TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

A MASTER THESIS ON

FRANCIS-V: FRAmework for iNtegrating Custom InstructionS into RISC-V systems

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Diplom-Ingenieur

(Equivalent to Master of Science)

in

Embedded Systems (066 504)

by

Florian Egert

11706012

Supervisor(s):

Projektass.in Dip.-Eng. Sofia Maragkou

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

Vienna, Austria

October 2023

Abstract

Embedded systems are essential building blocks in nearly every aspect of today's life. Simultaneously to increasingly tight design requirements, the steady growth of application fields raises the need for versatile systems. Application Specific Instruction Set Processors (ASIPs) offer an efficiency and versatility trade-off by providing a flexible base instruction set extensible with application-specific Custom Instructions (CIs). The emerging open-source architecture RISC-V introduces new challenges to the topic. The RISC-V ecosystem requires tools for automating the tedious CI development process. In particular, there is a lack of flexible methodologies to facilitate the integration of compatible and reusable components.

This thesis proposes FRANCIS-V, an integration framework to reduce the effort required for designing CI-based systems. Its main contribution is a flexible interfacing methodology for coprocessor generation based on the OpenHW Group's CORE-V eXtension Interface (XIF), complemented by a predefined RISC-V processor system as well as compilation, simulation, and verification features. The thesis further proposes a comprehensive analysis of the CI development process, recognizing CI identification, hardware generation, integration, and verification as its major challenges.

An expert estimation is performed to evaluate the potential development time reduced by utilizing FRANCIS-V, yielding a reduction of around 16 workdays or 34% of saved total time. The generated systems are further evaluated based on two use cases, AES and CRC. The evaluation yields a considerable cycle count decrease of up to 87.53% and a marginal LUT and FF utilization overhead of 0.85% and 2.75%, respectively, positioning the framework as competitive to comparable tools. The generated coprocessors support a broad range of CIs and are compatible with XIF-based RISC-V processors.

FRANCIS-V contributes to the identified challenges of CI integration and verification, and it is further designed for collaboration with prospective identification and hardware generation solutions. It significantly improves the CI development process for designers with varying expertise. The thesis emphasizes the need for flexible and standardized RISC-V CI interfacing solutions and motivates further work on CI integration and overall CI design automation.

Kurzfassung

Embedded Systems sind ein unverzichtbarer Bestandteil in nahezu allen Bereichen des heutigen Lebens. Während die Designanforderungen stetig steigen, bedingen wachsende Einsatzfelder eine gewisse Vielseitigkeit dieser Systeme. Application Specific Instruction Set Processors (ASIPs) besitzen ein solches vielseitiges Grundset an Instruktionen, welches zur Effizienzsteigerung mit anwendungsspezifischen Custom Instructions (CIs) erweitert werden kann. Die aufstrebende Open-Source-Architektur RISC-V belebt dieses Thema mit neuen Herausforderungen. Die RISC-V-Community benötigt Tools zur Automatisierung des aufwendigen CI-Entwicklungsprozesses, vor allem, um dem Mangel an flexiblen Integrationslösungen für kompatible und wiederverwendbare Komponenten entgegenzuwirken.

In dieser Arbeit wird FRANCIS-V vorgestellt, ein Framework zur Verringerung des Integrationsaufwands von CI-basierten Systemen. Das Framework generiert Coprozessoren basierend auf dem CORE-V eXtension Interface (XIF) der OpenHW Group und stellt ein zugehöriges RISC-V-Prozessorsystem, einen Compiler, einen Simulator, und Verifikationsfunktionen zur Verfügung. Weiters wird eine umfassende Analyse des CI-Entwicklungsprozesses präsentiert und Auswahl, Hardwaregenerierung, Integration und Verifikation von CIs als dessen herausforderndste Teilaufgaben identifiziert.

Um die Zeit zu ermitteln, die durch den Einsatz von FRANCIS-V im Entwicklungsprozess gewonnen werden kann, wurden Expertenmeinungen eingeholt. Die Auswertung ergibt eine Zeitersparnis von etwa 16 Werktagen oder 34% der gesamten Entwicklungszeit. Die durch das Framework generierten Systeme werden weiters auf Basis der Anwendungsfälle AES und CRC beurteilt. Bei der Evaluierung ergibt sich eine deutliche Reduktion der benötigten Taktzyklen um bis zu 87.53% und nur ein geringfügiger LUT- und FF-Utilization-Overhead von 0.85% und 2.75%, wodurch FRANCIS-V ähnliche Werte gegenüber vergleichbaren Tools erzielt. Die generierten Coprozessoren unterstützen ein vielfältiges Spektrum an CIs und sind kompatibel zu XIF-basierten RISC-V Prozessoren.

In dieser Arbeit wird die Notwendigkeit von flexiblen und standardisierten RISC-V CI Interfaces betont und die Weiterentwicklung von CI-Integration und der übergeordneten CI-Entwicklungsautomatisierung motiviert. FRANCIS-V trägt zur CI-Integration und Verifikation bei und ist im Hinblick auf ein Zusammenspiel mit zukünftigen Identifikations- und Hardwaregenerierungs-Tools konzipiert. Sowohl unerfahrene als auch versierte Entwickler können von dem optimierten CI-Entwicklungsprozess profitieren.

Preface

This thesis was written in collaboration with Siemens Technology. Parts of the thesis were funded by the EU project TRISTAN. I am very grateful for the provided resources and supervision, which greatly contributed to the quality of this research.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Copyright Statement

I, Florian Egert, hereby declare that this thesis is my own original work and, to the best of my knowledge and belief, it does not:

- Breach copyright or other intellectual property rights of a third party.
- Contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
- Contain material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.
- Contain substantial portions of third party copyright material, including but not limited to charts, diagrams, graphs, photographs or maps, or in instances where it does, I have obtained permission to use such material and allow it to be made accessible worldwide via the Internet.

Signature: _____

Vienna, Austria, October 2023 Florian Egert

Acknowledgment

Meine akademische Laufbahn erreicht mit dieser Arbeit ihren bisherigen Höhepunkt. Ohne Begleitung von Kolleg:innen, Freund:innen, und Familie auf diesem Weg wäre ich nie so weit gekommen. Ich möchte euch allen von Herzen danken und die nächsten Zeilen vor allem folgenden Personen widmen.

- Meiner Familie Birgit, Christian, Katharina, Erwin, Claudia, Michael, Gerlinde und Christian, die mir die letzten 26 Jahre emotional, kulinarisch und finanziell beigestanden sind. Ihr habt während meiner Studienzeit vermutlich weit mehr durchgemacht als ich selbst und hattet dennoch immer ein offenes Ohr und Rat für jegliche Art von plötzlich auftretenden Problemen.

- Meiner Partnerin Bianca, die mir die Motivation, Kraft und Inspiration verlieh, um diese Arbeit überhaupt zu vollenden. Du scheinst genau zu wissen, wann ich Ansporn oder eher Ablenkung brauche, und überraschst mich nach 7 Jahren immer noch täglich.

- Meinen Freund:innen aus Kalladorf, Guntersdorf, den diversesten anderen Dörfern, sowie aus Wien und überall. Obwohl ich oft lange Studientage priorisieren musste, gab es dennoch stets Raum für ausgleichende Spikeball-, Wander- und Brettspieltage sowie für einschlägige Abende und Nächte.

- Meinem Siemens-Team, allen voran Hannes, Martin, Herbert, und den Interviewteilnehmer:innen sowohl für die fachliche Unterstützung als auch die herzliche Aufnahme in euer Team vom ersten Tag an. Diese Arbeit wäre nicht ohne eure weitreichende Expertise zustandegekommen.

- My fellow students and ICT colleagues who have accompanied me through the demanding yet often amusing academic time. In our bachelor years we learned a lot from each other, whether in Prof. Stefan's lectures or during highly intellectual conversations in ski attire. The master years were full of yoga, pizza, olympic-level bouldering, butter chicken, and cake[1]. I will miss solving academic challenges together nearly as much as venting on unrealistic submission deadlines.

[1] Did we eat too much?

Last but not least, I wish to express my deep gratitude to my superivsors and reviewers.

- Hannes, thank you for our technical discussions and for seemingly having an answer ready for every problem occuring during all phases of my master thesis.
- Thank you Martin for all the tips and advice for my work during the year(s), and for providing the setting for its final phases.
- Markus, thank you for coordinating the initial setup of this work and for all subsequent meetings and suggestions to improve my strategy, presentation, and writing.
- Thank you Axel Jantsch for your scientific guidance and for providing an inviting environment that enables fruitful discussions and a professional exchange.
- Finally, thank you Sofia. You likely spent way too much thoughts on this work than is considered healthy. You never lost your patience during our countless discussions, reviews, rehearsals, and food breaks. I could go on about how much you helped me refine my thesis (multiple times). Since you would have to tediously review that too, however, I will leave it at:

Thank you. Danke.
Florian

Contents

List of Tables

List of Figures

Acronyms

AES Advanced Encryption Standard. iii, 2, 6, 14, 16, 19, 20, 21, 24, 32, 37, 39, 41, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 62, 63, 64, 65, 69, 71, 73, 74, 79

AI Artificial Intelligence. 71

ALU Arithmetic Logic Unit. 9, 10

ASIC Application Specific Integrated Circuit. 1, 8, 27, 38, 52, 71

ASIP Application Specific Instruction Set Processor. iii, iv, 1, 15, 16, 17, 19

BRAM Block RAM. 52, 65

CI Custom Instruction. iii, iv, v, 1, 2, 3, 6, 7, 8, 10, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 59, 60, 61, 62, 63, 64, 65, 66, 69, 70, 71, 72, 73, 74, 75, 76, 77, 79, 80

CRC Cyclic Redundancy Check. iii, 2, 19, 32, 37, 41, 42, 43, 47, 48, 49, 53, 54, 55, 56, 57, 59, 66, 67, 71, 74, 79

FF Flip Flop. 65, 73, 74, 79

FIFO First In First Out. 25, 45

FPGA Field-Programmable Gate Array. 8, 27, 37, 38, 46, 52, 65, 71

GCC GNU Compiler Collection. 42, 60

GPP General Purpose Processor. 1

HDL Hardware Description Language. 18, 19, 22, 29, 31, 37, 46, 47, 48, 51, 52, 56, 60, 62, 71

HLS High-Level Synthesis. 19, 21, 29, 31, 34, 60, 69, 72, 76, 79, 80

IP Intellectual Property. 7, 8

ISA Instruction Set Architecture. 2, 5, 6, 7, 8, 10, 19, 30, 31, 37, 42, 51

ISE Instruction Set Extension. 2, 6, 10, 14, 15, 16, 19, 34, 49

ISS Instruction Set Simulator. 18, 35, 51, 62, 76

LSU Load/Store Unit. 9, 24, 25

LUT Lookup Table. 65, 73, 74, 79

NFC Near Field Communication. 16, 24, 71

OBI Open Bus Interface specification. 9, 46

PMA Physical Memory Attribution. 8, 9

PQC Post-Quantum Cryptography. 16, 24, 71

RAM Random Access Memory. 52

RTL Register Transfer Level. 2, 8, 18, 19, 29, 30, 32, 34, 37, 43, 45, 46, 47, 49, 54, 55, 56, 58, 62

SHA Secure Hash Algorithm. 19, 71

SoC System on Chip. 5, 38, 39, 46

UVM Universal Verification Methodology. 8

VHDL Very High Speed Integrated Circuit Hardware Description Language. 43, 46

XIF CORE-V eXtension Interface. iii, iv, 2, 3, 7, 8, 10, 11, 12, 15, 16, 18, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 34, 37, 38, 39, 43, 44, 45, 46, 51, 52, 53, 54, 55, 60, 61, 62, 65, 66, 67, 70, 73, 74, 75, 76, 77, 79, 80

Chapter 1

Introduction

With embedded systems and their applications becoming increasingly complex, there is an ever-growing need for efficient solutions and optimized design methodologies. Today's state-of-the-art implementations must be optimized to meet the particular requirements of specialized applications. Established *General Purpose Processors (GPPs)*, with their entailed overhead, are too generic to exploit application-specific properties sufficiently. Thus, GPPs do not align with such tight design constraints. Highly specialized components based on *Application Specific Integrated Circuits (ASICs)*, by contrast, often lack the required adaptability and reusability for utilization in various designs and applications. In addition, developing ASICs is costly and requires experienced designers [1], [2].

Application Specific Instruction Set Processors (ASIPs) provide a trade-off between flexibility and efficiency. Similar to GPPs, such processors offer instructions to cover basic functionality. In addition, they support application-specific *Custom Instructions (CIs)* for more efficiency [1]. However, the ASIP design process partly inherits the increased time and effort of its ASIC counterpart. Designing and implementing CIs for given applications starts with identifying and selecting suitable instructions. It further includes implementing the custom functionality on hardware and integrating it with the processor and required system. Last, the resulting design has to pass verification. During these design tasks, the designer must modify the required C compilers, simulators, and synthesis tools for use with the introduced CIs [3]. Hence, the manual conduction of this process is tedious and error-prone. Automating the design process and integrating suitable instruction identification as well as software and hardware development into one design flow would significantly reduce its design time and effort.

Despite the substantial body of literature on CI design process automation [3], the lack of tools for emerging architectures and the growing interest in open-source solutions motivates further research on this topic. In recent years, there has been a particular interest in *RISC-V*, an emerging open-source Instruction Set Architecture (ISA). RISC-V aims to become a standard architecture in research and industry and a free alternative to closed-source solutions [4]. One main benefit of such an open-source ISA over commercial solutions is the absence of license fees and secrecy, enabling a growing open-source community to use and contribute to the RISC-V ecosystem. Furthermore, the base ISA can be extended with optional standard and non-standard Instruction Set Extensions (ISEs), making RISC-V a promising candidate for implementing CIs.

Current approaches for automated ISEs in RISC-V-based systems largely focus on one specific core or a small set of cores [5], [6]. The open-source community lacks a standard solution that works with several RISC-V cores. A first step towards such a solution would be a standard RISC-V interface to integrate CI logic in external coprocessors that are subsequently compatible with a broad range of RISC-V processors. A promising candidate for such an interface is the *CORE-V eXtension Interface (XIF)*. XIF enables ISEs in a standardized manner without modifying the RISC-V core's RTL representation [7]. However, the interface is currently under development and, at the time of writing this thesis, lacks a standard implementation or methodology for extending coprocessors with interface support.

1.1 Objectives

The aim of this thesis is twofold. First, motivated by the current lack of compatible and reusable RISC-V CI solutions, we analyze the development process of CI-based systems and propose potential approaches to automating its individual steps. In the second part of this work, we focus on CI integration with interfaces. We present a methodology for an automated integration of CI logic into RISC-V cores. The methodology supports various applications and processors. Its core is an adjustable XIF-based coprocessor or *wrapper* for given CI modules. Based on this methodology, we propose the CI integration framework FRANCIS-V, capable of performing the wrapper generation and integration into a predefined RISC-V system alongside code compilation, simulation, and verification of the generated design.

The framework's applicability for a wide variety of application fields is demonstrated with two selected use cases, AES and CRC. The major goal of FRANCIS-V is to reduce the design time in an otherwise expensive manual development process solely conducted by the designer. The generated CI-based systems aim at enhancing the application's performance with low hardware overhead.

1.2 Research questions

This thesis aims to answer the following research questions:

1. Which are the most challenging process steps in a typical Custom Instruction development process regarding time and effort?
2. How can the manual effort of integrating Custom Instructions with RISC-V cores be reduced using the CORE-V eXtension Interface?
3. How much design time can a designer save with the developed framework depending on their experience level and use case?

1.3 Structure of the thesis

The thesis starts with background on RISC-V and the CORE-V Family in *Chapter 2*. *Chapter 3* presents the state of the art on CI integration in RISC-V processors and goes into detail on CI interfaces and existing work utilizing XIF.

Chapter 4 proposes the methodology to answer the research questions. The chapter analyzes the CI development process and potential automation approaches and further proposes FRANCIS-V. In addition, it presents the thesis' approach to evaluate and discuss the potential time a designer can save when utilizing FRANCIS-V during their development process.

Chapter 5 provides implementation details of FRANCIS-V and presents the selected use cases and setup for evaluating the generated designs. The results of the thesis are presented in *Chapter 6*, namely, the analysis of the saved design time as well as of system metrics to assess the designs generated by the framework. *Chapter 7* discusses the main findings of the thesis, compares the results to the related work, and proposes potential approaches for future work on the topic. *Chapter 8* closes with a summary of the work, contributions, and main insights of the thesis.

Chapter 2

RISC-V

RISC-V is an open-source ISA aimed at becoming a standard architecture in research and industry and an open and free alternative to commercial solutions. The project started in 2010 at UC Berkeley. The RISC-V Foundation was established in 2015 to build an open and collaborative community around the RISC-V ISA. In 2020, the organization was named RISC-V International Association and incorporated in Switzerland. Its members are academic institutions, commercial and non-profit organizations, and individuals [8]. In 2022, the EU-funded TRISTAN project started, addressing the growing need for open-source solutions that can compete with commercial alternatives. It aims to expand, mature, and industrialize the RISC-V ecosystem [9].

RISC-V International maintains the ratified RISC-V ISA specifications in two volumes [10]. The first volume covers unprivileged instructions [4], and the second volume describes the privileged architecture, which covers privileged instructions and additional functionality required for operating systems and external devices [11]. RISC-V does not predetermine a particular microarchitecture or implementation technology, thus supporting both simple systems and highly parallel multicore implementations [4]. Various cores and SoCs already implement RISC-V ISAs [12]. Proprietary suppliers include but are not limited to Andes Technology and Codasip. Open-source examples are the CORE-V Family cores from the OpenHW Group or designs from the PULP Platform.

2.1 Instruction Set Architecture

RISC-V exists in both 32-bit and 64-bit address space variants[1]. The ISA is modular and consists of a small base integer ISA and optional ISEs. These extensions enable designers to customize their systems and optimize their design metrics. The specification defines and maintains standard extensions for additional functionality, such as integer multiplication, floating point, or cryptography. In addition, the RISC-V encoding space allows for custom extensions[2] to tailor highly specialized applications [4]. Table 2.1 gives an overview of selected ISAs and ISEs.

The RISC-V specification defines six instruction formats (R, I, S, B, U, J) in its base RV32I ISA, illustrated in Figure 2.1. R-*type* instructions have two source registers and one destination register and are used for arithmetic or logic instructions such as *add* or *sub*. I-*type* instructions include one source and destination register and a 12-bit immediate field. They are also used for computational instructions like *addi* ("add immediate") and for *load* instructions. S-*type* instructions encode *store* instructions. U-*type* instructions consist of an immediate of 20 bits. B-*type* encoding is a variation of *S-type* but for encoding conditional *branch* instructions, J-*type* instructions are a form of U-type encoding but for *jump* instructions[3].

