Among these directives, some can be used to instruct the HLS tool that the description is cycle-accurate. In practical terms this means that no `wait()` statement within the protocol region is pruned or added by the tool. In the following listings, the directive labeled as `PROTOCOL_REGION()` is used to specify such code regions. The designer can also choose to break the protocol region in specific code sections. In such areas the HLS tool freely decides how many FSM states will be created and which operations are assigned to them.

When following an RTL design flow, the control portion of a digital system is defined explicitly as an FSM using constructs provided by the chosen language. In my approach, the control part is implemented as an FSM that is declared implicitly by using `wait()` statements contained within protocol regions. This implies that the HLS tool might not be able to schedule the datapath operations within the user defined states. As a consequence, it is the designer's responsibility to understand how FSMs are inferred when using protocol regions.

To familiarize with implicit FSMs, an example is here presented.

Let's take Listing 3.1. This SystemC source code describes a module we want to synthesize. It is comprised of the `SC_MODULE` name `mod`. The first statement of the `mod_cthread` `SC_CTHREAD` is the definition of a protocol region, so the whole function is considered in a cycle-accurate manner. For each call to `wait()`, we are forcing the creation of a state (in the listing, states are in-lined as comments for easier comprehension).

```
1
2   SC_MODULE(mod){
3     sc_in < bool > rst_n;
4     sc_in < bool > clk;
5     sc_in < bool > cond;
6     sc_in < sc_int<32> > x;
7     sc_in < sc_int<32> > y;
8     sc_out < sc_int<32> > t;
9     sc_out < sc_int<32> > w;
10    sc_out < sc_int<32> > z;
11
12    sc_int<32> tmp_x, tmp_y;
13    void mod_cthread();
14
15    SC_CTOR(){
16      SC_CTHREAD(mod_cthread, clk.pos());
```

```
17          reset_signal_is(rst, false);
18        }
19
20        void mod::mod_cthread(){
21          PROTOCOL_REGION();
22          x = 0;
23          y = 0;
24
25                  while(true){
26                      wait();    // State 0
27                      tmp_x = in_1.read();
28                      tmp_y = in_2.read();
29
30                      if(cond.read() == true){
31                          t = tmp_x * tmp_y;
32                          wait();    // State 1
33                          w = tmp_x * tmp_y;
34                      }
35                      else{
36                          wait();    // State 2
37                          t = tmp_x - tmp_y;
38                          w = tmp_x + tmp_y;
39                      }
40
41                      wait();      // State 3
42                      z = tmp_x * tmp_y;
43                  }
44        }
```

Listing 3.1: Example SystemC source code input.

This translates to the FSM of Figure 3.1 and the Verilog RTL description of Listing 3.2. As expected the FSM is made of a total of four states.

```
1
2     module mod(rst_n, clk, x, y, t, w, z);
3       input rst;
4       input clk;
5       input cond;
6       input [31:0] x;
7       input [31:0] y;
```

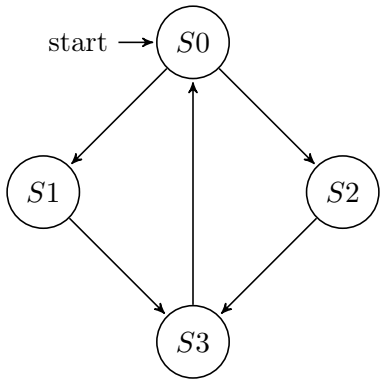


Figure 3.1: FSM resulting from the example SystemC module.

```
8         output  [31:0] t;
9         output  [31:0] w;
10        output  [31:0] z;
11
12        reg [31:0] rx, ry, rt, rw, rz;
13        reg [1:0] gbl_state;
14
15        always @(posedge(clk))
16          begin:
17            if(!rst) begin
18              rx = 0;
19              ry = 0;
20              gbl_state <= `S0;
21            end
22            else begin
23              case(gbl_state)
24                `S0: begin
25                  rx = x;
26                  ry = y;
27                  if(cond == true) begin
28                    rt = rx * ry;
29                    glb_state <= `S1;
30                  end
31                                  else
32                    glb_state <= `S2;
33                  end
34                end
35                `S1: begin
36                  rw = rx * ry;
37                  glb_state <= `S3;
38                end
```

```
39              `S2: begin
40                rt = rx - ry;
41                rw = rx + ry;
42                glb_state <= `S3;
43              end
44              `S3: begin
45                rz = rx * ry;
46                glb_state <= `S0;
47              end
48            endcase
49          end
50        end
51
52      assign t = rt;
53      assign w = rw;
54      assign z = rz;
55    endmodule
```

Listing 3.2: Verilog description resulting from the example SystemC module.

We see that starting from a SystemC description in which all `wait()` statements are contained within a protocol region, its Verilog description is easily predictable. There is a one to one correspondence between each `wait()` statement and the FSM states. As previously mentioned, this would not be the case in a region where the protocol is broken, as the FSM states would be generated according to the HLS tool's scheduling decisions.

3.2 LICs: Latency-Insensitive Channels

In my design, the concept of latency-insensitiveness introduced in Section 2.3.2 is put into practice with the use of LICs. These are point-to-point pipes through which inter-stage information flows. From a software standpoint, LICs are made of interfaces (one type for the receiving end and one for the transmitting end) and pipes. The former provide the means to send or receive the information, while the latter carry the data itself. One of the many advantages of using LICs is that they can be either simulated in TLM or at the signal level without changing the source code that is used to define them. By simply setting or not a preprocessor directive we can instruct the tool at which level we are intending to work at. The TLM version can be used for fast system-level prototyping (in terms of simulation speed), while the signal level implementation is used in high-level synthesis and is the one we are referring to in the following.

At the signal level, a LIC includes an arbitrary amount of wires dedicated to the data to

be transmitted and 2 wires for implementing a ready-valid handshaking mechanism like the one depicted in Figure 3.2, resembling a typical latency-insensitive protocol.

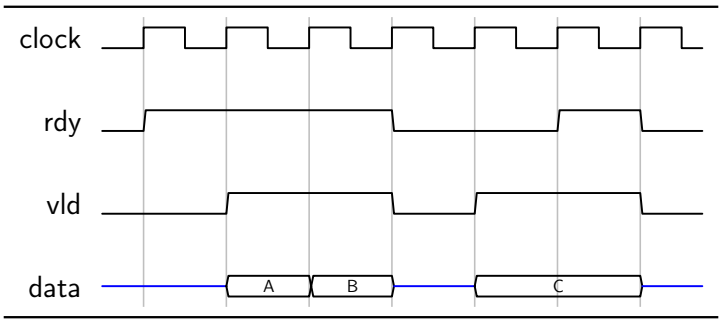


Figure 3.2: Signal-level LIC handshaking protocol.

LIC interfaces are either of type put (for transmitting) or get (for receiving) and each exposes 2 methods:

- void reset_put(): resets the put interface;
- void reset_get(): resets the get interface;;
- void put(T value): puts *value* on the associated LIC pipe;
- T get(): returns the data (if any) from the associated LIC pipe.

The reset methods initialize the interfaces and the channel to which these are attached to. This ensures that they start in a consistent state. The behavior of a call to `get()` or `put()` is best described in terms of their CDFGs (Figures 3.3 and 3.4) and behavioral code (Listings 3.3 and 3.4). It is clear that both of these invocations may result in a call to a `wait()` statement if data is not valid (in case of a `get()`) or if the receiving end is not ready (for calls to `put()`; this is equivalent to the concept of back-pressure proper of latency-insensitive design [23]). The darkened states represent the possibility of not waiting for the next clock cycle if data or the receiver are ready.

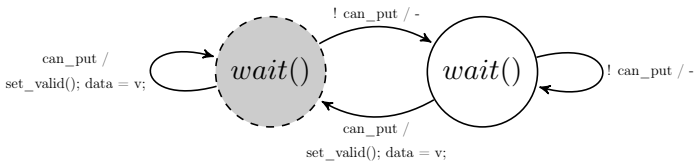


Figure 3.3: CDFG of the put method.

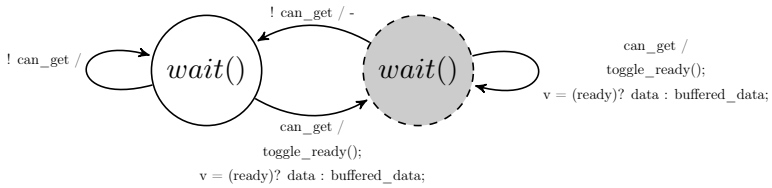


Figure 3.4: CDFG of the get method.

```
1  while(!can_put) wait();
2  toggle_valid_curr();
3  data = v
```

Listing 3.3: put() sample implementation.

```
1  while(!can_get) wait();
2  toggle_ready_curr();
3  v = (ready)? data :
     buffered_data;
```

Listing 3.4: get() sample implementation.

As shown in the next section, LICs are used at the interface of each pipeline stage and provide the means to clearly separate the computation and communication portions of the system.

3.3 An HLS Approach to Microprocessor Design

A first major challenge was to find the most effective way to describe a pipeline and its stages in SystemC. Previous experience with modeling a processor in RTL was very useful, but shifting the abstraction to a higher level meant a change in how the architecture should be conceived and not all practices of the RTL methodology could be re-used.
The result was to describe each stage as an `SC_CTHREAD` called within a dedicated `SC_MODULE`. The thread is comprised of two parts: a reset section where initializations are performed, and an infinite loop where normal communication and computation occur (Listing 3.5). This is the typical approach when describing digital hardware.

```
1  void pipeline_stage_cthread(){
```

```
2
3         {
4             PROTOCOL_REGION("pipeline_stage reset protocol");
5             from_previous_stage_if.reset_get();
6             to_next_stage_if.reset_put();
7             // ...
8         }
9
10        while(true){
11            PROTOCOL_REGION("pipeline_stage body protocol");
12            din = from_previous_stage_if.get();
13            dout = compute(din);  // DSE, if any, can be applied here
14            to_next_stage_if.put(dout);
15            wait();
16        }
17    }
```

Listing 3.5: Example pipeline stage thread

The first part is the reset region executed at system start up, this includes reset configurations such as initializing the LIC interfaces, and other stage-dependent reset operations. The second and main portion, is an infinite loop which acquires new data from the previous stage, performs some computation on such data and finally transfers the processed information to the following stage. It should be clear that the computation section of the loop can be *un-timed* (that is, no constraints are forced on the timing of the final hardware implementation related to such code section), hence behavioral synthesis tools can here be leveraged to perform optimizations. This is done by breaking down the protocol region and providing directives to influence the RTL generation. This way, there is a clear separation between the I/O sections, which are expressed in a way that is closer to real hardware, from the computation sections of the code. In particular, `from_previous_stage` and `to_next_stage` are LIC interfaces templatized to the same type of `din` and `dout`, the data structures associated to the pipes. As a matter of example, Listing 3.6 presents the data structures and LIC interfaces for the *memwb* stage (presented in Section 3.3.4). The members of each `struct` are of type `sc_bv`, this effectively models a single or a bundle of wires (for simplicity, the many contents of the `exe2memwb_t` structure are omitted in the listing). Whenever these wires should be used for computation it is possible to cast them to other types (such as `sc_int`) that support the C++ arithmetic and logic operators.

```
1
```

```
2   /* rvxred_datatypes.h */
3   struct exe2memwb_t{
4     // ...
5   };
6
7   struct memwb2fedec_t{
8       sc_bv <1> regwrite;
9       sc_bv <5> regfile_address;
10      sc_bv <32> regfile_data;
11  };
12
13
14  /* memwb.h */
15
16  // LIC interfaces
17  LIC_get_if<exe2memwb_t> memwb_get_if;
18  LIC_put_if<memwb2fedec_t> memwb_put_if;
19
20  // LIC data structures
21  exe2memwb_t memwb_din;
22  memwb2fedec_t memwb_dout;
```

Listing 3.6: Definition of the data structure and LIC interfaces for the memwb stage.

3.3.1 Architecture

After having defined the thread structure, writing the source code for the pipeline stages is straightforward. Each of the following subsections presents and describes the code that was developed.

Figure 3.5 graphically introduces the adopted pipeline structure. There are 3 `SC_MODULEs` (fedec, execute, memwb), each with an associated `SC_CTHREAD`, which are encapsulated in an upper layer (rvxred). The use of multiple `SC_MODULEs` was initially intended for simply keeping the design modular and thus easier to manage, however it has an additional advantage. In fact, an alternative solution could be to instantiate the 3 threads within a single module, but this does not enable the observation of signals exchanged among stages during logic simulation.

Threads have multiple data structures, some are used for storing temporary values while others are associated to the LICs' get and put initiators. Information among threads flows through these pipes, which effectively decouple communication from computation.

The execute stage is where most of the DSE procedures can be applied, mainly aiming at

reconfiguring the structure of the algorithms it implements (addition, subtraction, multiplication, division). Still, some HLS directives can be applied to the all stages, as discussed later (Chapter 4).

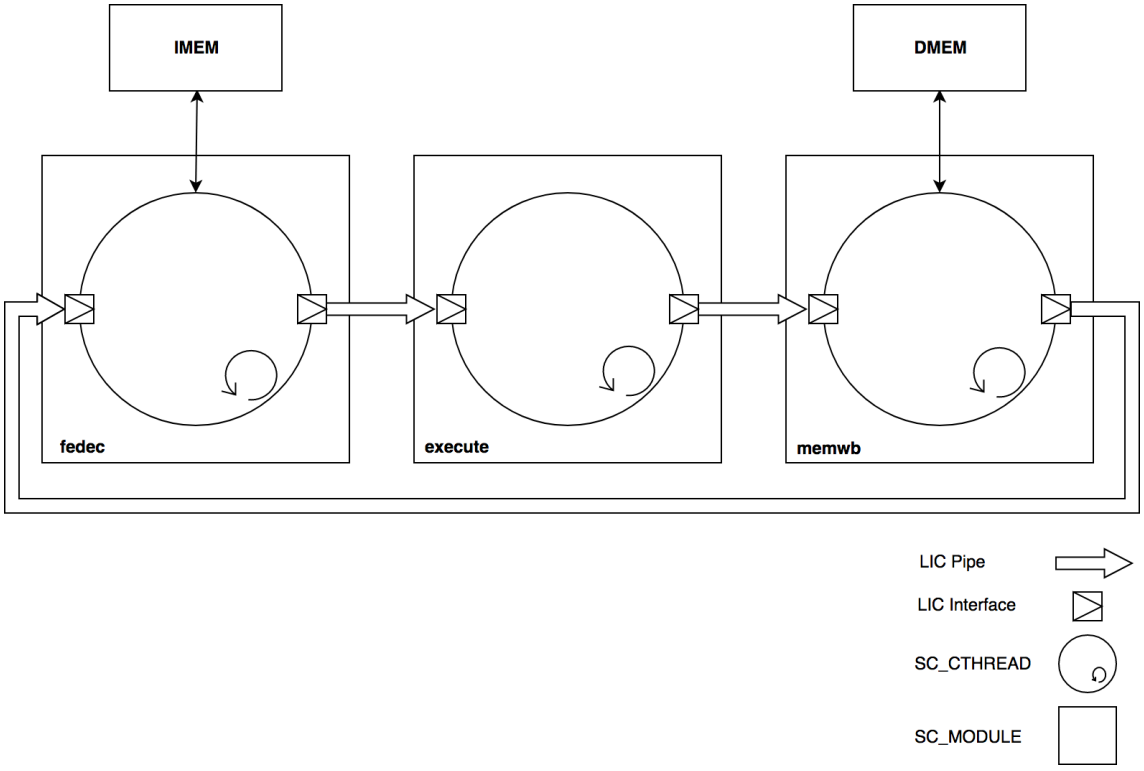


Figure 3.5: High-level architecture of the RVXRed pipeline.

3.3.2 Fedec

The fetch and decode stages are the beginning of a pipeline and in most architectures they are entities separated by pipeline registers. However, this concept turned out not to be explicitly definable when modeling in a high-level programming language. In fact, fetching involves reading from the instruction memory. Typically, the core requests an instruction by providing the address to the memory, which provides such data with one clock cycle of delay. My initial architecture had two separate threads, one for each stage, but intuitive as it may be, this is not the correct way to describe fetching and decoding. In addition, to the clock cycle required to obtain data from memory, one more is needed to propagate it the decode stage. This is in unacceptable characteristic and has led to merging both threads into one. Listing 3.7 reports parts of the fedec thread. Note that, before accessing the memory (line 17), a `wait()` statement is necessary to correctly imply the FSM and obtain a schedulable design. The same concept is applied to the memwb stage, which

combines the memory and writeback stages. The remaining parts of the fedec thread are straightforward, after fetching, it gets data from the writeback stage. It may then write to the register file, read from it and proceeds with decoding the instruction fields and propagating information to the following stage (*execute*).

From a SystemC viewpoint the register file is abstracted as a simple array of type `sc_bv` while the memory is generated automatically by the HLS tool. The designer is in charge of defining the interfaces from the core to the memory and instructing the tool on the characteristics of the desired memory (such as the number of ports, bit-width, number of entries and so on). The tool automatically generates the SystemC and Verilog descriptions of the memory based on these parameters.