Table 2.1: Selection of RISC-V base ISA subsets and standard ISEs [4], [13], [14].

Subset Type	Abbreviation	Description
Base Instruction Set	RV32I	32-bit Integer Set
	RV32E	Reduced Embedded 32-bit Integer Set
	RV64I	64-bit Integer Set
	RV128I	128-bit Integer Set
Standard Extensions	M	Integer Multiplication and Division
	F	Single-Precision Floating-Point
	D	Double-Precision Floating-Point
	C	Compressed Instructions
	Zba, Zbb, Zbc, Zbs	Bit-Manipulation
	Zkne, Zknd	AES Encryption and Decryption

[1]The specification also includes a 128-bit variant to be prepared for potentially required larger address spaces in future systems.
[2]The RISC-V specification distinguishes between standard and custom instruction set extensions. However, the term *Custom Instruction (CI)* used in this thesis means instructions added to the RISC-V core's offered base instruction set. A *CI* in this case includes instructions from both *standard* and *custom* extensions.
[3]The intention of this paragraph is to give an intuition regarding RISC-V's individual instruction formats. Details may deviate from this intuition, e.g., the jump instruction *jalr* using the I- instead of the J-type format.

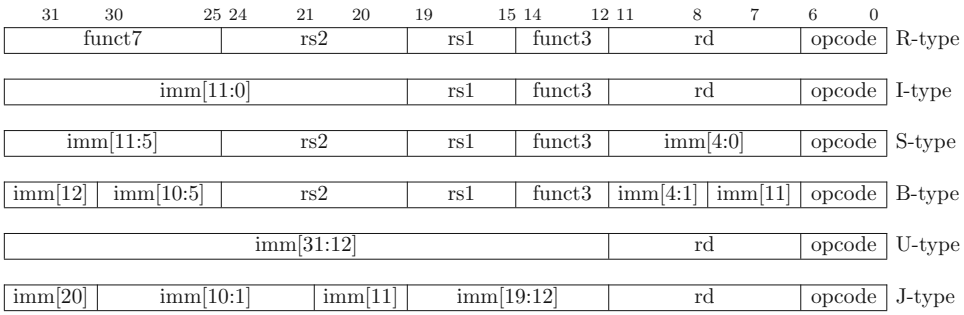


Figure 2.1: RISC-V 32-bit base instruction formats, adapted from Waterman et al. [4]. In general, R-type describe register-type instructions, I- and U-type define immediate and upper immediate instructions, S-type is used for store instructions, and B- and J-type portray branches and jumps. These descriptions can be seen as general guidelines, selected instructions may be realized differently.

2.2 CORE-V Family

A set of RISC-V cores is contributed by the not-for-profit organization and RISC-V International member OpenHW Group. These cores represent the CORE-V Family. The core IP is open source and comes with associated tools and software for design and verification [15]. The family also includes complementary IPs such as the XIF specification.

The CORE-V Family is divided into an embedded (CVE) and application (CVA) class. The embedded class consists of a low-power 2-stage core (CVE2) and a group of 4-stage cores aimed at efficiency (CVE4). The application class consists of 5-stage and 6-stage cores that offer additional features and increased computational power. A summary of the cores and their features can be found in Table 2.2.

The cores of the application class are distinguished by their stage count (CVA5, CVA6). The cores of the embedded class are named CV32E, which represents a 32-bit ISA, followed by their stage count and an additional version number. The 4-stage cores are further divided by a letter representing their different characteristics and application fields [16].

Table 2.2: Summary of CORE-V cores and their features [16].

Class	Name	Aim	Features
CVA	CVA6	Linux-like OS support	privilege levels, TLBs[+], hardware PTW[+], branch-prediction
	CVA5	FPGA implementations	extensible parallel and variable-latency execution units
CVE4	CV32E40P	small size, efficiency	optional DSP[+] extensions
	CV32E40X	**compute intensive applications**	**eXtension Interface (XIF) for external CIs**
	CV32E40S	security applications	Machine and User mode, enhanced PMP[+], anti-tampering
	CV32E41P	Zfinx and Zce ISE support	CV32E40P-fork with Zfinx and Zce extensions
CVE2	CV32E20	low-cost, low-power	high-energy efficiency, computationally limited

[+] TLB: Translation Lookaside Buffer, PTW: Page Table Walker, DSP: Digital signal processing, PMP: Physical Memory Protection

The CVE4 family started with the CV32E40P, former RI5CY from the PULP Platform. It is a 32-bit, 4-stage, in-order core that implements the RV32IM[F]C[Xpulp] ISA. It achieved Functional RTL Freeze in 2021 [17]. The other CVE4 cores are all based on the CV32E40P but are extended and altered with specific goals and applications in mind. The CV32E40S, for example, focuses on security applications and has been extended with the respective features described in Table 2.2.

2.3 CV32E40X

CV32E40X, the X-version of the CVE4 family and main core used within FRANCIS-V, is an in-order RISC-V core that targets compute-intensive applications. Its ISA comprises one of two possible base Integer Instruction Sets and several extensions, some of which are optional. Table 2.3 describes the individual extensions. The core has several features, such as external debug support, hardware performance counters, and a Physical Memory Attribution (PMA) unit. The CV32E40X is customizable, and many stated features can be optionally enabled or excluded. Its IP is fully synthesizable for both FPGA and ASIC implementations. A separate verification project provides testbenches, test cases, and a UVM environment for the CV32E40X and the other CVE4 cores [18].

CV32E40X's characteristic feature is the CORE-V eXtension Interface (XIF). This interface allows designers to extend the core IP with Standard and Custom Instructions without changing the internal structure. The resulting extension is tightly integrated with the core and can be used to speed up critical sections of its applications [19].

Table 2.3: A full list of the CV32E40X ISA, consisting of a configurable set of mandatory and optional base integer sets as well as standard and custom extensions [4], [19].

Group	Abbreviation	Description
Base Integer Instruction Sets	RV32I	Base Integer Instruction Set
	RV32E[+]	Reduced Integer Set for Embedded Systems
Standard Extensions	M, Zmmul[+]	Integer Multiplication and Division
	A[+]	Atomic Instructions
	Zca, Zcb, Zcmb, Zcmp, Zcmt	Code-size reduction
	Zba, Zbb, Zbc, Zbs[+]	Bit Manipulation
	Zicntr	Base Counters and Timers
	Zihpm	Hardware Performance Counters
	Zicsr	Control and Status Register Instructions
	Zifencei	Instruction-Fetch Fence
	Zbkc[+]	Constant time Carry-Less Multiply
Custom Extensions	Xif[+]	eXtension Interface

[+] optionally enabled

Figure 2.2 illustrates the interfaces and the pipeline of the core. CV32E40X has a 4-stage pipeline consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Writeback (WB):

- **IF** fetches instructions from the word-aligned 32-bit prefetch buffer. The buffer has three entries, thus allowing one instruction per cycle to be fetched if the instruction memory allows it. Compressed instructions are also pre-decoded at this stage.
- **ID** decodes the fetched instructions and reads potential register entries. Jumps are also performed at this stage.
- **EX** executes the decoded instructions. It contains an ALU and a unit for multiplication and division. The EX stage also executes branches and generates the Load/Store Unit (LSU) addresses.
- **WB** loads the results from the EX stage back to the register file.

The physical instruction and data memory attributes can be mapped at compile time due to the PMA unit. The instruction and data memory interfaces comply with the Open Bus Interface specification (OBI). Figure 2.3 shows the basic idea for the data memory interface. At the start of a transaction, the core provides the address, control signals, and optional data to be written. It subsequently indicates that the transaction starts with the request signal. When the memory is ready, it responds with the grant signal. If the transaction is a read, the read data is given along with the appropriate valid signal in the following cycle. The instruction interface works on the same principle but without write capability and with an additional error signal to indicate a bus error [19], [20].

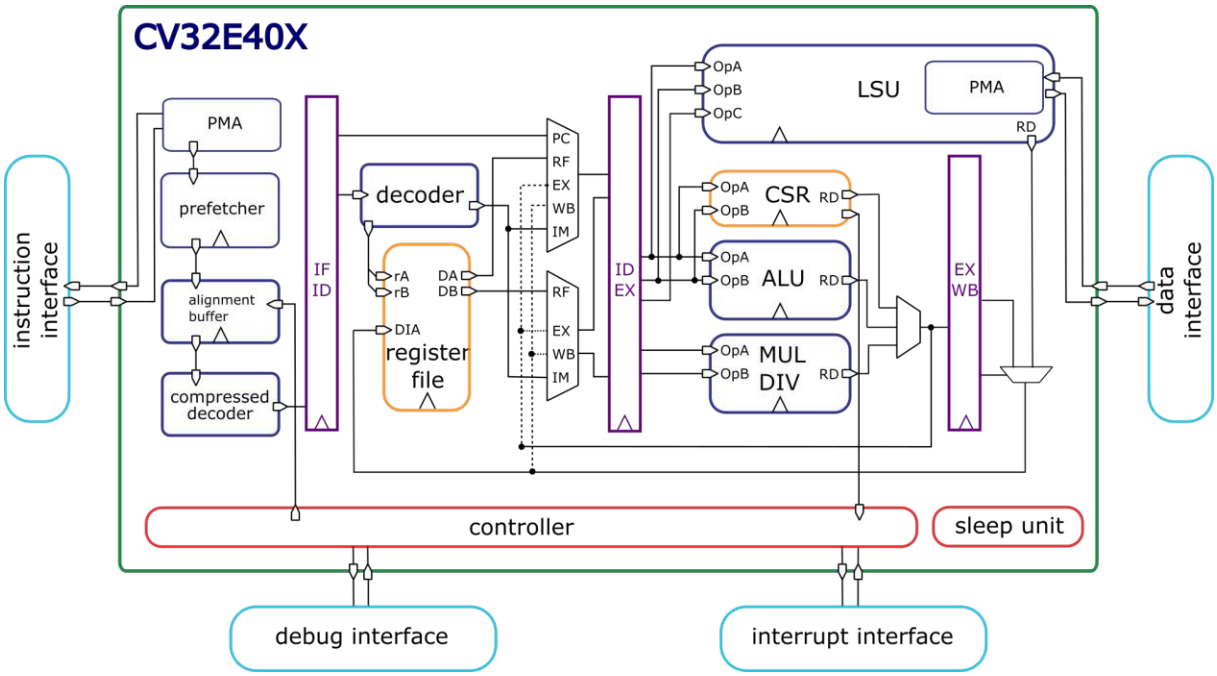


Figure 2.2: Block diagram of the CV32E40X, adapted from its user manual [19]. The pipeline consists of four stages, namely Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Writeback (WB).

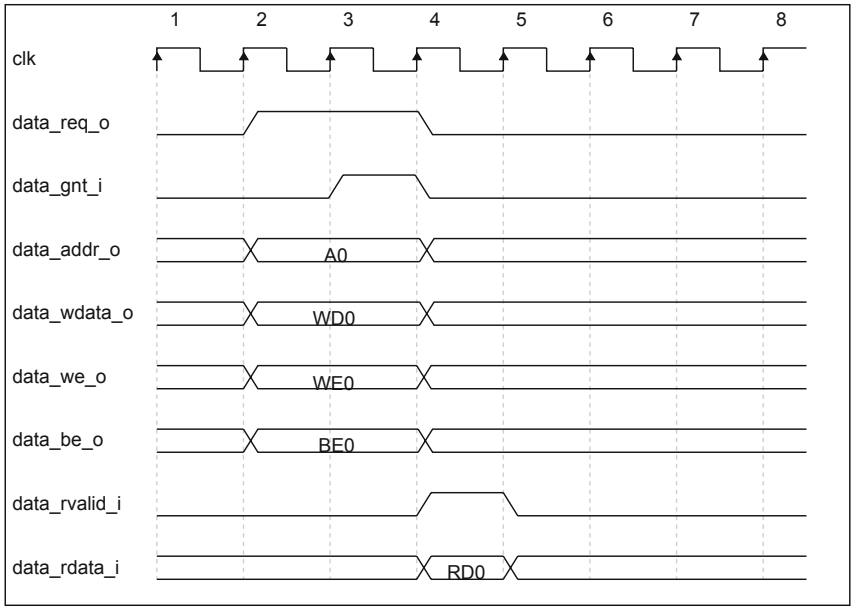


Figure 2.3: A basic transaction of the CV32E40X data memory interface, adapted from the CV32E40X user manual [19]. The core provides the memory address, control signals and optional data, and request signal. The memory responds with the grant signal. It also provides the valid signal and optional read data in the next cycle.

2.4 CORE-V eXtension Interface

The CORE-V eXtension Interface (XIF) is a CI interface for ISEs and coprocessors based on the RISC-V ISA and the chosen CI interface for FRANCIS-V. The goal of XIF is to provide a standardized design and verification methodology for CIs without the need to change the internals of the core. The CI functionality is implemented in external modules or coprocessors and is integrated with the cores at the system level. The interface is relatively new and still in active development, with the first pre-release specification published in August 2021. The current pre-release specification 0.2.0 from April 2022, when writing this thesis, is available on GitHub [21] and docs.openhwgroup.org [7]. Version 0.2.0 features a thorough revision of the overall interface structure and the introduction of its six subinterfaces.

XIF has independent channels for instruction offloading and result writeback and is tightly integrated with the core's register files. The following instruction types are supported:

- *ALU* instructions
- *Load/store* instructions
- *Control Status Register* and related instructions

Control-transfer instructions, such as jumps or conditional branches, are not supported.

XIF supports the following features:

- **Simple instruction encoding**: XIF CIs use standard RISC-V bitfields for the instruction operands and results. Unused bitfields can be repurposed.
- **Compressed instructions**: XIF supports 16-bit compressed instruction encoding to reduce code size[4].
- **Dual read and writeback instructions**: For 32-bit register width, XIF optionally enables instructions to perform dual reads or writebacks on even-odd register pairs.
- **Ternary operands**: Instructions with three input operands are supported. Thus, in combination with dual read, six 32-bit operands per instruction can be realized.

Structure

XIF consists of six subinterfaces named *Compressed*, *Issue*, *Commit*, *Memory*, *Memory Result*, and *Result*, summarized in Table 2.4 and illustrated in Figure 2.4. The interfaces are used in this respective order for every instruction. Each interface is responsible for a different phase in the overall flow of instructions, but not all interfaces are mandatory for all instructions. Issue, Commit, and Result are mandatory, while the Memory and Memory Result interfaces are optional. The Compressed interface per standard is mandatory but can be reduced to a trivial implementation[5] if no compressed CIs are used in the particular application.

Table 2.4: Summary of the CORE-V-XIF subinterfaces [7]. The direction field indicates if the data flow is in one or both directions. The initiator field states the component that starts the transaction. The Ready field indicates if the ready signal is explicitly implemented or not.

Name	Direction	Initiator	Ready	Functionality
Compressed^{+}	Bidirectional	Core	Explicit	Decompress compressed instructions
Issue	Bidirectional	Core	Explicit	Issue instructions, provide operands
Commit	*Unidirectional*	Core	*Implicit*	Commit or kill speculative instructions
Memory^{x}	Bidirectional	*Coprocessor*	Explicit	Initiate memory reads and writes
Memory Result^{x}	*Unidirectional*	Core	*Implicit*	Execute memory reads and writes
Result	*Unidirectional*	*Coprocessor*	Explicit	Signal instruction completion, provide results

$^{+}$ mandatory, but trivial implementation possible
^{x} optional

[4] According to Waterman et al., if compressed instructions are used for all possible instructions in a typical program, the resulting code size is reduced by around 20-30% [4].
[5] The core attempts to offload every compressed instruction that it does not recognize as legal. To prevent the core from freezing, hence, the coprocessor has to respond to every compressed request. However, the trivial method for doing so is holding a particular interface signal to zero to effectively reject every incoming request.

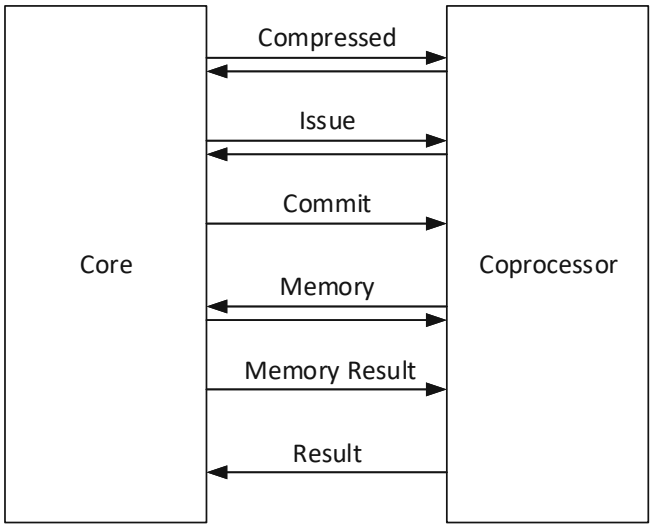


Figure 2.4: Diagram of a XIF-connection showing the six subinterfaces. The interfaces are ordered by their logical flow during execution. The arrows indicate the direction of the data flow. The initiator is the component from which the upper arrow originates. The instructions are issued with the Compressed and Issue interface. They are committed or killed by the Commit interface. The Memory and Memory Result interfaces are used by load/store instructions. The Result interface delivers the instruction result.

All interfaces perform a *valid-ready* handshake between the core and the coprocessor, with one of the two components being the initiator of the transaction. To start the transaction, the initiator requests a transaction with *valid*, and the responding component accepts the transaction with *ready*. In other words, the transmission is valid if both ready and valid are high in the respective cycle. The transferred data are grouped into the categories of request and response, with request data coming from the initiator and responder, respectively. Handshake, request data, and response data are all transferred in the same cycle, ensuring tight integration of the core and the external instruction unit.

Each interface is unidirectional or bidirectional and has an explicit or implicit ready signal. The bidirectional interfaces with an explicit ready signal (Compressed, Issue, Memory) behave as described above. The Commit and Memory Result interfaces are unidirectional with an implicit ready signal. Unidirectional in this context means that only the initiator can transmit data, and the responder is solely a receiver. Furthermore, there is no dedicated ready signal, but the receiving component is assumed to react to the incoming valid signal at any time and process the provided information. The Result interface is also unidirectional but has a dedicated ready signal, allowing the core to control when it processes the incoming results.

Operating Principle

The core attempts to offload every instruction it does not recognize as legal over either the Compressed or Issue interface. The coprocessor can accept implemented or reject unknown instructions. Compressed instructions are handled via the Compressed interface in the first step. The coprocessor delivers the non-compressed instruction.

All instructions[6] are offloaded via the Issue interface. The request includes register file operands needed for the instruction execution. The coprocessor responds to the issued instruction with acceptance or rejection and additional information about the instruction, e.g., if the instruction performs a writeback to the register file or if it is a load/store instruction. After this issuing, the coprocessor handles the instruction. The instruction is speculative until the core commits or kills it via the Commit interface.

In case of a load/store instruction, the coprocessor uses the Memory and Memory Result interfaces to access the core's load/store unit. It requests a memory transaction over the Memory interface and passes information needed for the transaction, like memory address, write enable, or write data. The core accepts the transaction via the Memory interface and delivers the resulting memory data over the separate Memory Result interface.

The coprocessor signals that it has completed the instruction through the Result interface. During this transaction, it also performs the optional result writebacks. The coprocessor must induce exactly one result transaction for each issued and committed instruction to signal the instruction completion to the core, even if no writeback is performed.

[6]Both originally non-compressed instructions and also compressed instructions previously converted via the Compressed interface are offloaded via the Issue interface.

2.5 RISC-V AES Extension

The RISC-V scalar cryptography extension is a standard extension for cryptography ISEs within the RISC-V specification, among others, including CIs for Advanced Encryption Standard (AES) encryption and decryption [10], [14]. AES is one of the two selected use cases within this thesis to propose and evaluate FRANCIS-V. In the specification, the AES instructions are grouped within the *Zkne* and *Zknd* extensions[7] for AES encryption and decryption, respectively. The two extensions feature the four different 32-bit[8] instructions, namely, *aes32esmi*, *aes32esi*, *aes32dsmi*, and *aes32dsi*, for middle and final round encryption and decryption, respectively. They are described in Table 2.5.

AES is a round-based algorithm. After an initial key expansion or *key scheduling*, the plaintext on the input is encrypted in iterative rounds [22]. The instructions of the crypto standard perform parts of these rounds. They are also utilized during the key scheduling routine prior to the encryption rounds [23].

Accompanying the specification, the developers of the standardization work offer hardware implementations for their proposed instructions. The instructions are implemented as combinational modules in Verilog. The standardization work further offers the source code of two AES software algorithms for comparison with the hardware implementation, namely, a *Reference* implementation and a more optimized *ttable* implementation.

Table 2.5: Overview of the 32-bit instructions of the RISC-V Scalar Cryptography Extensions *Zkne* and *Zknd* for AES encryption and decryption [14].

ISE	Mnemonic	Instruction	Description
Zkne	aes32esmi	Middle round encrypt	Forward SBox, partial forward MixColumn, XOR
	aes32esi	Final round encrypt	Forward SBox, XOR
Zknd	aes32dsmi	Middle round decrypt	Inverse SBox, partial inverse MixColumn, XOR
	aes32dsi	Final round decrypt	Inverse SBox, XOR

[7] Zkne and Zknd stand for **K**rypto **N**IST Suite AES **E**ncryption and **D**ecryption.
[8] The 64-bit instructions of the specification are not covered in this thesis.

Chapter 3

Related work

ASIPs, ISEs, and CIs certainly are not new areas of research. There is a significant body of literature from the past few decades on the development of ASIP-based systems and on the automation of its process steps [3]. However, the emergence of RISC-V and the growing interest in open-source solutions sparked new interest in the topic.

In this thesis, the related state of the art comprises existing RISC-V methodologies for CI development, particularly for CI integration. We differentiate between three relevant topics to position our thesis within this existing work: *RISC-V design methodologies*, *RISC-V CI interfaces*, and *XIF implementations*. Table 3.1 gives an overview of the relevant work on these topics.

The first topic consists of tools and methodologies for RISC-V CI design and integration compared to the proposed framework FRANCIS-V. In the respective section (Section 3.1), we group the discussed work on their integration technique, denoting them as either in-pipeline integration or interface-based integration approaches.

The latter, interface-based tools utilize CI interfaces as their integration approach. Those CI interfaces comprise the second research topic. Prominent CI interfaces are presented in Section 3.2, therefore, after addressing the tools that utilize them in Section 3.1. Section 3.2 also compares the interfaces to FRANCIS-V's utilized interface XIF.

The third topic (Section 3.3) is existing XIF-based coprocessors and accelerators compared to the resulting coprocessor or *wrapper* designs generated by FRANCIS-V.

Table 3.1: Overview of the related work covered in this thesis, ordered by their research topics. Work such as SCAIE-V or the thesis from Waage et al. covers aspects of multiple topics, thus is listed separately in each group. FRANCIS-V, its utilized CI interface XIF, and FRANCIS-V's generated wrapper solution are highlighted in bold.

Topic	Abbreviation	Authors	Description	Ref.
RISC-V CI design methodologies	Codasip Studio	Codasip	Commercial CI design tool for Codasip cores	[5]
	ACE	Andes	Commercial CI design tool for AndesCore™ cores	[24]
	OpenASIP	Hepola et al.	ASIP design toolset for a customizable core	[6]
	–	Hu et al.	ISE framework for the riscv-mini	[25]
	–	Waage et al.	AES XIF wrapper	[26]
	–	Poncino and Pala	Coprocessor framework for the ROCKET core	[27]
	SCAIE-V	Damian et al.	Interface-based CI integration tool	[28]
	FRANCIS-V	**This work**	**Interface-based CI integration framework**	–
RISC-V CI interfaces	PCPI	YosysHQ	CI interface for the PicoRV32	[29]
	RoCC	Asanović et al.	CI interface for the ROCKET core	[30]
	CCOPI	Nazarenus and Swierzy	CI interface for the VexRiscv	[31]
	NICE	Nuclei	CI interface for the Hummingbirdv2 E203	[32]
	TIGRA	Green et al.	Simplified and tightly integrated CI interface	[33]
	SCAIE-V	Damian et al.	Scalable and portable CI interface	[28]
	XIF	**OpenHW Group**	**Generalized CI interface**	**[21]**
XIF implementations	FPU_SS	Imfeld	Floating-point coprocessor	[34]
	NFC accelerator	Babbaro et al.	NFC accelerator	[35]
	Vicuna	Platzer and Puschner	Timing-predictable vector coprocessor	[36]
	Spatz	Cavalcante et al.	Compact vector coprocessor	[37]
	PQC coprocessor	Lee et al.	PQC coprocessor	[38]
	AES accelerator	Waage et al.	AES accelerator and basic coprocessor template	[26]
	XIF Wrapper	**This work**	**Generic coprocessor template**	–

3.1 RISC-V CI design methodologies

The survey by Jain et al. [3] from 2001 gives an extensive overview of work on ASIPs from the last decades. Albeit missing recent advances regarding RISC-V relevant to this thesis, the author's proposed ASIP development process structure still coarsely aligns with the structure of recent tools and methodologies presented subsequently in this section. The authors define and describe five main process steps of ASIP synthesis commonly realized by relevant work. They are illustrated in Figure 3.1. According to their definition, the synthesis starts with analyzing the high-level application code, architecture selection, and instruction set definition. The generation of object code and hardware generation follows these steps. The authors emphasize that not every design methodology implements all these steps in this explicit form.

In contrast to the extensive body of literature on general ASIP synthesis, less work on RISC-V-based CI design solutions, the first of the three mentioned research topics, is available. In this thesis, these design approaches are categorized into two groups of integration types, namely, *in-pipeline integration* and *interface-based integration*, as illustrated in Figure 3.2. These groups are adapted by the definition of

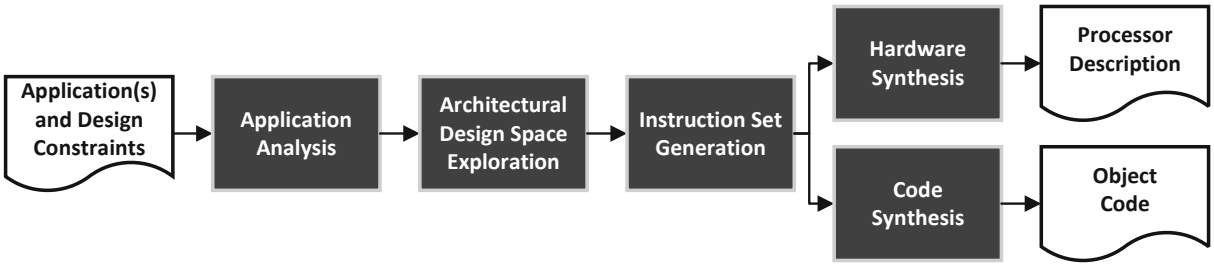
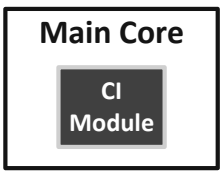


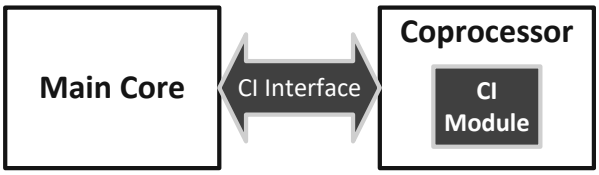
Figure 3.1: Flow diagram of the five main steps used in ASIP synthesis, adapted from Jain et al. [3, Figure 1]. The synthesis process starts with the application, analyzes it, explores architecture candidates, generates the CI Set, and produces the software binary and hardware for the system.

Damian et al.[1] [28]. In-pipeline integration approaches (Figure 3.2a) integrate the CI hardware directly into the processor's pipeline in the form of special functional units, as formulated by Galuzzi et al. [39]. Damian et al. state the advantage of this integration type over interface-based integration to be less latency overhead and more throughput potential as well as reduced area and resource usage. By contrast, the interface-based approach (Figure 3.2b) utilizes CI interfaces to connect the RISC-V core with an external coprocessor that, in turn, contains the CI logic. The advantage of this approach over in-pipeline integration is the compatibility with a broader range of processors, provided that they support the interface, and the flexible use and reuse of existing CI designs. Furthermore, the processor's internals do not have to be adapted, eliminating possible unintended behavior or design faults.

Table 3.2 outlines important features of the tools that this section discusses. The general features listed are the availability of a license and if the authors provide metrics for comparing their generated designs, such as runtime improvement or resource overhead. Implementation features include the integration type, which cores the tool supports, and whether the methodology is further applicable for various additional RISC-V cores. Design process features describe which parts of the CI development process (detailed in Section 4.1) are covered by each tool. Finally, types of supported instructions are listed.



(a) **In-pipeline integration** [28]: CI hardware added inside the main core as a separate execution unit.



(b) **Interface-based integration**: CI hardware integrated with a CI interface and an external coprocessor.

Figure 3.2: Processor systems with integrated CI hardware using in-pipeline integration (a) or a CI interface and an external coprocessor (b). The figure is adapted from Damian et al. [28, Figure 2].

[1]Damian et al. actually define three distinct approaches, further categorizing the *interface-based* approach by splitting it into an on-chip *coprocessor* and off-chip *accelerator* group.

Table 3.2: Features overview of the most promising RISC-V CI design tools discussed in this section. Codasip Studio and OpenASIP exhibit the most features compared to the other commercial and open-source solutions, respectively, but only support a specific set of RISC-V cores. SCAIE-V and FRANCIS-V support generic RISC-V cores. To support new cores, they have to be added to the tool SCAIE-V, while FRANCIS-V relies on developers adding XIF to their cores.

	Feature	Codasip Studio [5]	ACE Framework [24]	OpenASIP [6]	Hu et al. [25]	SCAIE-V [28]	FRANCIS-V (This work)
Gen.	**License**	commercial	commercial	open source	open source	open source	open source[+]
	Metrics provided	×	×	✓	✓	✓	✓
Implementation	**Integration type**	in-pipeline	in-pipeline	in-pipeline	in-pipeline	interface-based	interface-based
	Supported cores	Codasip cores	AndesCore™	OpenASIP-gen.	riscv-mini	VexRiscv Piccolo Orca PicoRV32	CV32E40X
	Generic core supp.	×	×	×	×	✓	✓
Design process	**CI Identification**	✓	✓	×	×	×	×
	CI Hardware gen.	✓	✓	✓	✓	×	×
	SW Compilation	✓	✓	✓	×	×	✓
	HDL Simulation	✓	✓	✓	×	×	✓
	Verification[x] features	Test case gen.	Test case gen. Reference comp.	NA	NA	NA	Testbench SW debug
Instr. type	**Multi-cycle**	✓	✓	NA	NA	✓	×
	R-type	✓	✓	✓	✓	✓	✓
	Immediate **Store** **Jumps, Branches**	✓	NA	NA	×	✓	×

[+] FRANCIS-V is open source except for QuestaSim. More details are presented in Section 5.3.
[x] The listed verification features are support for automated *test case generation*, *comparison* to a ISS *reference* simulation, automated *testbench* generation, or *debug* support for the application's *software* code.

3.1.1 In-pipeline integration

The companies *Codasip* and *Andes Technology* offer commercial tools and design flows for automated CI development. Codasip Studio [5] and the Andes Custom Extension™ (ACE) framework [24] support CI identification by providing code profiling methods to identify often-used code segments. After that, the designer-defined instructions can be simulated, verified, and implemented in RTL. The frameworks also extend the required tools and compilers with the CIs. Andes further states the support of scalar single- and multi-cycle instructions as well as vector instructions using loops. Moreover, immediate, register, or memory operands are supported. Unlike most similar toolsets, Codasip additionally supports control-flow type instructions like jumps or conditional branches, as also identified by Damian et al. [28].

Furthermore, neither Andes nor Codasip seem to provide any benchmarks of the resulting designs like runtime reduction or resource overhead [28]. In addition, the tools only support Codasip and Andes processors, respectively, and are therefore not suited for use with additional open-source RISC-V cores. Additionally, unlike the RISC-V ISA itself being open source, both proposed tools are closed source, denying the growing RISC-V open-source community opportunities to further extend, adapt, or port these tool and their generated designs.

An open-source alternative to Codasip Studio and the ACE framework is *OpenASIP 2.0* by Hepola et al. [6], a co-design toolset for ASIP synthesis. The designer can describe the semantics of the CIs either with HDL snippets or with a directed acyclic graph and invoke them in the code with generated C intrinsics. The tool then adapts its compiler and performs the code compilation. Furthermore, it generates the previously customized processor and hardware for the defined CIs. It also supports the simulation of the generated RTL system.

OpenASIP provides a customizable processor with a single-issue in-order pipeline with 3 to 4 stages. As of writing this thesis, the tool does not support code profiling, automated CI identification, or automatic utilization of CIs by the compiler. The designer has to manually select suitable CIs and insert the generated intrinsics in the C code. Aside from CI selection, OpenASIP's functionality compares to the previously stated commercial tools. In addition, it provides metrics regarding runtime reduction and post-synthesis properties, such as resource usage and maximum frequency, for the use cases AES, CRC, and SHA. Similarly to OpenASIP, the framework from Hu et al. [25] offers automated ISE generation out of graph-based descriptions of CIs. However, it does not feature compilation or simulation capabilities. The generated CI RTL logic is directly integrated into the simple 3-stage pipeline of the *riscv-mini* core.

Both open-source solutions from Hepola et al. and Hu et al. address similar problems as FRANCIS-V. Our framework as well features code compilation, CI integration, and simulation of the resulting RTL. While FRANCIS-V does not support CI hardware generation in contrast to these tools, it is also developed with HLS-based hardware generation in mind. However, the integration approach of the commercial and open-source tools discussed above differs from the one in this thesis. According to the definitions from the beginning of this chapter, Codasip Studio, OpenASIP, and the other proposed tools utilize in-pipeline integration. FRANCIS-V, by contrast, realizes integration by using a CI interface, therefore featuring the discussed advantages of supporting a broader range of cores without the need to adapt them to the integrated CIs.

3.1.2 Interface-based integration

The interface-based approach, as opposed to in-pipeline integration, is less common in literature. Like the in-pipeline-based tools presented above, existing interface-based approaches are mainly designed for one specific RISC-V core or a small set of cores. Thus, these approaches are not suited for use with a broad range of RISC-V cores. Work on these approaches mainly contains CI interfaces themselves[2] but does not include automation methodologies or tools for supporting the CI design process as the above-presented solutions, e.g., Codasip or OpenASIP, do. However, we identify two master theses and one tool that address aspects of this process automation by utilizing coprocessor interfaces.

The AES accelerator from Waage et al. [26] was published while this work was still in development. It has several parallels to this thesis. The authors utilize the same RISC-V core and CI interface for the implementation as FRANCIS-V, namely, the CV32E40X and XIF. Choosing CV32E40X is apparent, given that it is the only openly available core with XIF support at at the time of writing this thesis. Furthermore, their application field, AES cryptography, is also a use case chosen for this thesis, and the same standardization work is used for its hardware implementation. As proposed in Section 5.4.2, this already available standardization work is one major reason for choosing AES as a test case for FRANCIS-V. Waage et al. also present a basic wrapper for XIF, a similar approach to the XIF wrapper proposed in this thesis.

However, the work on the AES accelerator serves a different purpose. Waage et al. focus on implementing an accelerator with first-order side-channel attack security. By contrast, this thesis focuses on developing an interface-based methodology and only utilizes the AES use case to demonstrate and evaluate the developed framework. Waage et al., while proposing a basic wrapper and stating that it can be adapted for different purposes, do not provide further use cases or an underlying methodology to apply the wrapper for other CIs in an automated way as FRANCIS-V does[3].

The master thesis by Poncino et al. [27] covers RISC-V coprocessor development and gives guidelines for future coprocessor designs. Like the work from Waage et al., the thesis exhibits some similarities to our work upon initial examination. Poncino et al. conducted a case study where they developed a RISC-V coprocessor for cryptography that, among others, can also perform AES encryption. Furthermore, they outline design considerations for a generic coprocessor framework. This generic approach is similar to

[2] Note that CI interfaces are presented in the subsequent section, Section 3.2. This section only addresses design automation work that utilizes these interfaces.

[3] Since Waage's master thesis and the following thesis by Poncino, albeit discussing generic wrappers, do not focus on such approaches nor propose concrete automation tools for their generation, they are hardly comparable to the other proposed CI design tools and therefore not included in Table 3.2.

the idea behind FRANCIS-V's wrapper methodology. In addition, Poncino et al. also utilize a CI interface for their CI integration.