In the case of the instruction memory only one interface is needed (the read port), but for the data memory (accessed by the memwb thread) two distinct ports must be defined and used (one for reading, the other for writing).

```
1
2     while(true){
3         PROTOCOL_REGION();
4
5         // --- Fetch
6         if(self_feed.jump == "1")
7             pc = (sc_uint<PC_LEN>)self_feed.jump_address;
8         else if(self_feed.branch == "1")
9             pc = (sc_uint<PC_LEN>)self_feed.branch_address;
10        else if(!freeze)     // If fetching is unfrozen increment the PC.
11            pc = sc_uint<PC_LEN>(pc + 4);
12
13        freeze = false; // Un-freeze fetching
14        output.pc = pc;
15
16        wait();
17        insn = sc_bv<INSN_LEN>(imem_port[pc]); // Retrieve instruction from
      imem
18
19        if(insn == "11111111111111111111111111111111"){  // HALT instruction
20            program_end.write(true);          // Signal end of program
      execution.
21        }
22        // --- ENDOF Fetch
23
24    // --- Decode
```

```
25      feedinput = feed_from_wb.get(); // Get from writeback.
26          // Register file write.
27          if(feedinput.regwrite == "1" && feedinput.regfile_address != "00000"
        ){
28              regfile[sc_uint<REG_ADDR>(feedinput.regfile_address)] =
        feedinput.regfile_data;
29              sentinel[sc_uint<REG_ADDR>(feedinput.regfile_address)] = 0;
30          }
31
32          // Register file read.
33          output.rs1 = regfile[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range
        (19, 15)))];
34          output.rs2 = regfile[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range
        (24, 20)))];
35
36          // Handle jump instructions.
37          if(insn.range(6, 0) == OPC_JAL){
38              self_feed.jump_address = sc_bv<PC_LEN>((sc_int<PC_LEN>)
        sign_extend_jump(immjal_tmp) + (sc_int<PC_LEN>)pc);
39              self_feed.jump = "1";
40          }
41          else if(insn.range(6, 0) == OPC_JALR){
42              // ...
43          }
44          else{
45              self_feed.jump = "0";
46          }
47
48          // Handle branch instructions.
49          self_feed.branch = "0";
50          if(insn.range(6, 0) == OPC_BEQ){
51             switch(sc_uint<3>(sc_bv<3>(insn.range(14, 12)))){
52                 case FUNCT3_BEQ:
53                     if(regfile[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range
        (19, 15)))] == regfile[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range(24,
        20)))])
54                         self_feed.branch = "1";  // BEQ taken.
55                     break;
56                 // ... case statements for other branch instructions
57                 default:
58                     self_feed.branch = "0";  // default to not taken.
59                     break;
60             }
61          }
```

```
62
63          // Control word generation.
64          switch(sc_uint<OPCODE_SIZE>(sc_bv<OPCODE_SIZE>(insn.range(6, 2)))){
65              case OPC_ADD: // R-type instructions.
66                  output.alu_src  = (sc_bv<ALUSRC_SIZE>)ALUSRC_RS2;
67                  output.regwrite = "1";
68                  sentinel[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range(11, 7)
    ))] = 1;
69                  output.ld       = NO_LOAD;
70                  output.st       = NO_STORE;
71                  output.memtoreg = "0";
72                  trap = "0";
73                  trap_cause = NULL_CAUSE;
74                  // FUNCT7 decodes the class of R-type instruction.
75                  switch(sc_uint<7>(sc_bv<7>(insn.range(31, 25)))){
76                      case FUNCT7_SUB:     // SUB, SRA
77                          switch(sc_uint<3>(sc_bv<3>(insn.range(14, 12)))){
78                              case FUNCT3_SUB:
79                                  output.alu_op   = (sc_bv<ALUOP_SIZE>)
    ALUOP_SUB;
80                                  break;
81                              case FUNCT3_SRA:
82                                  output.alu_op   = (sc_bv<ALUOP_SIZE>)
    ALUOP_SRA;
83                                  break;
84                              default:
85                                  output.alu_op   = (sc_bv<ALUOP_SIZE>)
    ALUOP_NULL;
86                                  break;
87                          }
88                          break;
89                      // ... other R-type FUNCT7 case statements.
90                      default:
91                          output.alu_op   = (sc_bv<ALUOP_SIZE>)ALUOP_NULL;
92                          break;
93                  }
94                  break;
95                  // ... other OPCODE case statements.
96                  default: // illegal instruction
97                      output.alu_src  = (sc_bv<ALUSRC_SIZE>)ALUSRC_RS2;
98                      output.regwrite = "0";
99                      output.ld       = NO_LOAD;
100                     output.st       = NO_STORE;
101                     output.memtoreg = "0";
```

```
102                     trap = "1";
103                     trap_cause = ILL_INSN_CAUSE;
104                     output.alu_op = (sc_bv<ALUOP_SIZE>)ALUOP_CSRRWI;
105                     output.imm_u.range(19, 8) = (sc_bv<12>)MCAUSE_A;
106                     output.imm_u.range(7, 3) = (sc_bv<5>)ILL_INSN_CAUSE;
107                     break;
108         }
109
110         // Stall mechanism
111         if( ((sentinel[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range(19, 15))
    )] == 1) && // If read-after-write (RAW) hazard on RS1...
112             (insn.range(6, 2) != OPC_JAL) &&
113             (insn.range(6, 2) != OPC_LUI) &&
114             (insn.range(6, 2) != OPC_AUIPC)   )   || // ... or RAW on RS2,
    send a bubble and freeze fetching for the next cycle.
115             ((sentinel[sc_uint<REG_ADDR>(sc_bv<REG_ADDR>(insn.range(24, 20)
    ))] == 1) &&
116             ((insn.range(6, 2) == OPC_ADD) ||
117             (insn.range(6, 2) == OPC_SB) ||
118             (insn.range(6, 2) == OPC_BEQ))   ) )
119         {
120             // Bubble.
121             output.regwrite      = "0";
122             output.ld            = NO_LOAD;
123             output.st            = NO_STORE;
124             // @ Next cycle, don't fetch a new instruction
125             freeze = true;
126         }
127     // --- ENDOF Decode
128
129         dout.put(output);
130
131      }
```

Listing 3.7: Fedec thread body

3.3.3 Execute

The Execute stage is the centerpiece of the pipeline as it performs arithmetic and logical operations on operands and returns a result to be stored either in data memory or in the register file. Generally the first operand is statically mapped to RS1 (the first register operand), while the second (RS2) depends on the instruction: the second register operand or the immediate field of the instruction. A large switch statement, governed by the

previously encoded `ALU_OP` signal, selects the operation that must be performed on the operands. All operations except for the C++ operators `/` (division) and `%` (modulo, or remainder) were synthesizable by the adopted HLS tool. A separate division algorithm was implemented and encapsulated as a function in a separate SystemC file (Section 3.3.3).

```
    while(true){
        PROTOCOL_REGION();

        input = din.get();

        // Propagate some signals to the downstream stage
        output.regwrite = input.regwrite;
        output.memtoreg = input.memtoreg;
        output.ld = input.ld;
        output.st = input.st;
        output.dest_reg = input.dest_reg;
        output.mem_datain = input.rs2;

        // ALU 2nd operand multiplexing.
        sc_bv<XLEN> tmp_rs2 = (sc_bv<XLEN>)0;
        if(input.alu_src == (sc_bv<ALUSRC_SIZE>)ALUSRC_RS2)
            tmp_rs2 = input.rs2;
        else if(input.alu_src == (sc_bv<ALUSRC_SIZE>)ALUSRC_IMM_I)
            tmp_rs2 = sigext_imm_i;
        else if(input.alu_src == (sc_bv<ALUSRC_SIZE>)ALUSRC_IMM_S)
            tmp_rs2 = sigext_imm_s;
        else     // ALUSRC_IMM_U
            tmp_rs2 = zerofill_imm_u;

        // ALU body
        switch(sc_uint<ALUOP_SIZE>(input.alu_op)){
            case ALUOP_ADD: // ADD, ADDI, SB, SH, SW, LB, LH, LW, LBU, LHU
                output.alu_res = sc_bv<XLEN>((sc_int<XLEN>)input.rs1 + (
sc_int<XLEN>)tmp_rs2);
                break;
            case ALUOP_SLT: // SLT, SLTI
                if(sc_int<XLEN>(input.rs1) < sc_int<XLEN>(tmp_rs2))
                    output.alu_res = (sc_bv<XLEN>)1;
                else
                    output.alu_res = (sc_bv<XLEN>)0;
                break;

```

```
37              // Other ALU operations
38              // ...
39
40              case ALUOP_DIV: // DIV calls div_func
41                  div_res = div_func((sc_int<XLEN>)input.rs1, (sc_int<XLEN>)
    tmp_rs2);
42                  output.alu_res = (sc_bv<XLEN>)div_res.quotient;
43                  break;
44
45              default:    // ALUOP_NULL
46                  output.alu_res = (sc_bv<XLEN>)0;
47                  break;
48          }
49
50          dout.put(output);
51          wait();
52      }
53  }
```

Listing 3.8: Execute thread body

Divider

The 32-bit division algorithm supports the execution of the `div`, `divu`, `rem` and `remu` instructions. The main computation loop (`DIVIDE_LOOP`) is contained with `udiv_func()`, it performs a serial division on unsigned integers and is leveraged by `div_func()` for signed division by simply reversing the quotient's sign whenever the numerator and denominator differ in sign.

Loops are commonly used to model hardware functionality in software. In this case, the main loop is a protocol-free region and DSE can be done to obtain different implementations. For example, *loop unrolling* replicates the logic inside the loop by a number of times indicated by a user-defined parameter. This means, one can control how much the loop is parallelized i.e. hardware is duplicated in order to process multiple loop iterations in a single cycle. In traditional RTL synthesis, there is no control of such kind and loops are always completely unrolled. In this case, a division which by the definition of the algorithm takes 32 clock cycles (CC), may be transformed into different implementations which may take as little as a few clock cycles to perform the operation. On the down-side the replication of hardware yields a larger area occupation.

Another common loop optimization technique is loop pipelining. This method enables one iteration of the loop to begin before the previous one has terminated. The result is an

increase in throughput while keeping the possibility of sharing resources, and thus minimizing the required area. Unfortunately, loop pipelining can't be applied to the mentioned algorithm because the algorithm includes long data dependencies between loop iterations. This type of data dependency occurs whenever a loop iteration requires input data that is produced by the previous iteration. Yet, for short dependencies (e.g. the `i++` operation for the loop iterator) it is possible to apply pipelining. In general, the longest data dependency chain needs to fit within the initiation interval.

```
1   u_div_res_t udiv_func(sc_uint<XLEN> num, sc_uint<XLEN> den){
2       sc_uint<XLEN> rem;
3       sc_uint<XLEN> quotient;
4       u_div_res_t u_div_res;
5       rem = 0;
6       quotient = 0;
7
8   DIVIDE_LOOP:
9       for(sc_int<6> i = 31; i >= 0; i--){
10          BREAK_PROTOCOL_REGION();
11          UNROLL_LOOP(4); // other loop optimization directives can go
    here
12          const sc_uint<XLEN> mask = 1 << i;
13          const sc_uint<XLEN> lsb = (mask & num) >> i;
14          rem = rem << 1;
15          rem = rem | lsb;
16          if(rem >= den){
17              rem -= den;
18              quotient = quotient | mask;
19          }
20      }
21      u_div_res.quotient = quotient;
22      u_div_res.remainder = rem;
23
24      return u_div_res;
25  }
26
27
28  div_res_t div_func(sc_int<XLEN> num, sc_int<XLEN> den) {
29      bool num_neg;
30      bool den_neg;
31      div_res_t div_res;
32      u_div_res_t u_div_res;
```

```
33
34          num_neg = num < 0;
35          den_neg = den < 0;
36
37          if(num_neg)
38              num = -num;
39
40          if(den_neg)
41              den = -den;
42
43          u_div_res = udiv_func((sc_uint<XLEN>) num, (sc_uint<XLEN>) den);
44          div_res.quotient = (sc_int<XLEN>)u_div_res.quotient;
45          div_res.remainder = (sc_int<XLEN>)u_div_res.remainder;
46
47          if(num_neg ^ den_neg)
48              div_res.quotient = -div_res.quotient;
49          else
50              div_res.quotient =  div_res.quotient;
51
52          return div_res;
53      }
```