However, they do not utilize XIF but the RoCC interface of the Rocket Chip Generator. This interface is designed for the generated cores *ROCKET* and *BOOM* [30], while XIF is designed to support several RISC-V processors. Furthermore, the authors assess a particular latency when utilizing the RoCC interface. Therefore, they advise not to use the interface for tightly coupled coprocessors with frequent data exchange. By contrast, XIF appears suitable for this coprocessor type in our experiments. In addition, Poncino et al. discuss generic framework approaches but do not implement a generic coprocessor solution. Similarly to Waage et al. and in contrast to FRANCIS-V, they also do not offer a framework for automated generation of a CI-based system based on their proposed coprocessor design.

SCAIE-V by Damian et al. [28] describes both a CI interface for RISC-V processors and an associated open-source hardware generation tool. Like FRANCIS-V, Damian et al. aim to reduce the high engineering effort of CI-based systems, especially when updating systems or reusing components for successor systems. They state *scalability*, *portability*, and *flexibility* as benefits of SCAIE-V. Scalability means that the interface scales with the complexity of the CIs. Portability describes it as being portable with multiple RISC-V cores[4]. With flexibility, the authors mean to emphasize the interface's considerable amount of features. SCAIE-V supports register and memory instructions, I/O transfer instructions, and control flow instructions. It further supports multi-cycle and decoupled instructions that operate parallel to the main core's pipeline. It can reuse the pipeline of the main core and features hazard handling and pipeline stalling capabilities. The interface is realized with a set of mandatory and optional signals such as control, data, or address signals. SCAIE-V features tools for hardware generation and system integration. The designer has to configure SCAIE-V and provide the CI hardware implementation. Based on these inputs, its automatic hardware generator updates the main core and generates the interface to connect the CI hardware logic with the core.

SCAIE-V has several parallels to the work in this thesis. Both SCAIE-V and FRANCIS-V implement interface-based methodologies and tools for integrating externally provided CI hardware modules with RISC-V cores. SCAIE-V, among others, also uses an SBOX instruction (a part of the popular use case AES) to demonstrate its performance. In addition, HLS support for generating the CI hardware modules themselves is scheduled as a prospective feature of the tool [28, Conference presentation by Andreas Koch, DAC 2022]. Similar considerations are also incorporated into the design of FRANCIS-V.

[4]At the time of writing this thesis, SCAIE-V supports the four cores VexRiscv, Piccolo, Orca, and PicoRV32.

A major difference between SCAIE-V and FRANCIS-V is how the tools approach the support of multiple RISC-V cores. In order to use a particular core with SCAIE-V, it has to be adapted and added to the tool [40]. FRANCIS-V, by contrast, supports cores compatible with XIF. Therefore, a core without XIF support has to be extended with this interface functionality. At the time of writing this thesis, SCAIE-V already features four different cores, while FRANCIS-V only supports the CV32E40X. The designers of SCAIE-V can add additional cores by themselves, while FRANCIS-V relies on the developers of RISC-V cores to incorporate XIF. However, with the maturing and establishing of the interface, we expect that additional RISC-V cores will support XIF as time passes. Therefore, the resulting designs of FRANCIS-V may be inherently compatible with a larger set of cores in the long run since, by contrast to SCAIE-V, the tool itself does not have to be adapted to new cores. A further advantage of the approach in this thesis is that already verified XIF-based RISC-V cores are not internally altered by FRANCIS-V since the CI extension is realized solely over the already implemented and verified interface.

Regarding features, SCAIE-V supports elaborate instruction types, including multi-cycle and control transfer instructions. FRANCIS-V has more limited instruction capabilities. However, unlike FRANCIS-V, SCAIE-V does not feature software compilation, HDL simulation, and verification tools within its framework. Comparisons on quantitative metrics like utilization overhead are presented in Section 7.4.

3.2 RISC-V CI interfaces

After discussing the first research topic, CI design tools and methodologies, the second topic focuses on the particular CI interfaces used by such tools. In their paper about SCAIE-V, Damian et al. also give an overview of such potential CI interfaces. The main options listed by the authors can be grouped into generic CI interfaces, CI interfaces for specific domains, and existing interfaces originally developed for other purposes. We focus on the first group, since interfaces in the latter two categories are either too specific or exhibit a too high performance overhead to be suited for a flexible but efficient integration approach as desired in this thesis [28].

There are a number of solutions for generic RISC-V CI interfaces. Table 3.3 presents the most prominent candidates. Among these options, XIF is selected for FRANCIS-V for several reasons. First, when starting this work, it newly emerged and is still in active development at the time of writing this thesis. This recency designates XIF as the most interesting and prominent candidate for a novel integration framework compared to the preceding candidates, PCPI [29], RoCC [30], CCOPI [31], and NICE [32]. These four solutions, in addition, are developed with a specific core in mind, while XIF offers a more

generalized structure. It currently supports the CORE-V CV32E40X, and we envision an increased XIF utilization in future RISC-V processor designs.

The developers of the interface *TIGRA* [33] follow a different strategy than the other mentioned approaches. TIGRA is described as a zero latency interface aimed at achieving the advantages of in-pipeline integration, e.g., smaller overhead, while simultaneously reducing the manual implementation effort. The authors achieve this by defining a simplified interface that only requires internal signals that a typical RISC-V data path already entails, so no internal modifications of the core are needed. The downside of this strategy is the restricted functionality of the interface. In contrast to XIF, TIGRA only supports R-type instructions[5], so direct memory access is not provided. Another difference is that TIGRA does not have dedicated signals for committing or killing speculative instructions.

The tool SCAIE-V is already contrasted to FRANCIS-V in Section 3.1.2, which also covers the differences of the interface SCAIE-V compared to XIF. Aside from the discussed RISC-V core support differences, no major feature differences become apparent when only comparing the interfaces.

3.3 XIF implementations

The third relevant topic for this thesis is existing work on XIF coprocessors and accelerators. As already mentioned, XIF is a rather new interface. As of writing this thesis, it is implemented by the CV32E40X core from the OpenHW Group [19], and we anticipate further core support in the future. However,

Table 3.3: Overview of existing RISC-V CI interfaces. The comparison renders SCAIE-V and XIF as the most promising candidates with extended feature and core support.

Name	Year published	Extended features	Target	Ref.
PCPI	2015	–	PicoRV32	[29]
RoCC	2016	–	Rocket Chip Generator	[30]
CCOPI	2018	–	VexRiscv	[31]
NICE	2020	–	Hummingbirdv2 E203	[32]
TIGRA	2021	×	Generic cores	[33]
SCAIE-V	2022	✓	Generic cores	[28]
CORE-V XIF	2022[+]	✓	Generic cores	[21]

[+] This date corresponds to the 0.2.0 pre-release specification, since the previous version 0.1.0 from 2021 exhibits an entirely different structure and working principle.

[5]FRANCIS-V is currently also limited to R-type instructions, as reasoned in Section 4.2. However, XIF itself supports more elaborate instruction types, and further instruction support by FRANCIS-V is planned in future work (Section 7.6).

despite its recent emergence, some coprocessors and accelerators already utilize XIF as their interface. They are summarized in Table 3.4.

All listed accelerators and coprocessors hold different assumptions than our thesis. The designers focused on developing and optimizing their coprocessor for specific use cases, while we focus on a generalized solution for several applications. The implementations handle XIF transactions either directly inside the responsible components or in dedicated interface modules. Hence, all solutions are entangled with the specific application to some extent. The XIF implementations would have to be manually adapted to update functionality, add new CIs, or reuse the proposed architecture in other application fields. This thesis, by contrast, aims to minimize this manual effort by developing a generalized interfacing framework for a wide range of CIs and application fields. The only exception to these application limitations is the work from Waage et al. which proposes a basic wrapper but does not provide an underlying integration methodology, as already addressed in Section 3.1.2.

Despite the diverging focus of the listed work in contrast to this thesis, it is still of interest to analyze how and to which extent the listed coprocessors utilize XIF. This analysis helps to design and understand our proposed wrapper's structure (presented in Section 5.2.1). From Table 3.4, only FPU_SS [34] implements compressed instructions and thus all six XIF subinterfaces. The Near Field Communication (NFC) accelerator utilizes every interface but the Compressed interface [35]. The Post-Quantum Cryptography (PQC) and AES solutions, besides not using the Compressed interface, additionally do not implement load/store type instructions [26], [38]. Thus, they do not utilize the Memory and Memory Result interfaces responsible for accessing the LSU of the main core. They implement the three required interfaces Issue, Commit, and Result.

Table 3.4: Summary of coprocessors and accelerators utilizing XIF. Among these options, the developed wrapper in this work is the only solution supporting a broad range of CIs and offering related tool support. The "∼" in the AES accelerator column indicates that Waage et al. [26] shortly discuss their proposed basic wrapper but do not go into detail on potential automation approaches.

Name	Generic CI support	Functionality and Features	Ref.
FPU_SS	×	F and Zfinx single-precision floating-point extensions	[34]
NFC accelerator	×	CIs for NFC decoding	[35]
Vicuna	×	Zve32x vector extension, timing-predictable	[36]
Spatz	×	Zve32x vector extension, compact and energy-efficient	[37]
PQC coprocessor	×	CIs for PQC	[38]
AES accelerator	∼	Zkne and Zknd AES extensions, first-order SCA[+] secure	[26]
XIF Wrapper	✓	**Support of a broad range of CIs**	**This work**

[+] SCA: Side-Channel Attack

The accelerators Vicuna and Spatz focus on vector operations and implement vector load/store instructions and could, therefore, utilize the Memory and Memory Result subinterfaces. However, the authors of Spatz stated that at the time of developing their work, the XIF specification did not meet the bandwidth required to transfer vectors over these two interfaces efficiently[6] [37]. Thus, Spatz implements a separate Vector LSU and uses an adapted XIF to consider instructions with a bigger maximum bandwidth. Vicuna follows a similar approach by implementing a vector LSU. Furthermore, Vicuna implements the Memory and Memory Result interfaces. It optionally allows communicating directly with the memory over a custom memory interface instead, as becomes apparent when studying the respective RTL components [41].

Although the implementation details differ for the individual coprocessors, a coarse and abstract overview of the most common architectural components can be given. Most coprocessors and accelerators implement some kind of decoding unit for incoming instructions and FIFOs to help with handling multiple instructions or to allow selective committing or killing of instructions. Furthermore, a functional unit implements the custom instruction logic itself. Most architectures also include a register file for operands, results, and intermediate results in multi-cycle instructions. Finally, control logic or controllers of differing complexity manage the other components of the coprocessor and, among other things, handle interfacing, memory transactions, and commits or perform configurations of the functional units.

[6]The current maximum bandwidth is defined to be 256 bit per cycle. However, in early drafts, the size limit was 32 bit [21], which was likely the case when being assessed by Spatz.

Chapter 4

Methodology

This chapter proposes the methodology to answer our research questions by analyzing the typical CI development process, presenting FRANCIS-V, and discussing the thesis's design time evaluation strategy.

4.1 CI development process analysis

The first research question of identifying the major challenges within the RISC-V CI development process is addressed by analyzing the typical manual process steps a designer has to perform to design, implement, and integrate a CI-based RISC-V system. The analysis is based on the following assumptions to ensure a focus on the essential topics of this thesis:

- The development process and targeted system are based on RISC-V.
- The application is given in C code.
- The designated goal of the process is to realize CIs to enhance the application's performance with a resource overhead comparable to similar approaches.
- XIF is used for integrating the external CI hardware logic.
- The main core can be any core that supports XIF.
- The process does not include synthesizing and implementing the generated system on hardware such as FPGAs or ASICs.

The paper by Jain et al. [3], as presented in Chapter 3, gives a collective overview of common steps realized by CI design tools. Based on their generalized process analysis and the assumptions from above, we define the specific XIF-based CI development process as illustrated in Figure 4.1. The figure proposes

the process's development steps and their inputs and outputs. The steps are defined and detailed in the following paragraphs by dividing them into five groups: *CI identification*, *CI implementation*, *CI integration*, *code synthesis*, and *simulation and verification*.

The steps of the typical development process are conducted by the designer. When utilizing the framework FRANCIS-V, subsequently proposed in Section 4.2, some of these steps (blue in Figure 4.1) are performed by the interface, while others remain to be done manually (gray in the figure).

The process analysis and illustration omit the potential iteration loops during a practical process to enhance their contentual and visual clarity. Such iterations may be necessary when previous steps have to be revisited, e.g., due to erroneous or inadequate behavior of the system or its components. For example, consider a software bug that is identified during the system verification step but originates from an error done during code adaption. In that case, the designer has to go back to the C code adaption step, fix the error, and proceed through the subsequent process steps again.

CI identification

The defined development process starts with the C application. Given this application as input, the following step *CI Identification* comprises identifying suitable CIs [39]. This step can be compared to the application analysis, architectural design space exploration, and instruction set generation steps defined by Jain et al. According to their definition, the C application is profiled to identify often-used code segments, referred to as *hot spots* in literature [42]. With knowledge of these hot spots, the performance of different architecture and CI combinations is estimated to find the most suitable configuration [3].

When applying the assumptions of our defined development process, this problem is reduced to selecting the most suited XIF-supporting main core for the given cost, power, and performance requirements and to finding the optimal set of CIs for the given application. The designer has to conduct this selection based on experience or existing solution approaches for this application. Automation approaches for this group of tasks would include tools for code profiling [42] and techniques for architecture estimation, such as worst-case execution time analysis [43] or performance estimation. FRANCIS-V does not incorporate such identification techniques.

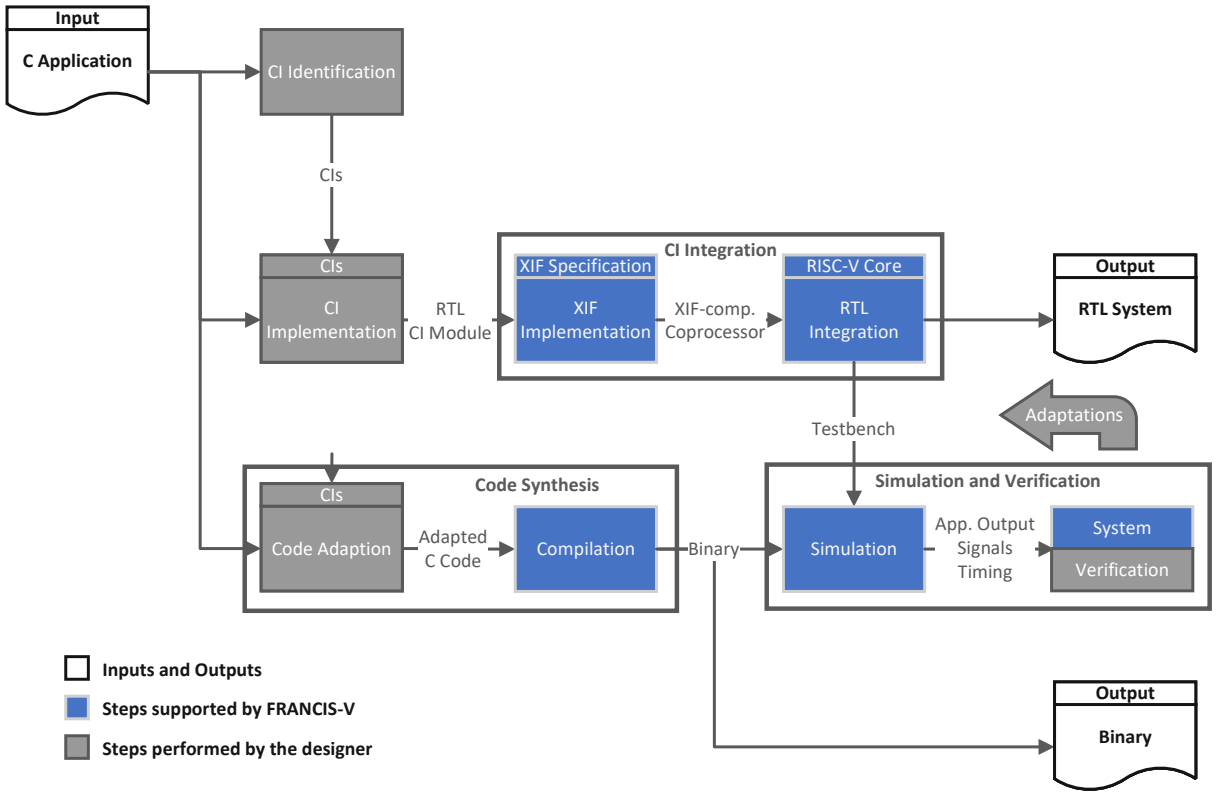


Figure 4.1: Flow diagram of the typical CI development process. The process is divided into steps, represented by the rectangular and colored (blue and gray) boxes. The steps are categorized into five groups, namely, *CI identification*, *CI implementation*, *CI integration*, *code synthesis*, and *simulation and verification*. The groups consist of either one step or more substeps, represented by the boarders around them. The arrows illustrate the step's inputs and outputs, the curved and white-colored boxes illustrate the input and outputs of the process. Blue steps are supported by FRANCIS-V, gray steps remain to be done manually by the designer. System verification is facilitated but not fully automated by FRANCIS-V. The bent adaptation arrow represents implied iterations between steps of the design process.

CI implementation

After choosing suitable CIs, they have to be realized in hardware (*CI Implementation*) with HDL languages in RTL. In our definition, this hardware generation task is split from the subsequent integration task. One major way of automating this task would be using High-Level Synthesis (HLS) to generate RTL hardware out of the given source code. FRANCIS-V does not contain such hardware generation approaches.

CI integration

In the next step, the resulting CI module has to be extended with hardware units that support XIF (*XIF implementation*). The resulting XIF-compatible coprocessor must be able to manage the timing and data format of the XIF subinterfaces to achieve compatibility with the main core. The subsequent step is

the *RTL integration* of the coprocessor and the main core into the RISC-V system, resulting in the RTL system output of the process. This step also includes the development of a testbench for subsequent simulation and evaluation. For these steps, approaches for reducing the manual effort comprise an automated XIF coprocessor or *wrapper* generation and integration into a predefined system including a chosen main core. This wrapper generation and system integration are significant contributions of FRANCIS-V, as detailed in Section 4.2.

Code synthesis

In parallel to developing the RTL system, a compilation step has to generate the binary of the application for the chosen processor, ISA, and CIs[1]. Before this compilation, the designer has to adapt the parts of the C code that are realized with CIs in hardware. The resulting *adapted* source code is further *compiled*, resulting in the binary that serves as an output of the process.

This division in code adaption and compilation within our defined process limits the code synthesis to approaches where the designer adds the CIs to the code beforehand. Consequently, the compiler does not have to interpret the CIs semantically or know their behavior since the CIs are already present in the binary at these explicitly defined code positions. The subsequent compilation step is thus simplified and can be performed with the standard ISA of the chosen main core. Hence, resulting automation tools are limited to an automated adaption of the compiler to the given basic ISA, as is supported by FRANCIS-V.

Simulation and verification

In the last steps, the generated design has to be evaluated. For that, the testbench and binary are *simulated*. As a result, the correctness of the code execution and timing can be analyzed, as represented by the *system verification* step. Tools and frameworks for simulation and verification of the generated design include scripts and testbenches for automated simulating and testing of the developed CI-based system, as provided by FRANCIS-V. Further approaches like automated test case generation or formal verification methods, albeit being a promising alternative for this problem, are not scope of this thesis.

[1]Jain et al. differentiate between two different Code Synthesis approaches, namely, retargetable *Compiler* and *Code Generator*.

4.2 Proposed framework

To answer the second research question, we propose the open-source framework FRANCIS-V for integrating CIs into RISC-V systems. FRANCIS-V addresses the deficiency in flexible CI integration tools for the emerging RISC-V ISA. We define *flexibility* in this context as a solution that is applicable and reusable for a broad range of applications and compatible with several RISC-V cores. The framework adopts the interface-based approach discussed in Section 3.1 to achieve this desired flexibility.

FRANCIS-V is structured into a software, hardware, and simulation part, respectively. Figure 4.2 gives a high-level overview of the features and relevant components of the framework. FRANCIS-V assumes that the designer provides the C application with included CIs and the associated CI hardware modules[2]. Based on these inputs, it integrates the CI module in a coprocessor with XIF support (*XIF wrapper*) generated by the framework. FRANCIS-V further provides code compilation, HDL simulation, and verification capabilities to generate and test the CI-based system.

Focus on CI integration

In comparison with the CI development process analysis from Section 4.1, FRANCIS-V mainly covers CI integration and further features code synthesis, simulation, and verification capabilities, while CI identification and implementation are out of scope. This thesis' focus on integration is justified due to its crucial role in the design process. Furthermore, compared to the CI identification and implementation, CI integration is evaluated as the most realistic step to be contained in a single master thesis and solved with open-source tools. The additional compilation, simulation, and verification features are added to FRANCIS-V because they naturally complement its proposed integration methodology.

Albeit being out of scope for this thesis, FRANCIS-V is still designed with CI identification and implementation in mind. The framework can seamlessly integrate CI modules generated by HLS-based implementation solutions[3]. Similarly, CI definitions generated by automated identification methods can be included in the subsequent framework-based design flow.

[2] In other words, the designer has to prepare the adapted C code and implement CI hardware, either manually or with external hardware generation solutions such as HLS tools.

[3] The given CI hardware does underlie some assumptions described in Section 4.2. However, all of these assumptions can be fulfilled by either manual or HLS-generated hardware designs.

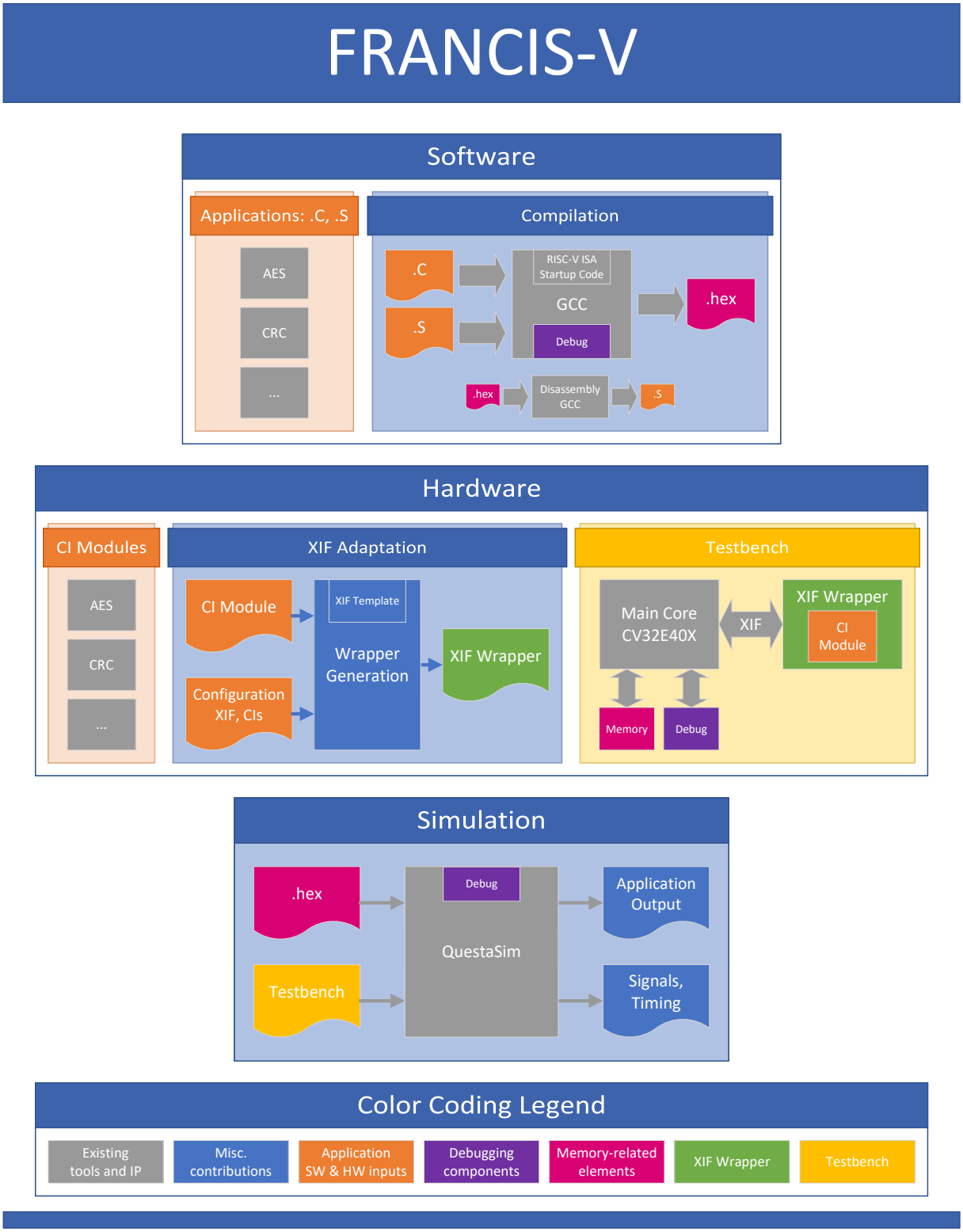


Figure 4.2: High-level overview of FRANCIS-V's features and components. The color gray illustrates existing standard components and IP by other designers. The colored parts represent the contributions of this thesis and emphasize various affiliations within the framework. Orange, purple, magenta, green, and yellow emphasize the relations regarding the application software and hardware inputs, debug capabilities, application binary and instruction memory, XIF coprocessor or *XIF wrapper*, and RTL system embedded in a testbench, respectively. Blue is the general color of choice for the other framework's parts. *AES* and *CRC* indicate the utilized test cases of this thesis (discussed in Section 5.4).

Supported instructions

Figure 4.3 illustrates the type of CIs that FRANCIS-V supports. We limit the instruction type to single-cycle *R-type*[4] instructions with two source registers (*rs1*, *rs2*) and one destination register (*rd*). The instruction type is limited to meet the requirements of a master thesis. Future extensions beyond these limitations, such as load/store or multi-cycle instructions, are discussed in Section 7.6.

Due to the single-cycle limitation, FRANCIS-V supports every software function as a CI module that can be realized in combinational hardware. This support implicates that any function without time dependence can be realized using a single CI as long as the resulting critical path stays within the given system requirements. Functions with an exceedingly lengthy critical path or time-dependent functions can be realized using multiple CIs. Therefore, the framework facilitates a wide range of potential applications, as long as they are realizable with reasonable effort in multiple CIs, with each respective CI being implemented in combinational hardware.

Software

The software aspects of FRANCIS-V (the software block in Figure 4.2) cover the code compilation and software debugging capabilities of the framework. These features address the binary creation as well as the software parts of the verification block in the development process from Section 4.1.

This software process starts with the given C application. After manually adding the CIs by the designer, the adapted source code is fed into FRANCIS-V. The framework provides the compiler structure to generate the binary out of this given application. This binary is subsequently used in the simulation and deployment of the developed system. Apart from compilation, the framework further features

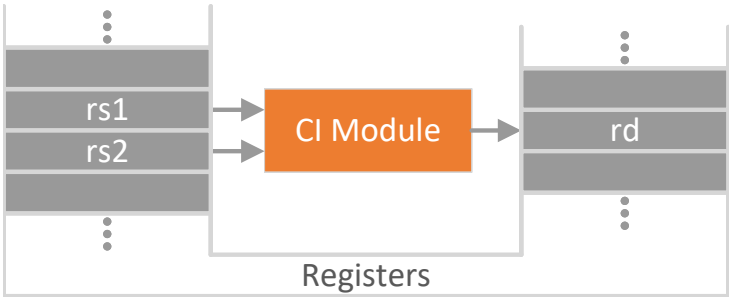


Figure 4.3: Schematic high-level overview of the type of supported CIs. The instruction type is limited to single-cycle R-type instructions with two inputs and one output.

[4]R-type stands for Register-type instructions, as explained in Section 2.1.

debugging functionality, including assembly and disassembly of the source and binary as well as libraries for simulation control from within the C code.

It is important to note that the compiler only uses the CIs previously added by the designer and translates them into a binary format, effectively inserting them at the respective predefined code parts. The compiler does not have behavioral information on the ISEs or, in other words, cannot interpret the semantics of the added CIs.

Hardware

FRANCIS-V's contribution regarding *hardware* components and features, as illustrated in Figure 4.2, consists of the XIF coprocessor or *XIF wrapper* generation and integration into the predefined RTL system (*Testbench* block). These components contribute to the CI integration part of the development process in Section 4.1.

The framework takes a given CI hardware module as an input, previously designed by manual implementation or with the help of HLS. The chosen CI module is integrated inside an external coprocessor that supports XIF. This coprocessor is called *XIF wrapper* in the thesis. The proposed *XIF template* serves as a generalized base coprocessor for the framework to generate the XIF wrapper specific to the given application's CIs.

This integration methodology is a major contribution of FRANCIS-V. At the time of writing this thesis, XIF is only given in the form of a written specification in natural language as a GitHub project and respective documentation [7], [21]. The authors do not provide a generalized adapter or template for XIF nor an example coprocessor implementation[5]. FRANCIS-V's wrapper generation methodology fills this gap of a generalized XIF implementation to simplify and shorten the RISC-V design process.

The framework further integrates the XIF wrapper with the RISC-V CV32E40X main core, resulting in a complete system. The RTL system is subsequently embedded into a testbench for simulation. In addition, the resulting system features the respective hardware units for the simulation control from within the C application.

[5]The closest form of reference implementation are the processor-specific modules handling XIF inside the CV32E40X.

Simulation

The previous software and hardware contributions converge in the simulation phase of FRANCIS-V. The framework simulates the proposed testbench and application binary. Regarding the defined development process, this phase covers the simulation block and parts of system verification in Figure 4.1. As mentioned in the software and hardware paragraphs, FRANCIS-V further offers debug capabilities for simulation control and output visualization of the C application. Further use-case-specific evaluations, such as a comparison to an Instruction Set Simulator (ISS) and timing analysis, are presented in Chapter 5 and Chapter 6.

The testbench and debug capabilities are features to assist the designer in verifying the generated system. However, the designer must still perform the verification step themselves, such as generating suitable test cases for their application and assessing the simulation results. Therefore, FRANCIS-V partly supports the verification process.

4.3 Design time estimation

To address the third research question, we evaluate how much the design time decreases when utilizing FRANCIS-V in the next step. We defined the typical process in Section 4.1 as a process that exclusively relies on manual design steps by the designer to develop the required CI-based system. Based on that definition, we estimate how much time a designer approximately needs to perform each process step. From the resulting values, we derive the time the designer saves when utilizing FRANCIS-V during their development process. With the results, we aim to identify trend values on the expected design time reduction to offer approximate, qualitative results of FRANCIS-V's potential.

Among the different estimation approaches proposed in literature, the common approach of expert estimation for quantifying the required design time is chosen for this thesis. Our methodology for this estimation consists of conducting an expert interview with a confined group of participants who estimate the expected duration of the tasks to be performed during the development process.

Estimation techniques

According to Jørgensen [44], expert estimation is the most popular estimation strategy for software development. The author defines *expert estimation* as estimation work being conducted by a person

recognized as an expert on the task. He describes this estimation work as consisting of both intuitive and explicit reasoning elements by this person.

There are known accuracy limitations and biases entailed with expert estimations. Jørgensen describes experts as biased towards over-optimism, meaning that the effort and time required to execute a task is typically underestimated. Furthermore, the author identifies over-confidence in the accuracy of the estimated values. However, his results indicate no favor in the use of expert estimation or formal estimation over the respective other technique.

Expert estimation, as described by Jørgensen, is assessed as this thesis's most promising and feasible technique for design time quantification. Still, a superficial preliminary investigation of the topic is conducted to preclude other alternatives, namely, model-based and metric-basd estimation approaches.

Bazeghi et al. propose *μComplexity* [45], a model-based methodology for design effort estimation and measuring of processor-based designs. The methodology divides the system into components, applies a statistical regression model to estimate the design effort, and scales the result with a productivity factor. The authors advise to recalibrate the model based on used EDA tools, recent projects, and the design team's productivity. These factors make the approach unsuitable and too cumbersome for this thesis since FRANCIS-V originates from a small team without recordings of required time for prior projects.

Bazeghi et al. further identify single-metric estimators based on metrics like *lines of code* or *fan in of logic cones* to roughly correlate to design time. This metric-based approach is not further pursued in this thesis due to its uncertainty.

Metric-based and model-based approaches only consider the hardware and software development time in their estimations. We expect a significant part of the development process to be knowledge acquisition by the designer, such as familiarizing themselves with the working principle of XIF. By contrast, the participants of the expert estimation can give their opinion on the amount of training time a designer likely requires.

Development tasks

In order to reduce the complexity of the estimation for the experts, the development process is split into tasks that are individually assessed on their expected design time. This development process assumes the application and the chosen CIs as inputs. The tasks build upon these inputs, detailing the CI properties and source code additions, and then continuing with hardware implementation and integration.

We define the following nine tasks:

1. **Opcode selection**: The opcodes of the given CI shall be selected. They shall not be already in use and legal within the RISC-V ISA specification.
2. **C code adaption**: The given C application's code parts to be implemented as CIs shall be substituted with the respective CIs.
3. **Compiler setup**: The RISC-V compiler toolchain shall be set up.
4. **CI implementation**: The CIs shall be implemented in an execution unit using combinational HDL code. This task shall be estimated for two concrete use cases, AES and CRC[6].
5. **XIF learning**: The designer shall familiarize themselves with XIF, including handshakes, timing properties, and data of the subinterfaces.
6. **XIF implementation**: The CI module shall be integrated into a coprocessor supporting XIF.
7. **RTL integration**: The coprocessor shall be integrated with the CV32E40X and supporting modules, resulting in the RTL system testbench.
8. **System verification**: The design shall be verified on correct timing and behavior by simulating the system's testbench[7].
9. **Synthesis**: The RTL system shall be synthesized on an FPGA.

The tasks are derived from the typical development process steps defined in Section 4.1. However, they are adapted for a clearer and simpler explanation during the time-limited interview. Figure 4.4 illustrates these adaptations. Since we assume already predefined CIs, the CI identification step is reduced to selecting suitable RISC-V opcodes for them in task 1. Tasks 2 to 4 correspond to their previous step counterparts. Tasks 5 and 6 divide the XIF implementation step into two parts to emphasize the training time needed to acquire knowledge about XIF during the interview since the designer does not need to conduct this task when using FRANCIS-V. Task 7, RTL integration, is also transferred as is, while simulation and verification are condensed to a single system verification task in task 8. Last, task 9 is added to include synthesis into the estimation since this step, albeit not the focus of this thesis, is still an interesting and important step in a practical development process.

[6] To give the experts a more precise context, the concrete *NIST Standard 128-bit* and *CRC-16-CCITT* algorithms where specified within the interview.

[7] *8. System verification*, the verification of the system, is explicitly differentiated from the verification of the single components. E.g., verifying the correctness of the CI hardware implementation prior to the subsequent XIF integration is included in the respective task *4. CI implementation*, and not in the subsequent system verification task.

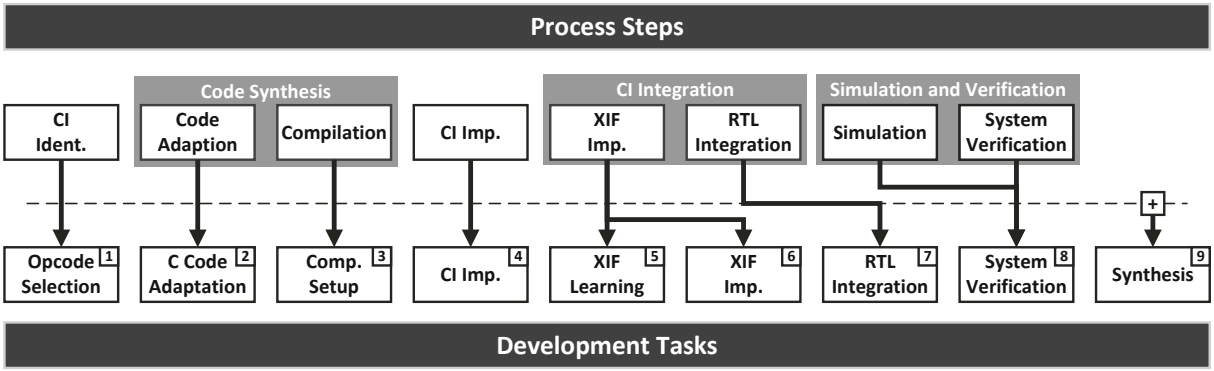


Figure 4.4: Derivation of the development tasks for the expert interview based on the steps of a typical process defined in Section 4.1.