Listing 3.9: Division algorithm

3.3.4 Memwb

This stage simply performs accesses to data memory whenever requested and sends information back to the register file. As discussed in Section 3.3.2 for the instruction memory, a `wait()` statement must be placed before an access to memory to correctly imply a schedulable design.

```
1   while(true){
2     PROTOCOL_REGION();
3     input = din.get();
4
5     // Memory access
6     wait();
7     if(input.ld != NO_LOAD) // Memory read (LOAD instruction)
8       mem_dout = dmem_port_2[input.alu_res];
9     else if(input.st != NO_STORE)    // Memory write (STORE instruction)
10      dmem_port_1[input.alu_res] = input.mem_datain;
11
```

```
12      // Writeback
13      output.regwrite          = input.regwrite;
14      output.regfile_address   = input.dest_reg;
15      output.regfile_data      = (input.memtoreg == true)? mem_dout : input.
        alu_res;
16
17      dout.put(output);
18    }
```

Listing 3.10: Memwb thread body

Chapter 4

Experimental Setup

This section covers the various tools used to test the device under test (DUT), as well as the environment that was set up in order to obtain results and performance metrics.
In the following, the DUT is a processor core: either a specific implementation of RVXRed or Zero-riscy (one of the RISC-V cores developed by researchers from ETH Zurich and University of Bologna [11]). In total, four instances of RVXRed were produced after a DSE phase:

- BASIC: a basic version that does not include the application of particular HLS knobs but has been the result of many code changes in the SystemC design space;
- ASAP: a faster version of BASIC obtained by controlling the composition of the datapath and control portions of the generated Verilog. This means that the HLS tool can optimize the latency of the system (at the expense of a larger area occupation) by blurring the distinction between these two parts. In fact, the default scheduling algorithm is designed for datapath-oriented designs (those that have a minimal control unit). The ASAP optimization changes approach by transforming control statements (such as `if`, `else`, `switch`, etc.) into datapath elements.
- UNDIV2: a version where the division algorithm's loop has been unrolled by a factor of 2;
- UNDIV4: as in UNDIV2, but with an unrolling factor of 4.

Figure 4.1 reports the experimental flow that was followed, starting from the SystemC description of the processor to the final results (when working on Zero-riscy the entry point was its Verilog description, thus the first step was skipped). Following logic simulation,

the left branch represents steps for the CMOS implementation while the right branch represents the steps for the FPGA design flow.

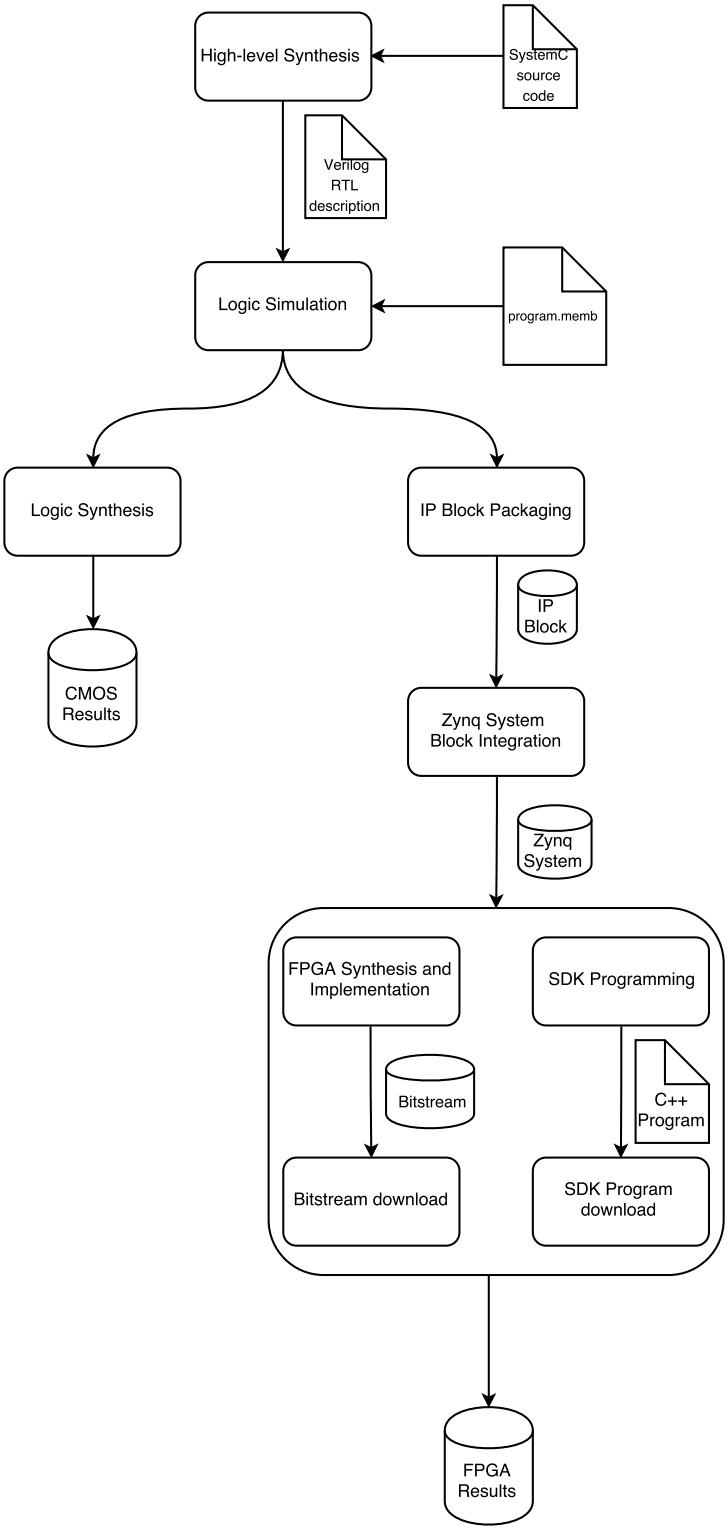


Figure 4.1: Experimental setup design flow.

4.1 Logic Simulation and Synthesis

In order to verify the DUT, logic simulation was performed. The goal was to assess the functional correctness of each core given a benchmark program consisting of all the supported instructions and checking the behavior of the DUT.

The test-bench used to feed input stimuli and observe the output results has the following structure:

```
1  `timescale 1ns / 1ps
2
3  `define CLK_PERIOD 40.00ns // 25 MHz
4
5  /* Test-bench module */
6  module tb
7  #(
8    parameter CLK_CNTR_WIDTH = 20
9  );
10 // Support logic wires
11 logic        prog_end;
12 logic        [CLK_CNTR_WIDTH-1:0] clk_counter;
13 logic        fetch_en;
14
15 // Clock and Reset
16 logic        clk_i;
17 logic        rst_ni;
18
19 // IMEM Block interface
20 logic        [31:0] imem_addr_o;
21 logic        imem_clk_o;
22 logic        [31:0] imem_din_o;
23 logic        [31:0] imem_dout_i;
24 logic        imem_en_o;
25 logic        imem_rst_o;
26 logic        [3:0]  imem_we_o;
27
28 // DMEM Block interface
29 logic        [31:0] dmem_addr_o;
30 logic        dmem_clk_o;
31 logic        [31:0] dmem_din_o;
32 logic        [31:0] dmem_dout_i;
33 logic        dmem_en_o;
34 logic        dmem_rst_o;
```

```
35  logic        [3:0]  dmem_we_o;
36
37
38  // Input program file handling.
39  integer        data_file
40  integer        scan_file
41  bit          [31:0] captured_data;
42  `define NULL 0
43
44
45    /* Instantiate the DUT */
46    rvxred_top DUT_rvxred_top(
47        // Support logic
48        .fetch_en_i(fetch_en),
49        .prog_end_o(prog_end),
50        .clk_count_o(clk_counter),
51
52        // Clock and Reset
53        .clk_i(clk_i),
54        .rst_ni(rst_ni),
55
56        // IMEM Block interface
57        .imem_addr_o(imem_addr_o),
58        .imem_clk_o(imem_clk_o),
59        .imem_din_o(imem_din_o),
60        .imem_dout_i(imem_dout_i),
61        .imem_en_o(imem_en_o),
62        .imem_rst_o(imem_rst_o),
63        .imem_we_o(imem_we_o),
64
65        // DMEM Block interface
66        .dmem_addr_o(dmem_addr_o),
67        .dmem_clk_o(dmem_clk_o),
68        .dmem_din_o(dmem_din_o),
69        .dmem_dout_i(dmem_dout_i),
70        .dmem_en_o(dmem_en_o),
71        .dmem_rst_o(dmem_rst_o),
72        .dmem_we_o(dmem_we_o)
73    );
74
75
76    // Clk-gen process
77    always
78        #(`CLK_PERIOD/2) clk_i = ~clk_i;
```

```
79
80
81      // Rst process
82      initial
83      begin
84          $display($time, "<< Starting the simulation >>");
85          data_file = $fopen("./program.memb", "r");
86          if (data_file == `NULL) begin
87              $error("program.memb handle was NULL");
88              $finish;
89          end
90          clk_i = 1'b0;
91          rst_ni = 1'b0;       // Reset the DUT
92          fetch_en = 1'b0;
93
94          #1500ns;
95          rst_ni = 1'b1;       // Release reset
96
97          #1500ns;
98          fetch_en = 1'b1;     // Start executing program.
99      end
100
101     // Prog-end process
102     always_comb begin
103         if(prog_end === 1'b1) begin
104             $display("**prog_end received");
105             $finish;
106         end
107     end
108
109     // Mem-mgmt process
110     always_ff @(posedge clk_i) begin
111         // Once we enable fetching, we send one instruction per clock cycle
112        // Instructions are read from the 'program.memb' file
113         if(fetch_en === 1'b1 && imem_en_o === 1'b1) begin
114           scan_file = $fscanf(data_file, "%b\n", captured_data);
115           if (!$feof(data_file)) begin
116               $display(captured_data);
117               imem_dout_i <= captured_data;
118           end
119           else begin
120               $finish;  // Reached EOF, end simulation
121           end
122            // Emulate a DMEM that responds with a fixed value of dout equal
```

```
    to 20
123         if(dmem_en_o === 1'b1 && dmem_we_o === 4'b0000) begin
124                 dmem_dout_i <= 32'h00000014;
125             end
126             else begin  // If a read is not requested output 0.
127                 dmem_dout_i <= 32'h00000000;
128             end
129         end
130         else begin
131             imem_dout_i <= 32'h00000000;
132             dmem_dout_i <= 32'h00000000;
133         end
134     end
135
136     endmodule
```

Listing 4.1: Logic simulation test-bench.

The test-bench instantiates the DUT and has four processes:

1. Clk-gen: the clock generation process, which inverts the `clk_i` signal every `CLK_PERIOD/2`;
2. Rst: opens the program file handler, resets the DUT and sends the signal for starting the execution (`fetch_en`);
3. Prog-end: sends the `$finish` system task to the simulator when `prog_end` is asserted;
4. Mem-mgmt: at each clock cycle it reads a line from `program.memb` before sending it to the DUT, and emulates the data memory. Note: just for demonstration purposes any data memory reads (load instruction in the processor) returns the hard-coded value of 20. This value has no particular meaning and any value different from 0 could have been outputted to simulate a read operation. This approach was only adopted in the logic simulation step, where the focus was on studying the behavior of the processor as executing all supported instructions. In the experiments conducted on the FPGA (Section 4.2), memory blocks were used in order to enable the execution of real programs.

The program to be executed is contained in `program.memb`. It is a text file containing the binary encoding of an instruction per line. To obtain it, a script which calls programs provided by the RISC-V software tool-chain was written. Starting from a C/C++ or assembly source file, the object code resulting from compilation is parsed and manipulated to obtain the binary encoding of each instruction.

This setup was run in a commercial logic simulation program. Both CPU interfaces to memories were examined in a waveform window to monitor their behavior. Instruction fetching was observed on the instruction memory interface, while any write or read operations were observed on the data memory interface. Additionally, the signals composing the 3 LICs between the stages were tracked to ensure the correct behavior of each individual stage. This was extremely useful as new instructions or functionalities were added to new versions of RVXRed.

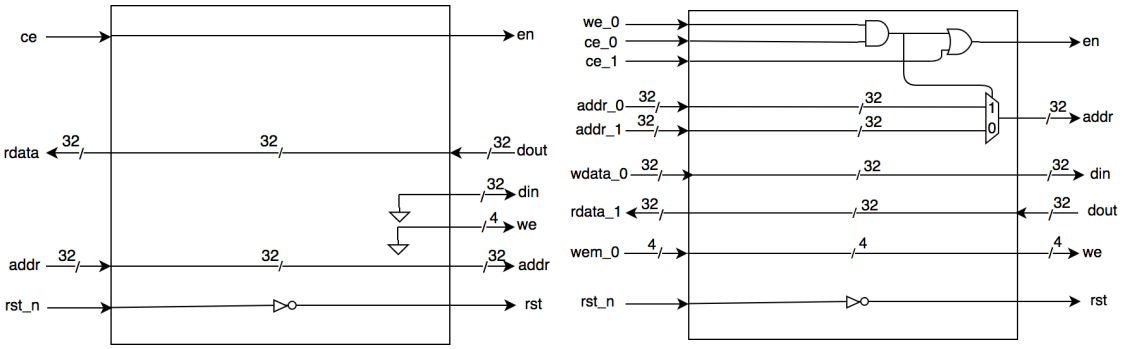
By counting the clock cycles from the start of an instruction's fetch to a write operation in the register file, the latency of the execution of a single instruction was determined. In the ideal case (no dependencies of any sort) for a RVXRed core, an instruction is fetched at each clocked cycle and the latency for committing it is equivalent to 5 clock cycles. A store operation instead needs 4 cycles as there is no writeback operation to the register file. Additionally, there is no clock cycle penalty for branch and jump instructions.

Once the DUT was validated, its gate-level description was obtained by resorting to a commercial logic synthesis tool. This operation yielded the first set of results used for comparing RVXRed with its rival implementation. The metrics of interest were area occupation and clock frequency. Refer to section 5.2 for the comparisons and discussion of the CMOS implementations.

4.2 FPGA Verification

Before deploying and testing the DUT on an FPGA, The RTL description was packaged as an intellectual property (IP) module, the basic building block for modern FPGA-based tools. IP packaging enables the re-use of a module in separate projects and systems.

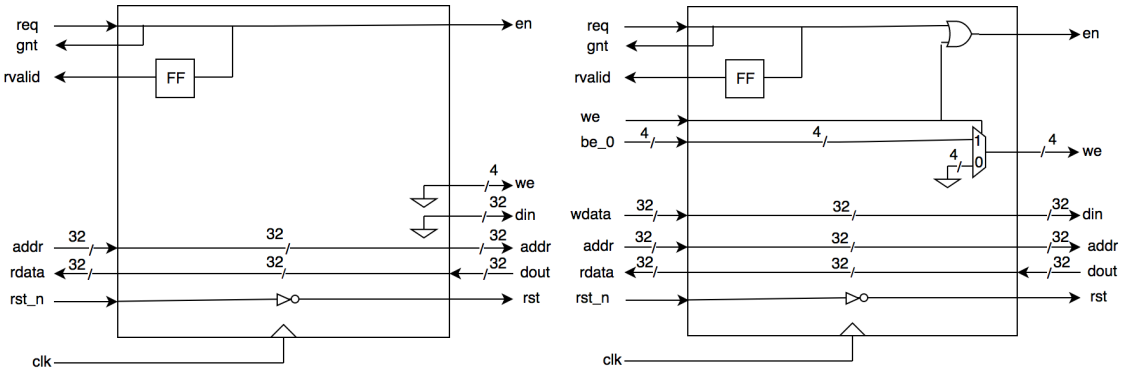
The module was integrated with proprietary IP blocks within the Xilinx Vivado Design Suite. The full system was then deployed on the Xilinx Zynq-7000 AP SoC ZC702 Evaluation Kit. To integrate a core into the FPGA system, two memory adapters were written in Verilog to translate the memory protocol implemented in the core to the one supported by the adopted SRAM IP blocks. A total of two pairs of adapters were written, one for RVXRed and the other for Zero-riscy. Each pair includes an instruction memory adapter and a data memory adapter as shown in Figures 4.2 and 4.3.



(a) RVXRed imem adapter

(b) RVXRed dmem adapter

Figure 4.2: RVXRed memory adapters.



(a) Zero-riscy imem adapter

(b) Zero-riscy dmem adapter

Figure 4.3: Zero-riscy memory adapters.

The RVXRed adapters are of easier comprehension with to respect to Zero-riscy's. In fact, the latter implemented a request-grant memory transaction protocol as the one depicted in Figure 4.4. For any memory transaction, the core starts by providing a valid address and asserting `req`. The memory then answers by setting `gnt` high when it is ready to serve the request, which may happen in the same cycle as the request was sent or any number of cycles later. When ready to provide data for a read request, the memory answers with `rvalid` set high and data on `rdata` (this may happen one or more cycles after the core has received the grant). Since the SRAM IP blocks always satisfies the requested operation in the following clock cycle, the `gnt` signal is directly connected to the `req` line, and `rvalid` is a delayed version of `req`.

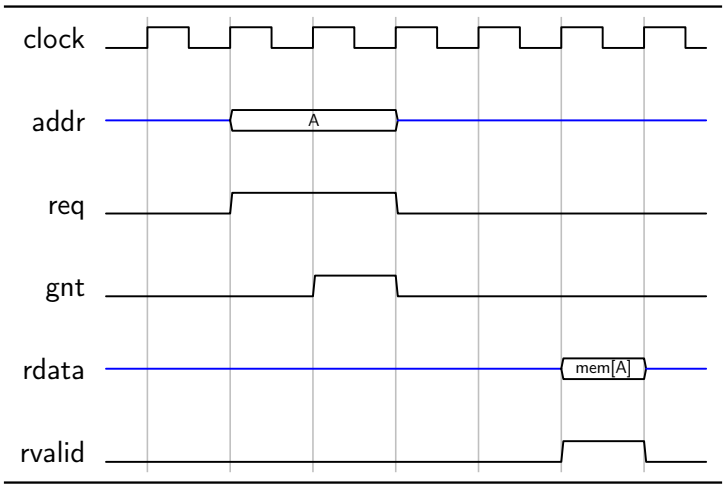


Figure 4.4: Zero-riscy's request-grant memory protocol for a read transaction.

Before packaging a core as an IP block, a final top-level Verilog description was written. This encapsulates the core, the memory adapters and a counter which is in charge of counting the clock cycles since the fetch enable signal has been sent to the core. The purpose of this counter is to have a reliable metric for measuring the duration of execution of a program in the form of clock cycle count, later used to compute the CPU time (a key performance metric in this experimental setup).