The number and scope or *granularity* of the tasks to be estimated are chosen over a set of other possible configurations to focus on the important features of FRANCIS-V without neglecting the remaining steps. Tasks 5 and 6 (*XIF learning* and *XIF implementation*), for example, could also be encapsulated in a single task but are split for a more accurate estimation of the XIF integration process. On the other hand, depending on the application, complex tasks like task 4 (*CI implementation*) could be split into further subtasks for a potentially more accurate estimation. Note that the tasks are also not designed to be within the same order of magnitude, time-wise. Instead, it was expected prior to the conducted interview that simpler tasks such as *Opcode selection* might potentially be scored lower than cumbersome tasks like *CI Implementation* by a significant factor, as confirmed by the results in Section 6.1.

Increasing the task granularity and balancing the expected magnitude difference of the individual tasks may lead to more accurate estimation results with less uncertainty, especially in rather comprehensive tasks like task 4. However, a more fine-grained division with a detailed description of the possible implementations and many subtasks might entice the participants to think of certain predetermined solutions. Preferably, the task descriptions are generic enough not to lead participants to a specific development and implementation approach to prevent biased opinions.

Participants

The participants of the expert estimation were five senior engineers from the department of Siemens, with which the thesis is developed in cooperation. These experts specialize in varying fields of SoC, ASIC, and FPGA designs. Each chosen participant is well-versed in a different hardware, software, or system development are. The specific participants were selected based on their long-time experience.

The number of selected participants and their origin in the same company entails some shortcomings. Due to the relatively small number of five experts, the estimation likely contains a non-negligible degree of uncertainty. Furthermore, the participants are all part of the same department at Siemens. This limited participant diversity likely constrains the generalizability of the results to a group of designers with varying backgrounds and job specialization. Repeating the interview involving other participants with different backgrounds would likely yield deviating design time results. The choice and size of the participant group are attributable to the restricted requirements of the master thesis.

However, although the partaking experts share a similar level of experience, their diverse specialization areas can be stated as a strength of this estimation. Despite their mutual experience in SoC design, the enlisted participants have varying backgrounds and specialization domains, ranging from software development over hardware engineering to co-design, testing and verification, and system design. Furthermore, the choice of experienced senior engineers over junior engineers is made to get a more accurate and experience-based opinion[8]. The experts' long-term experience in their designated area of expertise likely increases the accuracy of the estimation.

Interview

The expert estimation was conducted by interviewing the participants during an online meeting. The interview was planned to take thirty minutes and include all five participants. The first phase consisted of a concise ten-minute introduction to set the context of the subsequent estimation tasks. This phase presented the topic, goals, the two exemplary use cases, and concepts of the thesis, such as XIF. It was followed by five minutes of answering additional questions.

The remaining 15 minutes were reserved for the second phase, consisting of the interviewing process. There was a short written and complementary oral introduction of the purpose and content of each of the nine tasks to estimate. After this description, the development time expected for the task conduction in workdays was anonymously submitted by each participant using the survey tool *Microsoft Forms*. After each submitted estimation, the participants were invited to justify their decisions and optionally discuss deviating opinions. The participants also had the option to abstain from single tasks if they were not well acquainted with it, e.g., if they had not used or implemented an AES algorithm before.

Similar to the rationale on chosen development task granularity, the set length and schedule of the interview have implications for the result accuracy and quality. Within 15 minutes, the participants can

[8]The argument of higher estimation accuracy of expert engineers was also the main reason of excluding our own estimation later in the result analysis (Section 6.1).

only receive a coarse overview of background information, inputs into the process, and requirements and details on the specific tasks. As already reasoned for task granularity, the interview duration and presented context were set to ensure a balance between an accurate and an unbiased result and a limited timeframe of all involved participants, focusing on the most prominent features of FRANCIS-V.

Besides the estimations of the five participants, we also provide our own time estimation of the tasks. This estimation was done before the interview was conducted but after designing and implementing the main body of the framework.

The interview is intended to indicate trends in potential manual development time. It stands apart from a typical extensive study due to the limitations described in the above sections. In the next step, the resulting estimation values of the typical development process are used to infer the time when utilizing FRANCIS-V within this process. This analysis and the results are presented in Section 6.1.

Chapter 5

Implementation

This chapter goes into detail on the implementation and evaluation aspects of FRANCIS-V. The framework is divided into a *Software*, *Hardware*, and *Simulation* segment[1]. Albeit supporting arbitrary single-cycle CIs with two inputs and one output, the designs generated by FRANCIS-V can only be evaluated with specific use cases in mind. We selected the test cases AES and CRC to demonstrate the framework's working principle and to analyze the runtime and utilization metrics of the generated system.

5.1 Software

The software part of FRANCIS-V is responsible for compiling the application's source code. The code is assumed to be given at the start of the development process in C. Before starting the compilation within FRANCIS-V, the designer must insert the CIs chosen to speed up the algorithm into this source code. For this reason, they have to substitute the code segments performed in hardware with their respective CI counterparts by using Custom Assembler instructions. The following code segment lists the CI inside the C code when implementing the CRC use case:

```
1 asm volatile(".insn r 0x6B, 0, 1, %0, %1, %2" : "=r" (crc) : "r" (data_conv), "r" (0));
```

Listing 5.1: Custom Assembler instruction for the CRC use case. It specifies *0x6B* as the opcode and *funct3* and *funct7* values of 0x0 and 0x1, as well as the variable output *crc* and inputs *data_conv* and *0* for resulting check value, input data, and CRC initalization seed of zero, respectively.

[1]Figure 4.2 gives a high-level overview of these three parts. The relevant illustrated segments are repeated in this chapter.

The *asm volatile()* routine allows using C variables within the compiled assembly representation of the source code. It inserts the assembly directive *.insn*, representing the opcode of the CRC CI. This code segment has to be replaced with the original CRC software routine in the C code.

The resulting adapted code serves as input to the compileflow of FRANCIS-V, as illustrated in Figure 5.1. The framework features compilation to generate the binary out of the adapted source code. The binary is subsequently executed on the chosen main RISC-V core CV32E40X. Therefore, the compilation is done for the particular RV32IMA ISA of the CV32E40X. In addition, startup code is added, among others, for initialization tasks such as interrupt, exception, and system call handling as well as pointer initialization. In addition to the compilation feature, FRANCIS-V offers disassembly to help the designer locate potential bugs within the code in its assembly representation.

The compile toolchain of the framework is based on the GNU Compiler Collection (GCC), an open-source collection initially developed for the GNU operating system [46]. It supports C and various other languages. As described by Poorhosseini et al. [47], GCC is the first compiler to support RISC-V and, to this day, is the most popular one alongside the LLVM compiler infrastructure [48]. Poorhosseini et al. find that LLVM and GCC perform similarly regarding binary size and resulting clock cycles, making both compilers viable candidates for this thesis. Following an initial evaluation, GCC is chosen over LLVM due to the more intuitive integration of CI Instructions in the source code using inline assembly. However, it has to be emphasized that both compilers support this method.

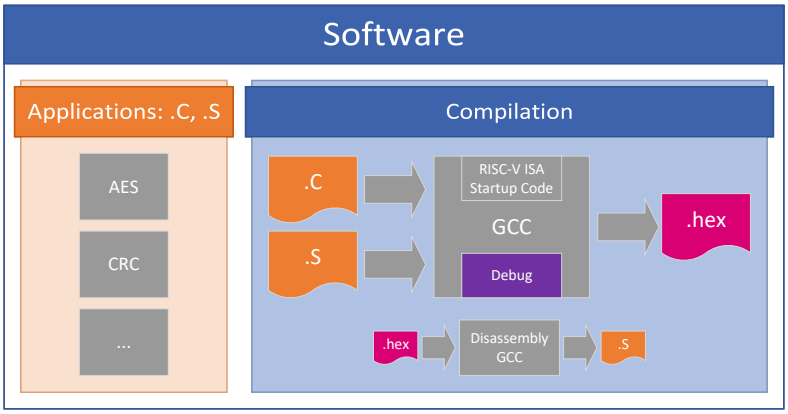


Figure 5.1: Software aspects of FRANCIS-V (segment of high-level diagram in Figure 4.2). The two groups represent the compilation capabilities of the framework and the source codes of the proposed test cases. The application's adapted C code that contains the CIs and the startup code of the CV32E40X are compiled by the framework on the core's ISA.

5.2 Hardware

The hardware development methodology of FRANCIS-V manages the integration of CI hardware into a predefined system. The CI hardware unit is required as an input given by the designer. The hardware unit must be combinational due to the limitation to single-cycle instructions. Moreover, a specific scheme of input and output ports of the unit is required. The 32-bit input ports *rs1* and *rs2* (register source) and the 32-bit output port *rd* (register destination) represent the inputs and output of the CIs. The 32-bit encoding of the instruction is also given to the hardware unit to allow for CIs that change functionality depending on certain bitfields of the opcode. Furthermore, since only one hardware unit represents all CIs, a selection signal named after the respective instruction has to be present for each CI to choose which CI the hardware unit shall perform. For the use case CRC with one single CI, e.g., the port definition may look as follows:

```
1  module CRC_module #()
2  (
3      input  logic [31:0] rs1,
4      input  logic [31:0] rs2,
5      output logic [31:0] rd,
6      input  logic [31:0] instr,
7      input  logic instr_CRC,
8  );
9  ...
10 endmodule
```

Listing 5.2: Port definition of the CI module for the CRC use case, defining the register inputs and output, instruction opcode, and instruction enable signal.

A module with multiple CI would implement more input signals like *instr_CRC*. These adaptions to the port requirements have to be done by the designer prior to using FRANCIS-V. The framework is written in SystemVerilog, but the hardware unit can be written in Verilog, SystemVerilog, or VHDL. The hardware process of the framework is illustrated in Figure 5.2. It integrates the given CI hardware unit into a wrapper compatible with XIF. The wrapper is further integrated into a predefined RTL system with the CV32E40X. The system testbench is subsequently used for the system's simulation process.

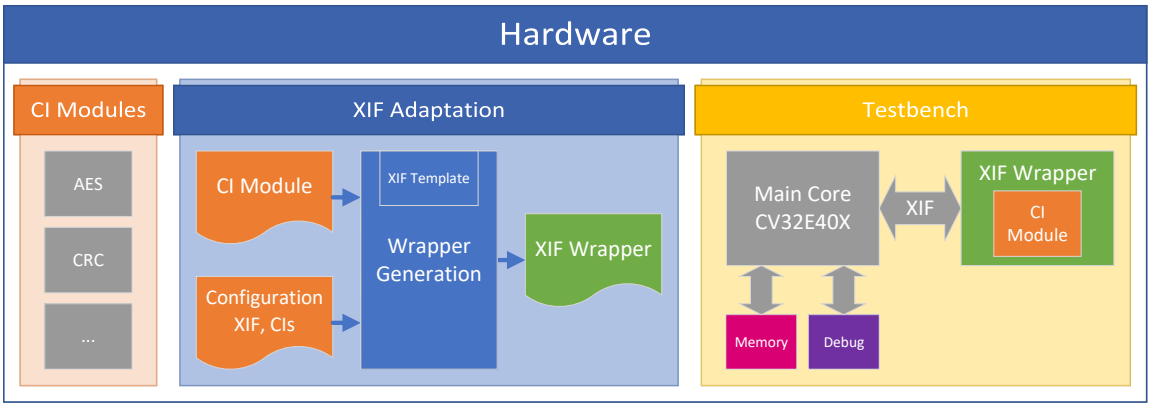


Figure 5.2: Hardware components and features of FRANCIS-V (segment of high-level diagram in Figure 4.2). The adapted CI modules of the first block as well as naming and opcode information of the CIs are used to generate the XIF-compatible wrapper out of a predefined template, illustrated in the second block. The wrapper is integrated into the predefined system testbench together with the CV32E40X, as shown in the third block.

5.2.1 Wrapper Generation

The adapted CI module serves as an input to the framework's *wrapper* generation methodology, as illustrated in the middle block of Figure 5.2. This wrapper generation is realized using a Python script. Besides the CI module, it requires information about the CIs' name and opcode in a JSON format. The generated XIF wrapper manages the execution of the CI issued by the main core, effectively fulfilling the role of a coprocessor.

The wrapper consists of the given CI hardware unit, an instruction decoder, and supportive logic to handle instruction and data requests and responses of XIF. A high-level overview of the architecture is illustrated in Figure 5.3. The *decoder* accepts all instructions issued by the CV32E40X that the designer previously defined in the JSON configuration file. The decoder further extracts the operands and instruction data and passes them to the *CI module*. The module performs the instruction and delivers the result to the *result generation*. Until this point, only combinational operations are performed. Thus, the execution is within the same cycle as the instruction is issued. In the second cycle, the result generation stores the result. Once the main core commits the instruction, the wrapper offers the CI's result on the Result interface and informs the main core. This commit may already happen in the second cycle parallel to storing the result. The main core can confirm the result transmission in the same or the following cycles, which completes the CI execution process. The result generation also manages several corner cases, such as a simultaneous commit of the current and issue of a subsequent instruction.

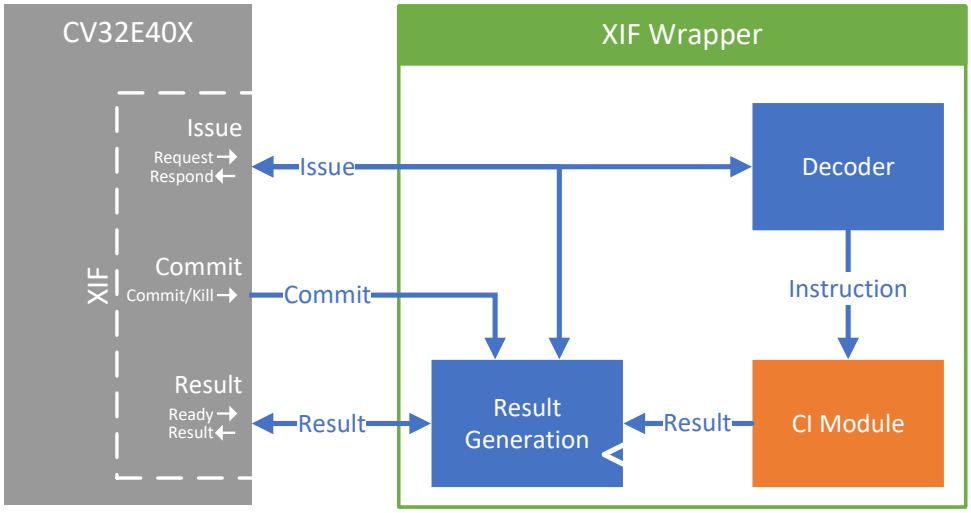


Figure 5.3: High-level overview of the architecture of the XIF wrapper. Three of the six XIF subinterfaces, namely, *Issue*, *Commit*, and *Result*, are used by the CV32E40X to issue CIs, to commit or kill them, and to receive their results, respectively. Within the wrapper, in the same cycle of the issuing, the decoder accepts and decodes the instruction and the CI module performs it. The result generation block outlines the synchronous components of the wrapper, managing the result delivery and intermediate result storage based on the CV32E40X's subsequent issue and commit behavior.

The wrapper architecture can be compared with the other XIF-based coprocessors consisting of decoder, FIFO, CI module, register files, and control logic, as summarized in Section 3.3. Decoder and CI modules similarly exist in the wrapper, while the FIFO and control logic functionality is implicitly included within the Result Generation block. The wrapper can operate without a separate register file due to the limitation to R-type single-cycle instructions. Section 7.6 further discusses potential architectural extensions to overcome these limitations.

The XIF wrapper is generated based on a *template*. The template includes the predetermined components and structure of the wrapper. It further contains placeholders on the application-dependent parts of the wrapper, namely, at the CI module instantiation, the instruction decoder, and at selected XIF-related internal signals that scale with the number of CIs.

5.2.2 System Integration

In the next step, the generated XIF wrapper is integrated within the predefined RTL system, as illustrated in the third block of Figure 5.2. The system connects the wrapper with the CV32E40X over XIF.

As the main core, the CORE-V processor CV32E40X is chosen since, at the time of writing the thesis, it is the only processor with XIF support. Furthermore, the core offers a good trade-off between size and performance for embedded applications targeted by FRANCIS-V. Its four stages are well-suited for an

efficient XIF communication, as shown in Section 6.3. CV32E40X executes the application's binary. It outsources each instruction it does not recognize (hence, every CI) to the wrapper via XIF, controls if the instruction shall be committed or killed, and further processes the received results.

The binary generated by FRANCIS-V's compiler toolchain is located inside the instruction and data memory of the HDL system. The memory is realized as a dual-port memory with one read-only port for the instruction and one read-write port for the data interface. The CV32E40X's memory interfaces comply with the OBI specification. A separate module manages the protocol conversion between the CV32E40X and the memory.

5.3 Simulation

The binary and RTL system generated by the software and hardware parts are combined in the subsequent simulation part of the framework, as illustrated in Figure 5.4. FRANCIS-V simulates the system embedded into a testbench with the binary running on the CV32E40X. TCL scripts realize the simulation routines inside FRANCIS-V. Questa advanced simulator (QuestaSim) [49] is chosen for this HDL simulation. The tool is part of Siemens' Questa verification solution, a simulation and debug engine for complex FPGA and SoC design. It supports mixed simulation of HDL languages, including VHDL and SystemVerilog.

QuestaSim is the only commercial tool used in the otherwise open-source framework. The open-source alternative Verilator was explored in the early phases of FRANCIS-V's development but was later discarded. Its developers claim similar performance compared to closed-source simulators such as QuestaSim while being free of license fees [50]. However, one benefit of using QuestaSim over Verilator is its debugging features to facilitate the framework development process. Another consideration is

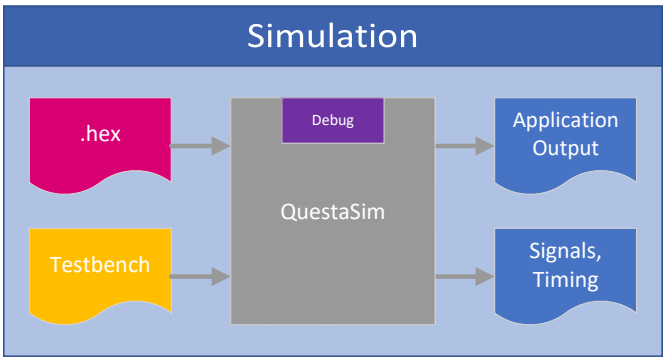


Figure 5.4: Simulation aspects of FRANCIS-V (segment of high-level diagram in Figure 4.2) representing the HDL simulation within the framework. The simulator QuestaSim is used to simulate the system testbench, with the application's binary running on the CV32E40X.

the support for mixed simulation of various HDL languages within the system, allowing for more flexibility regarding the framework's RTL design and the CI hardware units given to FRANCIS-V. Furthermore, during this exploration phase, the usage of Verilator seemed less intuitive than other simulators. The developers of Verilator themselves also imply this observation, as they suggest the open-source alternative Icarus Verilog [51] for small projects [50].