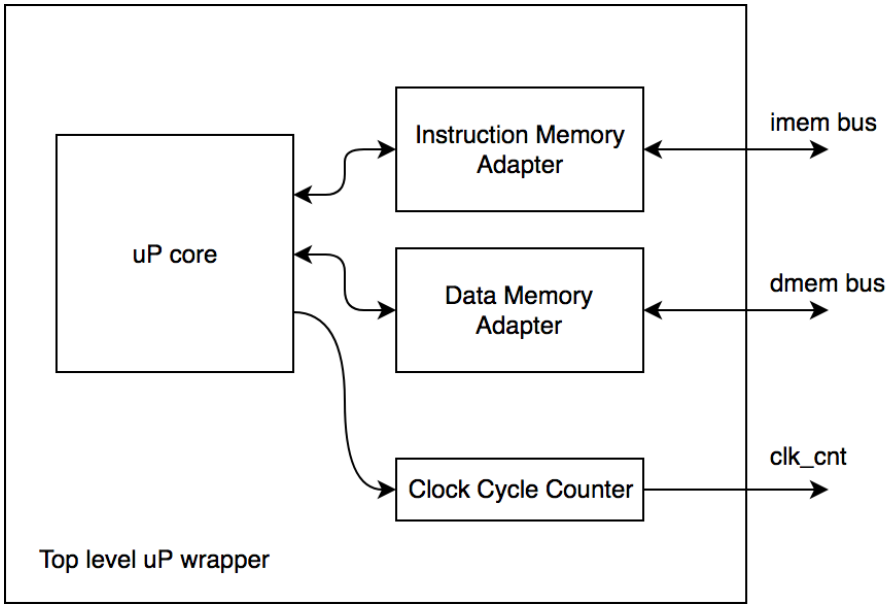


Figure 4.5: Top-level wrapper module.

The core IP module was integrated with the following:

1. ZYNQ7 Processing System: this IP represents the processor contained within the adopted FPGA. A C/C++ program was executed on such processor to initialize, manage and monitor the other blocks via an AXI bus.

2. SRAM modules: two identical blocks were used for instruction and data memories. These have two interfaces, one for the DUT and the other for the ZYNQ7.

3. SRAM AXI controllers: enable interaction between the SRAMs and the ZYNQ7, which initializes the memory contents at startup and dumps them at simulation end for comparing them with a golden model.

4. Core controller: it is in charge of receiving commands from the ZYNQ7 and driving signals for reset and fetch enable to the core. Additionally, it reads the core clock counter and checks whether program execution ended, signaling this event to the ZYNQ7. The functionality of this module was described in C and the IP block was obtained through Xilinx Vivado HLS. This was yet another demonstration of the ease and speed with which HLS tools can be used for hardware design.

5. Logic analyzer: this module is optional as it is used for debugging and tracking signal values on wires between blocks. After debugging, it is removed to get the correct value of the system clock frequency (the high complexity of this module greatly reduces such value).

Figure 4.6 illustrates the full system. Note, the clock tree is not reported for simplicity but it is a single one driven by the ZYNQ7 and shared among all blocks.
The system was synthesized and its FPGA implementation provided automatically by the FPGA software suite, which produced the final bitstream file to be downloaded on the FPGA. For each new implementation of RVXRed generated during DSE, a new IP block was packaged and integrated in the system.
Finally, the program to be run on the ZYNQ was written in C++. This file is shared among all implementations as it is core independent.
The program operates as follows. First, the core controller, data and instructions memories are initialized. Then, the reset signal and fetch enable signals are sent to the controller which relays this information to the DUT. The controller is now polled until it asserts the end of program signal (this occurs whenever the DUT signals the end of the program execution). At this point, data memory contents are compared with the golden model and the clock counter is read and printed to monitor (this value is later used to compute the effective latency of the core, as described in Chapter 5). If using a logic analyzer, any

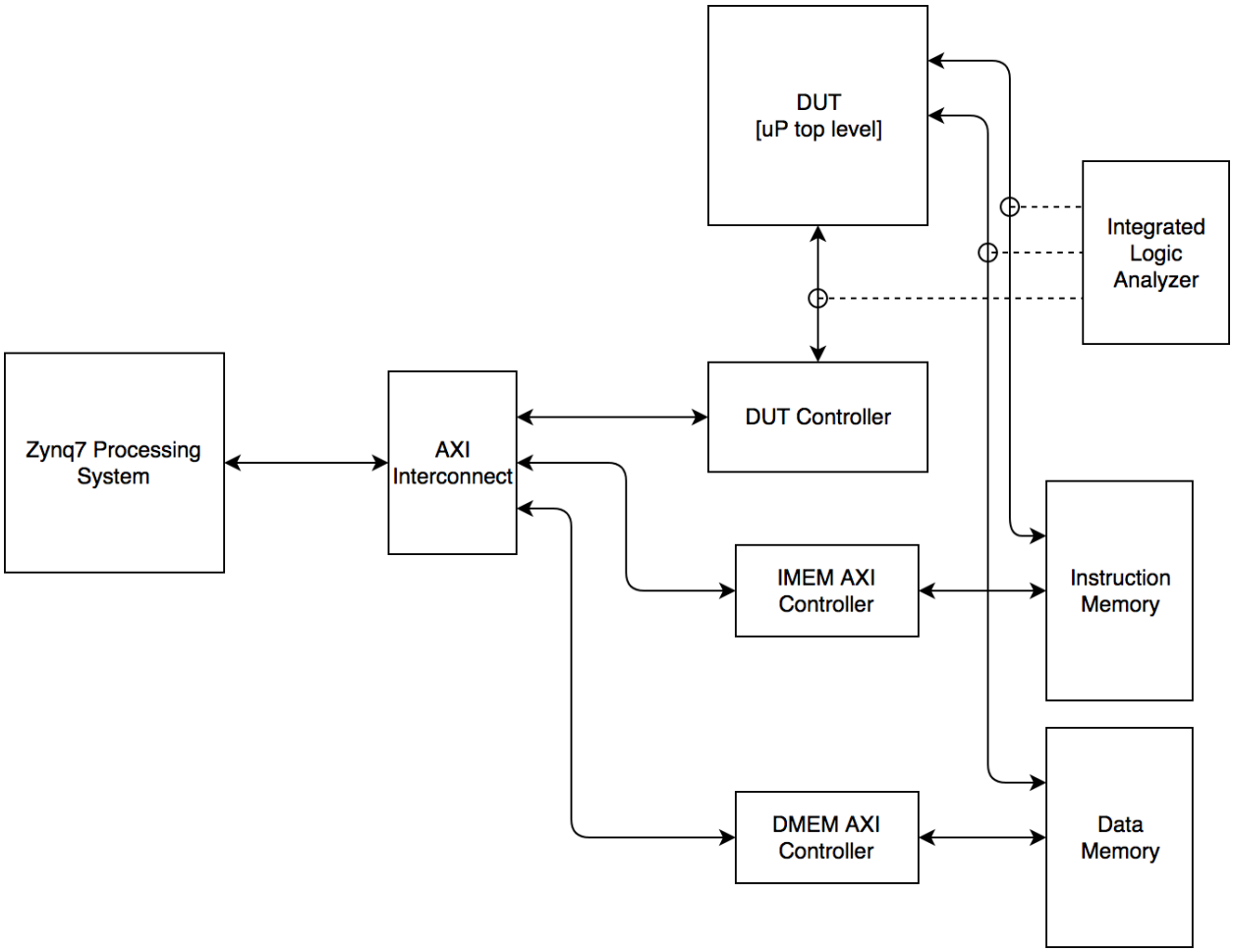


Figure 4.6: Complete FPGA IP block system.

signal between the IP blocks can be observed. This was a useful way to ensure the correct behavior of the DUT, especially in regards to the memories, in a similar fashion to what was done during logic simulation. Listing 4.2 reports a simplified C++ program similar to the one adopted during experiments.

In this setup, whenever a new benchmark must be tested, the designer is simply in charge of changing the value of the `num_insn` variable and the contents of the `program` array. These lines are automatically generated by the script mentioned in Section 4.1, making this process effortless.

```
1  // Enum with commands for the core controller block
2    enum cmd {rst_on = 0, rst_off, go};
3
4    // Pointers to the SRAM AXI controllers
5    unsigned *dmem_ptr = (unsigned*) SRAM_0_BASEADDR;
```

```
6     unsigned *imem_ptr = (unsigned*) SRAM_1_BASEADDR;
7
8     // Core controller instance
9     Core_ctrl doCore_ctrl;
10    Core_ctrl_Config *Core_ctrl_cfg;
11
12  // Initialize the core controller
13    void init_ctrlCore()
14    {
15        int status = 0;
16        Core_ctrl_cfg = Core_ctrl_LookupConfig(CORE_DEVICE_ID);
17        if (Core_ctrl_cfg)
18        {
19            status = Core_ctrl_CfgInitialize(&doCore_ctrl, doCore_ctrl_cfg);
20            if (status != SUCCESS)
21            {
22                printf("ERROR: Failed to initialize core controller\n");
23            }
24        }
25    }
26
27    // Program entry point
28    int main(int argc, char **argv)
29    {
30        init_ctrlCore();
31
32        // Clear data memory content
33        for (unsigned idxX = 0; idxX < len(dmem_ptr); idxX++)
34            dmem_ptr[idxX] = 0;
35
36        // Clear instruction memory content
37        for (unsigned idxX = 0; idxX < len(imem_ptr); idxX++)
38            imem_ptr[idxX] = 0;
39
40        // Fill the temprorary program array
41        unsigned num_insn = 32;
42        unsigned prog[num_insn] =
43            {
44            52429075,
45            //...,
46            4294967295
47            };
48
49        // Load program into instruction memory
```

```
50          for (unsigned idxX = 0; idxX < num_insn; idxX++)
51              imem_ptr[idxX] = prog[idxX];
52
53          // Reset the DUT
54          Core_ctrl_Set_axi_cmd(&doCore_ctrl, rst_on);
55          usleep(1);
56
57          // Remove reset from DUT
58          Core_ctrl_Set_axi_cmd(&doCore_ctrl, rst_off);
59          usleep(1);
60
61          // Send the fetch enable signal
62          Core_ctrl_Set_axi_cmd(&doCore_ctrl, go);
63
64          // Wait until the core has finished executing the provided program
65          while (!Core_ctrl_Get_end_of_prog(&doCore_ctrl));
66
67          // Dump clock counter
68          printf("Clock counter: %u\n", clk_cnt);
69
70          // Dump data memory
71          for (unsigned idx = 0; idx < len(dmem_ptr); idx++)
72              printf("Data memory: index=%d value=%d\n", idx, dmem_ptr[idx]);
73
74          // Dump instruction memory
75          for (unsigned idx = 0; idx < len(imem_ptr); idx++)
76              printf("Instruction memory: index=%d value=%u\n", idx, imem_ptr[
    idx]);
77
78          return(0);
79      }
```

Listing 4.2: C++ program for FPGA system verification.