The intuitive and advanced simulation features are decisive for selecting QuestaSim for FRANCIS-V. While adding further simulator support for FRANCIS-V is not within the scope of this thesis, QuestaSim could, in general, be substituted with the aforementioned open-source alternatives.

One use case of the HDL simulation is to verify that the application executes as intended regarding the correctness of its results and timing properties. FRANCIS-V offers a C debugging library to aid the designer during the verification process. It consists of software routines for simulation control and debugging within the C source code:

- **sim_printf()**: This routine enables printing a string to the QuestaSim console during simulation.
- **sim_end()**: Calling this routine ends the simulation.
- **sim_sig()**: This routine sets a debug signal within the HDL testbench to simplify the localization of specific code parts within the HDL simulation.

When called within the C code, the routines write to specific reserved memory addresses otherwise not utilized by the CV32E40X. A separate debug module within the system identifies these write attempts and forwards them to the testbench. *sim_end()*, when triggered, causes a *$finish*-command, while *sim_sig()* sets a special debug signal that is active for one cycle. *sim_printf()* splits the given string into character symbols, with each character triggering individually, causing a console output.

5.4 Use cases

The algorithms CRC and AES serve as exemplary use cases for FRANCIS-V. AES is chosen to quantify the considered system metrics. CRC is utilized as a more straightforward test case to demonstrate FRANCIS-V's features and working principle. It has to be noted that the framework's applicability extends beyond these examples. The framework can be utilized for a broad range of applications.

5.4.1 CRC

Cyclic Redundancy Check (CRC) is an error-detecting method, among others, used for data transmission between processors. The algorithm represents the input as a binary number and divides it by a predefined value. This divisor can be described with a *checking* polynomial. The remainder of the division, the check number, is appended to the input number before transmission. After transmission, the algorithm is reapplied to the transmitted data. If the result is zero, the message is supposed to be correct. Otherwise, a transmission error is assumed [52]. There are several standards and implementations of CRC with varying details, such as different polynomials or input bits [53].

Since CRC is a rather straightforward algorithm, we chose to implement it using a single CI. The concrete implementation uses a 32-bit input word and the polynomial $x^{16} + x^{12} + x^5 + 1$. The required inputs to FRANCIS-V are the hardware unit and a C algorithm. The combinational Verilog hardware module is generated using the CRC HDL generator by Büsch [54]. A top module is manually written for the hardware unit to rename and extend the inputs and outputs of the unit for subsequent use in the FRANCIS-V wrapper generation.

As source code, a C test program is written for comparing the CI hardware implementation with a software-based CRC algorithm. The CI is defined using a Custom Assembler instruction. The software counterpart is adapted from a *CRC-16-CCITT* implementation by the AutomationWiki [55]. The program performs the CRC routine on the selected input data *0x00006162* ("*ab*" in ASCII representation). The resulting check value is subsequently appended to the input data (*0x616274FF*), representing the data to be transmitted in a conceptual application. Applying the CRC routine on this data yields zero, assuming a correct algorithm. Afterwards, the program alters the previously appended data to simulate a transmission error. A third execution of the CRC routine should yield a value other than zero, implying a recognition of the error. The whole routine is performed twice, once utilizing the pure CRC software routine and once using the CI.

Based on these inputs, the framework generates the binary and RTL system and simulates the resulting design. The following simulation output illustrates that both implementations operate as expected and generate identical outputs:

```
# SW Test:
# Check number:  0x74FF
# Correct transmission test:  0x0000
# Wrong transmission test:  0x1EF0
#
# CI Test:
# Check number:  0x74FF
# Correct transmission test:  0x0000
# Wrong transmission test:  0x1EF0
```

Listing 5.3: Simulation output for the CRC test routine, indicating identical results for both software and CI-based hardware implementation of the CRC algorithm.

5.4.2 AES

Advanced Encryption Standard (AES) or *Rijndael* is a symmetric block cipher. It won a contest for a new encryption standard organized by the National Institute of Standards and Technology (NIST). It is capable of encrypting and decrypting 128-bit data blocks using 128-, 192-, or 256-bit-sized keys in 10, 12, and 14 rounds, respectively. Among others, the requirements for contest participants was the possibility of implementing the algorithm in software on 8-bit processors as well as using dedicated hardware implementations [22], [26].

The choice of selecting AES for evaluating FRANCIS-V is based on various reasons. First, it is a suitable candidate for evaluating a CI-based framework for hardware integration like FRANCIS-V since it can be implemented relatively cheaply in hardware and software. Another argument is the popularity of the algorithm and the availability of hardware implementations. Not only is AES a popular example use case in related literature on ISEs and used in various applications, but also numerous solutions for AES implementations in hardware exist [56].

In addition, the available RISC-V AES standardization work (presented in Section 2.5) greatly facilitated the development of FRANCIS-V's AES use case. Apart from minor adaptions and runtime optimizations, the featured C source code and CI hardware modules are well-suited as inputs for FRANCIS-V to generate the CI-based system for the AES use case. Furthermore, the standard's benchmark collection features two AES software solutions (*Reference* and *ttable*) for comparing the CI hardware solution with software-only

approaches. The standardization work also facilitates the runtime evaluation of FRANCIS-V's generated AES system design proposed in Section 5.5.1.

As detailed in Section 2.5, the crypto specification defines four CIs (*aes32esmi*, *aes32esi*, *aes32dsmi*, *aes32dsi*) for subtasks within the rounds of the AES algorithm. The CIs are also reused in the algorithm's key scheduling segments. All four CIs are implemented in combinational RTL modules and encapsulated into one unit. This unit is manually instantiated in a top module to define the input and output ports according to FRANCIS-V's needs.

The C program to test the AES hardware and two software implementations comprises four code segments, as illustrated in Figure 5.5. All three implementations pass through this sequence of the four segments, respectively. In the first step, the key is expanded (Key Scheduling Encryption or *KSE*) prior to the subsequent encryption routine (*ENC*) of the given plaintext, yielding the ciphertext. Afterwards, key scheduling (*KSD*) is executed again, followed by the decryption (*DEC*) of the ciphertext to the original plaintext. The resulting plaintext on the output was expected to be equal to the plaintext on the input, which is the case for all three simulations[2].

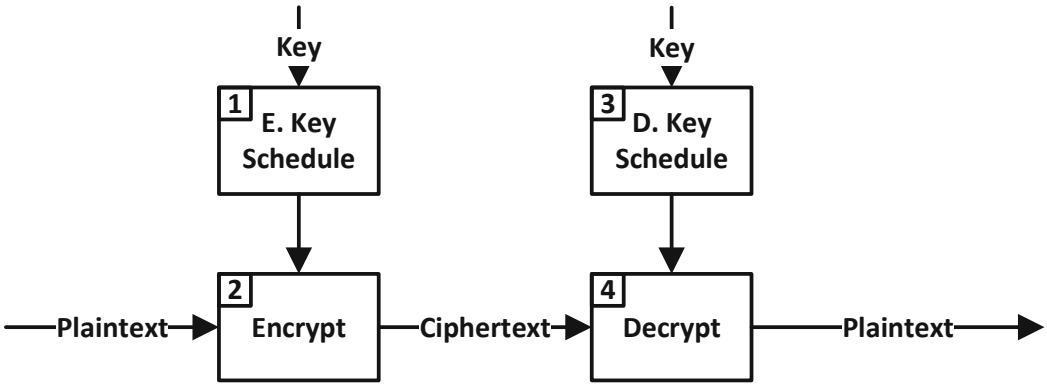


Figure 5.5: High-level flowchart of the AES test program. It is traversed three times for the two software implementations and the CI hardware implementation, respectively.

[2]The following defined inputs and resulting outputs are present in the test program:

- Key: `2b7e151628aed2a6abf7158809cf4f3c`
- Plaintext Input: `3243f6a8885a308d313198a2e0370734`
- Ciphertext: `3925841d02dc09fbdc118597196a0b32`
- Plaintext Output: `3243f6a8885a308d313198a2e0370734`

5.5 Evaluation setup

The HDL simulation features of FRANCIS-V allow for verifying that the generated designs operate as intended regarding both functionality and timing. In addition, it is of interest to assess how competitive the resulting designs are in contrast to related solutions. Therefore, we evaluate the runtime saved and the utilization overhead caused by the implemented CIs in contrast to a software-only approach without custom hardware for the AES use case.

5.5.1 Runtime evaluation

A comparison between the hardware and software implementations is done to verify the assumption of reduced runtime due to the CI hardware implementation. The base for this evaluation is the AES test program described in Section 5.4.2. We propose a benchmarking script to perform the program with the hardware implementation and two software-only approaches on the same system. The simulation is conducted with FRANCIS-V's QuestaSim-based HDL simulation flow and with the ISS Spike. Section 6.2.1 presents these results.

While the HDL simulation is cycle-accurate, Spike simulates on an instruction level [57] without information on cycle count. Its inputs are the binary of the source code and the RISC-V ISA, including the added CIs. Spike does not consider the particular properties of the CV32E40X, XIF, or implemented hardware in general. Therefore, Spike has reduced accuracy. It is included in the AES benchmarking process to verify the HDL simulation by comparing the instruction count of both simulations. Note that, while the HDL simulation with QuestaSim is part of FRANCIS-V and applicable for all of the framework's generated designs, the benchmarking script, Spike, and the synthesis are not part of FRANCIS-V. The evaluations are only designed for evaluating the AES use case specifically.

The proposed comparison script and Spike are only set up for AES because they cannot be generalized as easily as the QuestaSim simulation. The script depends on the console output of the specific application. Therefore, it has to be manually adapted to each use case. The framework would have to support adding CI behavior to the simulator to generalize simulation with Spike, which is out of scope for this thesis. The AES CIs, on the other hand, are standardized (see Section 2.5) and, therefore, are already available within Spike for this specific use case only.

5.5.2 Synthesis

In contrast to the CI hardware implementation achieving an advantageous runtime improvement, the synthesized implementation is expected to come with the drawback of increased resource consumption. As mentioned at the beginning of this section, a artly automated synthesis of the generated systems on FPGAs or ASICs is not supported by the framework itself. Rather, the focus lies on integration and HDL simulation.

However, getting a coarse idea of the potential overhead of FRANCIS-V's designs, particularly due to the XIF wrapper and the integrated CI module, still complements the thesis. Therefore, we synthesize and analyze the generated system using the Xilinx tool *Vivado* on the xc7z020clg400-3 FPGA from the Zynq-7000 SoC Family[3]. The resulting utilization values are presented in Section 6.2.2.

The synthesized systems' derivated utilization values only apply to the particular AES use case. The design is synthesized in two configurations to compare the CI-based with a software-only solution. One configuration includes the whole system, while the other system misses the XIF wrapper. The system's module, otherwise instantiated within the testbench, serves as the synthesis' top module. Its inputs are clock and reset, while the trigger and print signals of the debugging unit are reutilized as outputs.

The AES module, XIF wrapper, main core CV32E40X, and supporting modules generated by the framework are all synthesizable with two exceptions. First, the simulation-only clock gating cell within the CV32E40X has to be substituted with a board-specific clock buffer. Secondly, FRANCIS-V's RAM implementation is adapted. In order to maximize resource efficiency while still being able to fit the AES binary, the RAM's size is lowered to 64 kB[4]. The word length of 32-bit is split into four byte-wide memories to realize a write process with byte-enable required by the CV32E40X, resulting in a required memory of 16 kB for each of the four byte segments. The selected FPGA offers 36kbit dual-port RAMs, translating to four BRAMs per byte segment or 16 BRAMs for the entire RAM of the system.

[3]Further informations on the Zynq-7000 Family can be found on the respective Xilinx website [58].
[4]The simulation originally defines several GB of RAM to facilitate potential applications that result in a larger binary than AES.

Chapter 6

Results

This chapter presents and discusses the relevant metrics of this thesis. Given the first and third research questions, we propose the results of the design time estimation and subsequently use them to analyze the potential amount of saved time due to utilizing FRANCIS-V in the development process. In addition, the metrics of FRANCIS-V's generated systems are of interest to assess if the resulting designs are competitive regarding other relevant automation solutions. The particular system metrics presented are the runtime improvements of the CI-based hardware systems over software-only approaches and the resource overhead of the implemented CI hardware. Last, we analyze the influence of XIF on the overall time behavior of the system.

6.1 Design time analysis

The outcome of the conducted expert interview proposed in Section 4.3 is illustrated in Table 6.1. The time in workdays needed for performing the nine design tasks was estimated by the five participants for the two specific use cases AES and CRC. The five participants are denoted with numbers from #1 to #5, the *mean* values of these five opinions are presented in the last column. Two different estimations are given for the *CI implementation* task depending on the specified use case since this task, in particular, is potentially influenced by the application to a large degree. As two participants had no prior familiarity with AES, they did not give an opinion on the respective design task.

Our own estimations are presented in column #0. The tasks about *CI implementation* and *System verification* are not estimated due to our missing experience on these tasks in an industrial setting. This estimation was conducted prior to the interview but after developing FRANCIS-V. Therefore, the

Table 6.1: Results of the interview on estimating the time required for a CI typical development process. The time is given in workdays [*d*]. Participants are labelled from #1 to #5. Our own estimation is labelled with #0. The estimation of task 4 is given for the AES and CRC use case, respectively.

Task			Auth. Est. [d]	Participant Est. [d]					
ID	**Name**		**#0**	**#1**	**#2**	**#3**	**#4**	**#5**	**Mean**
1	Opcode selection		1	1	1	1	1	1	1.00
2	C code adaption		3	3	3	3	2	2	2.60
3	Compiler setup		3	2	2	2	3	2	2.20
4	CI implementation	AES	–	20	–	–	5	8	11.00
		CRC	–	3	3	4	3	3	3.20
5	XIF learning		5	5	3	5	4	4	4.20
6	XIF implementation		5	4	3	4	5	5	4.20
7	RTL integration		10	5	4	8	5	4	5.20
8	System verification		–	10	15	14	10	10	11.80
9	Synthesis		3	4	5	5	5	3	4.40
	Sum (AES)		–	54	–	–	40	39	46.60
	Sum (CRC)		–	37	39	46	38	34	38.80

The five expert participants are included in the mean value calculation, #0 is *excluded*. The AES mean sum is calculated out of the individual mean values (46.60 workdays) rather than averaging the sum values of #1, #4, and #5 (44.33 days).

estimated values of #0 are influenced by our practical experience on the manual process. They are presented for a coarse verification of the solely estimated expert data with values derived after passing through the design process. However, #0 is not included in the *mean* calculation or subsequent evaluation. Our experience on the topic differs from senior engineers due to less overall design experience and more experience regarding the thesis' specific concepts and components, and therefore, would presumably distort the results.

The majority of presented task times estimated by the participants are situated within the same magnitude. However, the relative deviations of the values are still relatively high. The largest outlier is the AES implementation task, ranging from 5 to 20 workdays. In addition, this particular task was not estimated by two of five participants due to a lack of experience with the AES algorithm. Still, the results represent a coarse quantification of the typical development process and provide insights for answering the research questions. Results for the first research question are directly inferred from Table 6.1 and illustrated in Figure 6.1. The most time-consuming subtasks are identified to be CI integration[1] with around 14 workdays, verification with around 12 workdays, and CI implementation with 11 workdays for the AES use case. The implications of these results are discussed in Section 7.1.

[1]XIF learning and implementation and RTL integration combined.

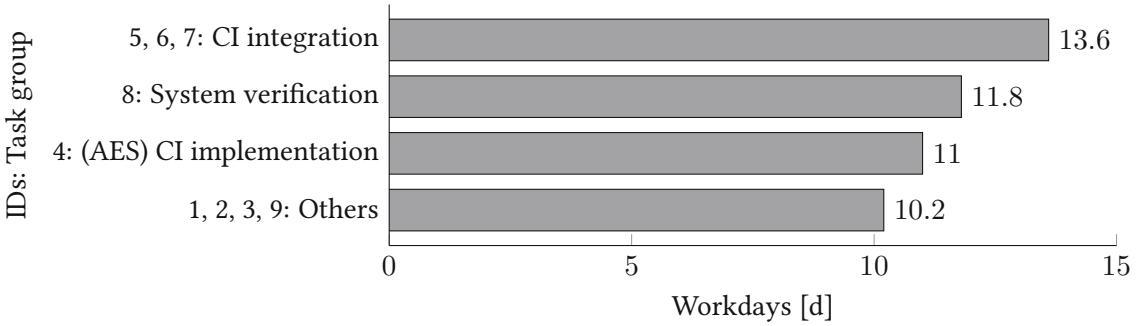


Figure 6.1: Comparison of the most time consuming tasks of the CI development process, according to the results of the expert interview. The time is given in workdays. *CI integration* groups XIF learning and implementation as well as RTL integration. The group *others* consists of opcode selection, C code adaption, compiler setup, and synthesis, all amounting to less than five estimated workdays.

6.1.1 Framework-based design time

To assess the third research question, the time estimation results of the typical development process are used to derive the development time when utilizing FRANCIS-V. Subsequently, the absolute and percentage *reduction* in design time can be inferred, indicating how much potential time can be saved using FRANCIS-V.

The framework-based development process consists of the same nine tasks defined in Section 4.3, summarized in Table 6.2. Tasks 1., 2., 4., and 9. are not supported by the framework and remain to be done manually. Hence, they take the same time whether FRANCIS-V is utilized or not. The tasks 3., 5., 6., 7., and 8., on the other hand, are supported. Apart from task 8., all supported tasks are fully covered by the framework. Therefore, we assume no required design time for these tasks. This assumption neglects the time required to learn about and operate FRANCIS-V. This influence is instead included in the *designer training time* described below. The task *8. System Verification* is partly supported by the framework because FRANCIS-V does offer testbench and debugging capabilities. However, the verification process itself still has to be conducted and assessed by the designer.

Furthermore, while the design time of most listed tasks only scales insignificantly with the chosen use case[2], tasks 4., 8., and 9. do depend on the application to a certain degree. Most notably, the CI implementation task responsible for the hardware generation depends on the chosen CI and the complexity of the particular application. As seen in Table 6.1, the estimated implementation time for AES and CRC differs by a factor of around 3.5. More complex applications may consume even more design time. Therefore, claims regarding design time estimation can only be made with the assumption of a specific use case and cannot be easily generalized on arbitrary applications.