4.3 Test Programs

To better understand and compare the speed of each implementation, three programs were written and then loaded into the instruction memory. These are:

1. 1D CONV: one-dimension convolution;

2. 2D CONV: two-dimension convolution;

3. HIST EQ: histogram equalization.

Listings 4.3, 4.4 and 4.5 report code snippets for each program. 1D CONV and 2D CONV perform MAC operations on vectors and matrices respectively; while HIST EQ includes several matrix operations, including a loop with a division, which yields interesting results when run by processor implementations with an optimized division algorithm (Section 5.2).

```
1   for(int i = 0; i < samples; i++){
2     y[i] = 0; // reset before MAC
3     for(int j = 0; j < kernels; j++){
4       y[i] += x[i - j] * h[j];   // MAC
5     }
6   }
```

Listing 4.3: 1D CONV

```
1
2   kern_center_X = kern_cols / 2;
3   kern_center_Y = kern_rows / 2;
4
5   for(int i=0; i < rows; i++){   // rows
6     for(j=0; j < cols; j++){   // columns
7       for(int h = 0; h < kern_rows; h++){ // kernel rows
8         hh = kern_rows - 1 - h;
9         for(int l = 0; l < kern_cols; l++){ // kernel columns
10          ll = kern_cols - 1 - l;
11          ii = i + (h - kern_center_Y);
12          jj = j + (l - kern_center_X);
13
14          if( ii >= 0 && ii < rows && jj >= 0 && jj < cols )
15            out[i][j] += in[ii][jj] * kernel[hh][ll]; // MAC
16        }
17      }
18    }
19  }
```

Listing 4.4: 2D CONV

```
1   for(int i = 0; i < 256; i++){ // build probability table
2     prob_tab[i] = histogram[i]/pixels;
```

```
3     }
4
5     cdf[0]=prob_tab[0]; // cdf: cumulative distribution function
6     for(int i = 1; i<256; i++){
7       cdf[i] = prt[i] + cdf[i-1];
8       if(cdf[i] > cdfmax)
9         cdfmax = cdf[i];
10      if(cdf[i] < cdfmin)
11        cdfmin = cdf[i];
12    }
13
14    for (int i=0; i < lines;i++){ // final image
15      for(int j = 0; j < columns; j++){
16        image_out[i][j] = cdf[image_in[i][j]] * 255;
17      }
18    }
```

Listing 4.5: HIST EQ

4.4 CPU Time as a Performance Metric

For each program execution, the clock cycle count (CLK_CNT) was extracted and used to compute the CPU time. Consider equation 2.1 from Section 2.1.1.
Note that:

$$avg\_CPI = \frac{CLK\_CNT}{NCI} \tag{4.1}$$

And so we obtain an alternative expression to easily compute the CPU Time:

$$CPU\ Time = CP * CLK\_CNT \tag{4.2}$$

This equation is used in Chapter 5, which reports numerical results gathered during the experimental process described in this chapter and draws conclusions regarding a collection of microprocessor implementations.

Chapter 5

Evaluation and Results

The focus of this chapter is on the results obtained with the FPGA and CMOS experiments. These numerical indicators are useful for comparing the RVXRed versions with Zero-riscy. Additional measures and characteristics must be taken into consideration for a more exhaustive comparison. For such reason, Section 5.3 tries to compare the effort required by developing a processor at the system and register transfer levels.
As previously mentioned, both processor architectures support the RV32I and RV32M instruction subsets (listed in Appendix A). Zero-riscy also supports the execution of the RV32C subset, a compressed version (16-bit long instructions) of the RV32I instructions.

Before deploying the target processor to FPGA or performing logic synthesis, the adopted HLS tool has proven to be effective in indicating characteristics of the architecture it was synthesizing. In fact, it provided relevant details such as the achievable clock frequency, area distributions and the CDFG associated to the input description. These were useful indicators for exploring a varied collection of configurations, which included different HLS directives and also diverse coding styles. In an iterative fashion, results were compared with previous implementations in order to find the configurations that yielded the best Quality of Results (QoR). Finding the optimal coding style has proven to be quite time consuming, but once discovered, a guideline can be drafted and used in later projects.

5.1 FPGA Implementation

All processor cores were synthesized and implemented on a Xilinx Virtex 7 FPGA (model identifier "xc7z020clg484-1"). Table 5.1 reports the achievable frequency and resource utilization values for each solution.
The basic logic element in the Series 7 FPGAs by Xilinx is a Configurable Logic Block

(CLB). Each CLB contains two slices and is connected to the interconnection matrix via a dedicated switch. Slices are the fundamental resource and most importantly include Look-Up Tables (LUTs) and registers. The former can be used to implement combinational logic and combined with the latter to form a sequential resource. Additionally, multiplexers are embedded within slices to route internal signals. Special resources, known as DSP slices are high-speed blocks which include, among others, circuitry for multiplication.

It is clear that Zero-riscy dominates all RVXRed implementations under all aspects for the FPGA design (clock frequency and resource utilization). The main reason is that the adopted HLS tool is not apt for FPGA design, and only in future releases will be capable of delivering better results. Nonetheless, following the FPGA design flow was necessary to obtain the clock cycle count used to compute the CPU time for each test program (Section 5.2).

	RVXRed BASIC	**Zero-riscy**	**RVXRed ASAP**	**RVXRed UNDIV2**	**RVXRed UNDIV4**
Frequency (MHz)	55	70	60	50	40
Slice	1790	977	1877	1916	2159
Slice LUTs	4521	2390	4699	5325	6080
Slice Registers	3944	1529	3974	3899	3932
F7 Muxes	323	281	304	426	358
F8 Muxes	4	0	5	90	90
LUT-FF Pairs	1853	404	1855	1796	1778
DSPs	3	1	3	3	3

Table 5.1: FPGA clock frequency and resource utilizations

5.2 CMOS Implementation

For the CMOS implementation, each core was synthesized with Synopsys Design Compiler using a commercial 32 nm CMOS library. Table 5.2 lists the achievable clock frequency and area occupation values for each solution. All RVXRed solutions have a larger footprint than Zero-riscy, mainly, this is due to the overhead introduced by describing the RVXRed cores at a higher level of abstraction. Designing the processor in an RTL language enables a finer tuning and control over the architecture with respect to a high-level description. Nevertheless, different coding styles as well as new HLS tools and updates may yield better results. ASAP and Zero-riscy have the highest achievable clock frequency with a value of 2 GHz, with BASIC, UNDIV2 and UNDIV4 following. The comparison of Tables 5.1 and 5.2 clearly indicate that the adopted HLS tool performs better for CMOS implementations. Figure 5.1 reports CPU time versus area plots for the test programs.

	RVXRed BASIC	**Zero-riscy**	**RVXRed ASAP**	**RVXRed UNDIV2**	**RVXRed UNDIV4**
Clock Frequency (GHz)	1.9	2	2	1.8	1.8
Area (μm^2)	17789	12948	17565	19022	19912

Table 5.2: CMOS clock frequency and area occupation

	RVXRed BASIC	**Zero-riscy**	**RVXRed ASAP**	**RVXRed UNDIV2**	**RVXRed UNDIV4**
Latency	32	32	32	16	8
Throughput	1/32	1/32	1/32	1/16	1/8

Table 5.3: Division characteristics for all implementations.

The plots show that although always dominated in terms of area occupation, some RVXRed solutions give the lowest values of CPU time for the given set of test programs. For 1D CONV and 2D CONV, all RVXRed solutions are faster. These programs mainly perform multiply-accumulate (MAC) operations. The C++ `*` operator is broken down into the RISC-V assembly instruction sequence reported in Listing 5.1.

```
1    mulh x3, x2, x1   # stores in x3 the upper 32 bits of the result
2    mul  x4, x2, x1   # stores in x4 the lower 32 bits of the result
3    add  x5, x5, x3
4    add  x6, x6, x4
```

Listing 5.1: MAC RISC-V assembly code snippet.

In all implementations, the `mul` instruction is executed by a multiplication circuit that returns the result in 1 clock cycle (CC). On the contrary, `mulh` requires 4 CCs in Zero-riscy's multiplier and only 1 CC in any RVXRed version.

This means that the above instructions require 7 CCs to be executed by Zero-riscy, while only 4 CCs by RVXRed, and thus the value of the clock count when executing the convolution programs, which perform many multiplications, is lower in the RVXRed implementations.

As mentioned in Section 4.3, histogram equalization, among other operations, performs a division in the body of a loop which iterates over all points of the given histogram.

The drawbacks of the UNDIV2 and UNDIV4 architectures are a higher area occupation and lower clock frequency. Nonetheless, they employ less time to perform divisions and, in a division-oriented algorithm like the chosen one, they are the clear winners in terms of CPU time.

To conclude, Zero-riscy is the winning solution for area occupancy, but some RVXRed versions yield better values for CPU time and take part of the Pareto set, while others

remain dominated with respect to both metrics.

5.3 Qualitative Results: Lines of Code

It should be noted that the effort and time required by the proposed design activity is notably less than that involved into the traditional RTL design flow. Let alone, the ease with which different implementations were obtained in the DSE phase.
These aspects are hardly measured in numerical terms, however one indicator can be the lines of code (LOC). Table 5.4 shows a clear difference in LOC between the SystemC (RVXRed) and Verilog (Zero-riscy) designs. This was a predictable result due to raising the level of abstraction, and thus omitting several details that must be written explicitly when using RTL languages. For completeness, Table 5.5, instead, reports the LOC considering the Verilog RTL code that was automatically generated starting from the SystemC files.

	RVXRed BASIC	**Zero-riscy**	**RVXRed ASAP**	**RVXRed UNDIV2**	**RVXRed UNDIV4**
LOC	2042	6205	2042	2042	2042

Table 5.4: Comparing LOC, considering the manually written SystemC code for the RVXRed versions.

	RVXRed BASIC	**Zero-riscy**	**RVXRed ASAP**	**RVXRed UNDIV2**	**RVXRed UNDIV4**
LOC	17900	6205	17692	18541	18740

Table 5.5: Comparing LOC, considering the automatically generated Verilog RTL code for the RVXRed versions.

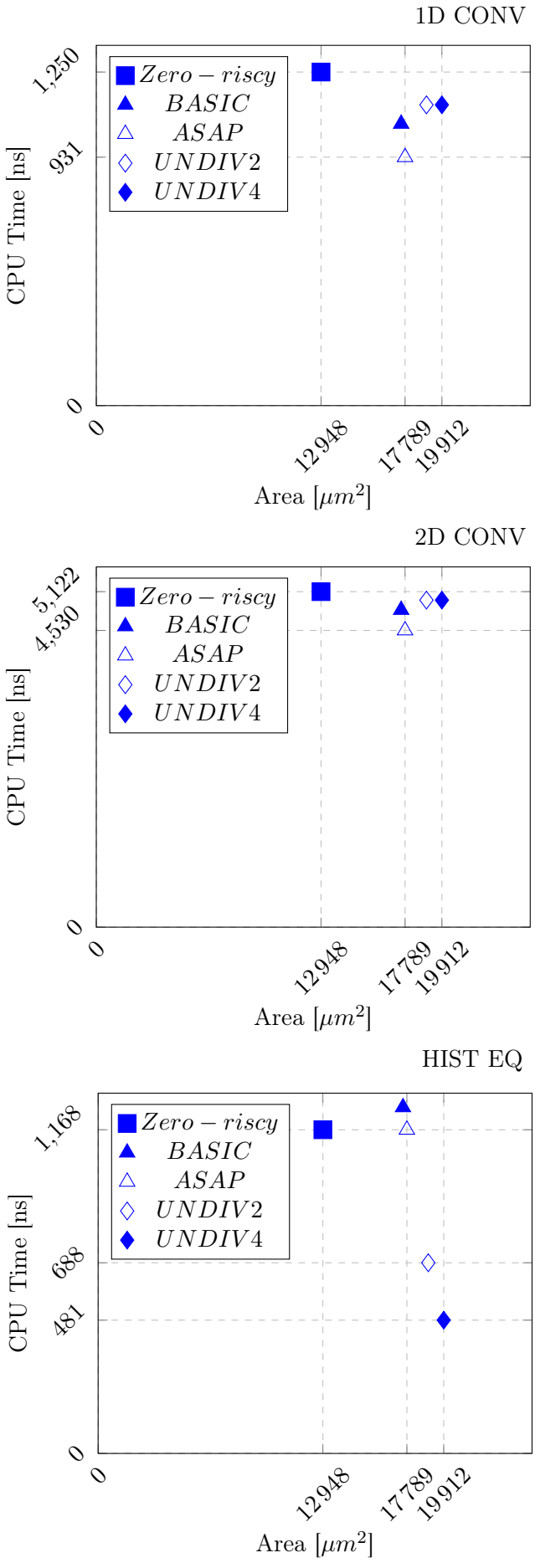


Figure 5.1: CPU Time vs Area plots

Chapter 6

Conclusion

6.1 Achievements

A 5-stage pipelined 32-bit instruction set processor compatible with the RISC-V ISA has been implemented in a high-level language and synthesized with a commercial HLS tool. Design options and issues have been analyzed and overcome, leading to a new methodology for microprocessor design at the system level. The proposed core has been validated following steps in an experimental setup which included: logic simulation, logic synthesis and FPGA verification. Finally, several implementations were obtained starting from a single SystemC description and were compared with a modern RISC-V core supporting the same instruction set. The outcome of these comparisons clearly reveals that the proposed approach yields significant results and clears the path for future developments which will adopt this methodology.

6.2 Future Works

Many enhancements may be applied and introduced in the proposed core.
At an architectural level, data forwarding schemes, exception handling techniques, out-of-order execution, among other features, should be implemented.
Additionally, the core can be extended to support other RISC-V subsets, such as the RV64I, RV128I and the F/D/Q extensions for floating point operations. The modularity of the proposed architecture easily enables such additions.
Furthermore, solutions and results provided by other commercial HLS tools could be investigated and compared with the ones listed in this report.

Appendix A

RVXRed Instruction Set

Integer Register-Register	Integer Register-Immediate	Control Transfer	Load and Store	System
add	addi	beq	sb	csrrw
slt	slti	bne	sh	csrrs
sltu	sltiu	blt	sw	csrrc
sll	slli	bge	lb	csrrwi
srl	srli	bltu	lh	csrrsi
sra	srai	bgeu	lw	csrrci
or	ori	bgeu	lbu	ecall
and	andi	jalr	lhu	ebreak
xor	xori	jal		
sub	lui			
	auipc			

Table A.1: RV32I instruction subset

Multiplication	Division
mul	div
mulh	divu
mulhu	rem
mulhsu	remu

Table A.2: RV32M instruction subset

Bibliography

[1] Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. *High-Level Synthesis of Accelerators in Embedded Scalable Platforms.* Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC), 2016.

[2] Yen-Ju Lu et al. *Microprocessor Modeling and Simulation with SystemC.* International Symposium on VLSI Design, Automation and Test, 2007. VLSI-DAT 2007., 2007.

[3] G. De Micheli and D.C. Ku. *HERCULES-a system for high-level synthesis.* Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE, 1988.

[4] Gordon Moore. *Cramming more components onto integrated circuits* . Electron. Mag 38(8), 1965.

[5] ITRS. *The International Technology Roadmap for Semiconductors, 2009 Edition.* International SEMATECH: Austin, TX 2009, 2009.

[6] Andrew B. Kahng. *The ITRS design technology and system drivers roadmap: process and status.* Proceedings of the 50th Annual Design Automation Conference (DAC), Article No. 34, 2013.

[7] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Fifth Edition.* Morgan Kaufmann Publishers, 2013.

[8] Andrew Waterman et al. *The RISC-V instruction set.* Hot Chips 25 Symposium (HCS), 2013 IEEE, 2013.

[9] Yunsup Lee et al. *An Agile Approach to Building RISC-V Microprocessors.* IEEE Micro (Volume: 36, Issue: 2), 2016.

[10] Andreas Traber and Michael Gautschi. *PULPino: Datasheet.* ETH Zurich and University of Bologna, 2016.

[11] Pasquale Davide Schiavone. *zero-riscy: User Manual.* University of Bologna and ETH Zurich, 2017.

[12] Andrew Waterman Yunsup Lee, David Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.1.* University of California, Berkeley, 2016.

[13] Andrew Waterman, Krste Asanovic, and SiFive Inc. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Privileged Architecture Version 1.10.* University of California, Berkeley, 2017.

[14] Mentor Graphics, Inc. *Google Develops WebM Video Decompression Hardware IP Using High-Level Synthesis.* Mentor Graphics, Inc, 2015.

[15] Mentor Graphics, Inc. *Bosch Visiontec Rapidly Brings New Automotive IP to Market Using the Catapult HLS Platform.* Mentor Graphics, Inc, 2015.

[16] Andrew Putnam et al. *BA Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services.* 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014.

[17] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit.* 44th International Symposium on Computer Architecture (ISCA), 2017.

[18] D. Aarno and J. Engblom. *Software and System Development using Virtual Platforms.* Morgan Kaufmann Publishers, 2014.

[19] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up, Second Edition.* Springer, 2014.

[20] Luca P. Carloni. *From Latency-Insensitive Design to Communication-Based System-Level Design.* Proceedings of the IEEE, vol. 103, no. 11, 2015.

[21] Cadence Design Systems, Inc. *Stratus High-Level Synthesis User Guide.* Cadence Design Systems, Inc, 2017.

[22] Cadence Design Systems, Inc. *Stratus High-Level Synthesis Reference Guide.* Cadence Design Systems, Inc, 2017.

[23] Luca P. Carloni. *The Role of Back-Pressure in Implementing Latency-Insensitive Design.* Second International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Architectures (FMGALS '05), 2006.

[24] Luca P. Carloni. *The Case for Embedded Scalable Platforms.* Proceedings of the Design Automation Conference (DAC), 2016.

[25] Christian Pilato, Qirui Xu, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. *On the Design of Scalable and Reusable Accelerators for Big Data Applications.* Proceedings of the International Conference on Computing Frontiers (CF), 2016.

[26] Andreas Traber. *RI5CY Core: Datasheet.* ETH Zurich and University of Bologna, 2016.