[2]E.g., the interface integration itself does not depend on the number or complexity of the chosen CIs.

Table 6.2: Overview of the CI development process tasks to be estimated. Column *Framework Support* indicates which of these design tasks are handled or at least partly facilitated by FRANCIS-V. Column *Application-dependent* indicates if the design task is dependent on the given use case or not.

ID	Task	Framework Support	Application-dependent
1	Opcode selection	no	no
2	C Code adaption	no	no
3	Compiler setup	*yes*	no
4	CI implementation	no	*yes*
5	XIF learning	*yes*	no
6	XIF implementation	*yes*	no
7	RTL integration	*yes*	no
8	System verification	*partly*	*yes*
9	Synthesis	no	*yes*

Apart from this application dependency, further factors must be considered prior to inferring a particular value on the framework-based design time. First of all, to fully utilize FRANCIS-V, the provided RTL system has to be used. If specific system requirements demand an adaption of the system, e.g., a transition to a different RISC-V processor or further HDL components, the additional time needed for manually extending or adapting the proposed RTL system has to be added.

Another important point is the assumed experience of the responsible system designer. To get a more accurate estimation result, experts in their respective fields are elected as interview participants. However, the actual required design time may be higher for more inexperienced junior designers, e.g., to search for bugs in the HDL hardware.

Based on these reflections, we define influence factors on the actual framework-based design time, as outlined in Table 6.3. For the *use case* dependency, the time of task 4, t_{app}, is varied depending on whether the application is CRC, AES, or any other use case[3]. *Designer experience level* is considered by introducing an experience factor f_{exp} to all manual tasks in both the manual and framework-based analysis. For *system reusability* and *verification benefit*, we introduce factors f_{sys} and f_{ver} that reduce the design time of the respective tasks 7 and 8. For the designer training time t_{train}, we assume a fixed estimated time of three workdays for every further analysis ($t_{train} = 3$).

[3]The potential use-case dependency of the other tasks is neglected in this assumption.

Table 6.3: Summary of parameters influencing the design time reduction due to utilizing FRANCIS-V.

Parameter	Symbol	Impact
Use case	t_{app}	Application-dependent task 4 design time
Designer experience level	f_{exp}	Experience factor increasing all manual tasks
System reusability	f_{sys}	Factor for task 7 in FRANCIS-V-based design time
Verification benefit	f_{ver}	Factor for task 8 in FRANCIS-V-based design time
Designer training time	t_{train}	Learning offset for FRANCIS-V-based design time

Based on the typical process time estimations and the above-defined parameters, we can derive generalized formulas for the typical and framework-based design times, t_{man} and t_{fra}, and absolute and percentage time reduction, t_{red} and p_{red}:

$$t_{\text{man}} = f_{\text{exp}}\left(t_{t1-3} + t_{\text{app}} + t_{\text{t5-9}}\right) = f_{\text{exp}}\left(35.60 + t_{\text{app}}\right) \tag{6.1a}$$

$$\begin{aligned} t_{\text{fra}} &= f_{\text{exp}}\left[t_{t1,2} + t_{\text{app}} + t_{\text{t7}}(1 - f_{\text{sys}}) + t_{\text{t8}}(1 - f_{\text{ver}}) + t_{\text{t9}} + t_{\text{train}}\right] \\ &= f_{\text{exp}}\left[11.00 + t_{\text{app}} + 5.20(1 - f_{\text{sys}}) + 11.80(1 - f_{\text{ver}})\right] \end{aligned} \tag{6.1b}$$

$$\begin{aligned} t_{\text{red}} &= t_{\text{man}} - t_{\text{fra}} = f_{\text{exp}}\left(t_{\text{t3}} + t_{\text{t5,6}} - t_{\text{train}} + t_{\text{t7}} f_{\text{sys}} + t_{\text{t8}} f_{\text{ver}}\right) \\ &= f_{\text{exp}}\left(7.60 + 5.20 f_{\text{sys}} + 11.80 f_{\text{ver}}\right) \end{aligned} \tag{6.1c}$$

$$p_{\text{red}} = \frac{t_{\text{red}}}{t_{\text{man}}} = \frac{7.60 + 5.20 f_{\text{sys}} + 11.8 f_{\text{ver}}}{35.60 + t_{\text{app}}} \tag{6.1d}$$

6.1.2 Evaluation scenarios

For a specific quantization of the design time reduction and to illustrate the respective parameter's influences, we define three different exemplary scenarios, #1, #2, and #3, for a conservative, average, and optimistic set of expected parameters. The terms conservative and optimistic can be interpreted in the sense of achieving the lowest and highest realistic absolute and percentage design time reduction by choosing the influencing parameters accordingly.

As *use case*, we choose CRC as the least complex example, AES as the prime case, and an additional hypothetical use case with a doubled design effort compared to AES. A less complex use case yields a lower percentage difference p_{red}, thus, the cases are allocated according to their complexity to scenarios #3, #2, and #1, respectively. A lower *experience level* yields a higher experience factor, therefore, a higher absolute time reduction t_{red}. We assume a junior designer to have a 1.5 times higher design effort for the optimistic scenario #3 and a senior designer with a factor of 1 for the other scenarios. For *system*

reusability of the provided RTL system, we only differ between no reusability for #1, and full reusability otherwise. For *verification benefit*, we define a factor of around 25% or three workdays to be saved by FRANCIS-V, but assume no benefit in the conservative scenario #1. The training time is set to a fixed value of three workdays in all scenarios. The three configurations are summarized in Table 6.4.

The three scenarios are evaluated using the Equations (6.1a) to (6.1d). The results are illustrated in Table 6.5. Depending on the assumed scenario, they indicate a design time reduction of 7.60, 15.75, or 23.63 workdays or of 13.19%, 33.80%, or 40.59%. Therefore, despite a certain uncertainty of these quantifications, we can identify the general trend of a significant reduction in design time by using FRANCIS-V. For simpler applications and typical system requirements that the proposed system already fulfills to a large degree, the estimations indicate a significant reduction of design time of around 15 to 24 workdays or at least one third of design time saved. Even in the more conservative scenario, assuming a complex application and limited reuse potential of the RTL system due to specific processor and system requirements, there is still an estimated design time reduction of more than a workweek.

Table 6.4: Three scenarios with differing parameters to represent conservative, average, and optimistic assumptions for the deduction of design time reduction.

Parameter		**Scenarios**					
Name	**Symbol**	**#1: Conservative**		**#2: Average**		**#3: Optimistic**	
		Value	**Description**	**Value**	**Description**	**Value**	**Description**
Use case	t_{app} [d]	22	Fictional case	11	AES	3.2	CRC
Designer experience level	f_{exp}	1	Senior designer	1	Senior designer	1.5	Junior designer
System reusability	f_{sys}	0	No reusability	1	Full reusability	1	Full reusability
Verification benefit	f_{ver}	0	No benefit	0.25	Partial benefit	0.25	Partial benefit
Designer training time	t_{train} [d]	3	Average est.	3	Average est.	3	Average est.

Table 6.5: Design time comparison of three scenarios with conservative, average, and optimistic parameters and assumptions that influence the analysis. The design times of the typical and the framework-based design process are given in workdays. The absolute and percentage reduction of design time when utilizing the framework over a pure manual process is given in workdays and percent, respectively.

Design time		**Scenarios**		
Name	**Symbol**	**#1: Conservative**	**#2: Average**	**#3: Optimistic**
Typical design time	t_{man} [d]	57.60	46.60	58.20
FRANCIS-V-based design time	t_{fra} [d]	50.00	30.85	34.58
Design time reduction	t_{red} [d]	7.60	15.75	23.63
Percentage reduction	p_{red} [%]	13.19	33.80	40.59

Use case dependency

The impact on the absolute and percentage reduction (Equations (6.1a) to (6.1d)) differs for each parameter. The influence of the use case is illustrated in Figure 6.2. As evident from the Equations (6.1a) to (6.1d), the use case has no influence on the absolute design time reduction in our model since the application-dependent steps remain manual in both development process variants. However, the percentage reduction p_{red} lowers with a more complex use case since the influence of the other automated tasks of the framework-based process diminishes in comparison to a complex manual hardware implementation. This trend aligns with prior expectations, validating the choice of CRC and the complex fictional use case for the optimistic and conservative scenarios, respectively.

Due to the application dependency of the design time reduction, FRANCIS-V is more efficient when used in simpler applications where the CI implementation time is confined, or in cases where the CI hardware is already available, e.g., due to previous projects. For more complex use cases, the implementation task is a major task of the manual part of the development process. The results indicated that, apart from the system verification task, CI implementation is the task with the most accounted workdays for the moderately complex AES use case. A more complex use case would likely further elevate this influence, resulting in less percentage time reduction due to FRANCIS-V for complex applications.

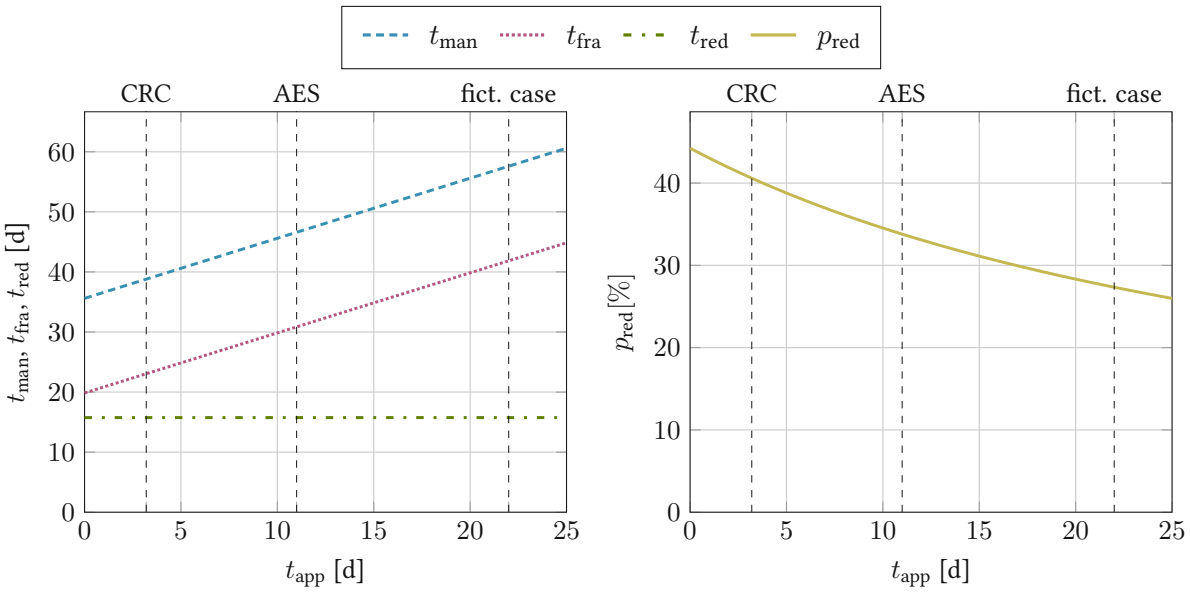


Figure 6.2: Diagram of typical and framework-based design time and absolute design time reduction (left) as well as percentage reduction (right) over the use-case-dependent design time required for task 4, t_{app}. The diagrams illustrate an influence on percentagebut not absolute design time.

To increase the design efficiency in these complex use cases, an automation of the CI implementation task would greatly benefit the overall design time reduction. One potential automation approach would be utilizing HLS to generate the CI hardware out of the source code of the given application. HLS, instead of manual hardware implementation, would directly link to the subsequent integration methodology proposed by FRANCIS-V and, subsequently, would imply a certain scalability for complex applications.

Designer experience

As shown in Figure 6.3, an increase of the introduced experience factor implicates an increase in the absolute but no change in the percentage reduction of design time since, in the derived equations, it is assumed that an inexperienced designer requires the same factor of additional time for all manual tasks.

This approach of experience level affecting all manual tasks must be seen as a rough approximation since the individual task times may fluctuate based on individual knowledge, especially for junior designers with limited experience. As an example, a junior engineer who is already familiar with the use case and HDL simulation but not with RISC-V, XIF, or GCC may tendentially save more design time because FRANCIS-V supports exactly the compilation and XIF-related tasks that may take this individual designer longer. On the other hand, a designer already familiar with RISC-V and XIF but with

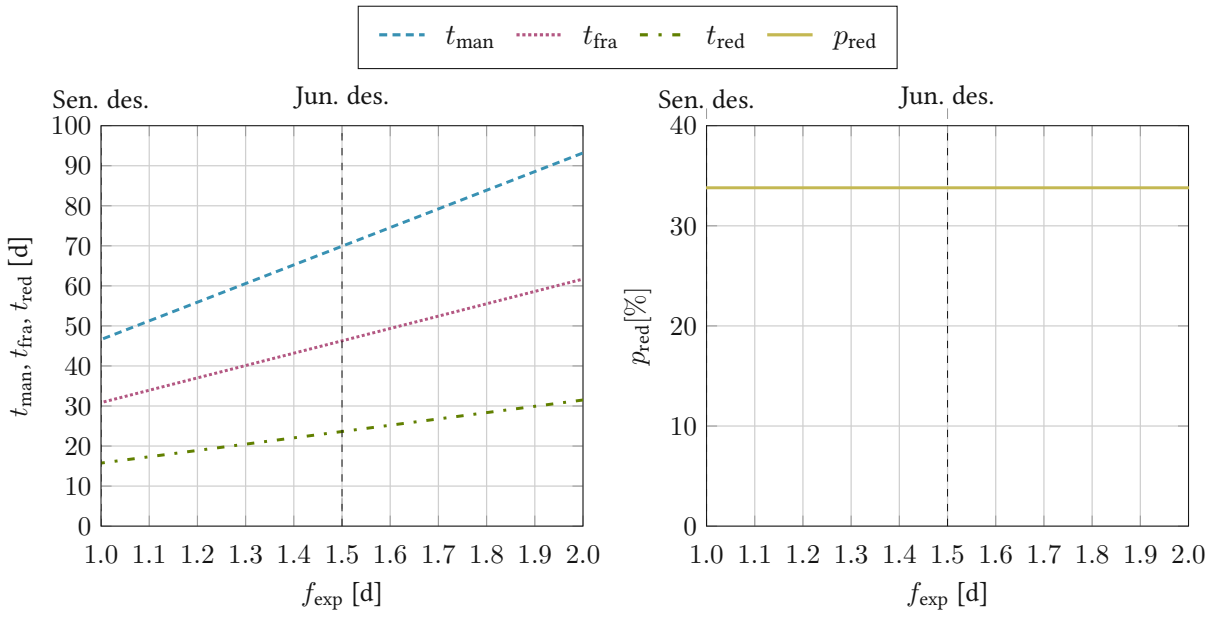


Figure 6.3: Diagram of typical and framework-based design time and absolute design time reduction (left) as well as percentage reduction (right) over the experience factor f_{exp}, introduced to model designer experience level. The diagrams illustrate no influence on percentage but on absolute reduction.

no prior experience on the application may find a solution for the XIF-related aspects in a reasonable time but struggle with the CI hardware implementation, therefore benefitting less from FRANCIS-V.

Further influences

The influence of the assumed verification benefit on both absolute and percentage reduction is illustrated in Figure 6.4. The impact of the system reusability factor is not illustrated in a figure. Its influence is similar to that of the verification factor but to a lesser extent since the system integration task is estimated to take less time than the verification task.

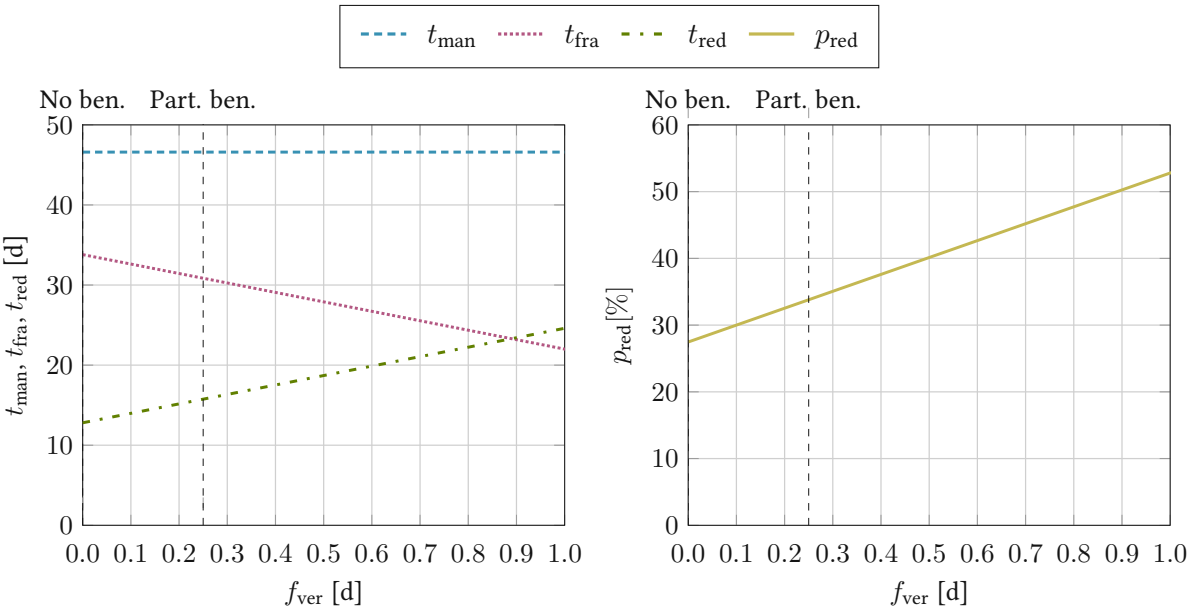


Figure 6.4: Diagram of typical and framework-based design time and absolute design time reduction (left) as well as percentage reduction (right) over f_{ver}, the factor that approximates how much the verification tools of the framework support the designer during the verification task. The diagrams illustrate an influence on percentage and absolute design time.

6.2 System metrics

The FRANCIS-V-generated system's runtime improvement and utilization overhead due to the XIF wrapper are evaluated by HDL simulation and synthesis, respectively.

6.2.1 Runtime improvements

One apparent expectation of using designated CI hardware for parts of the application is a reduction in the code's runtime. We evaluate this runtime by analyzing the instruction and cycle count of the proposed CI system generated by utilizing FRANCIS-V for the specific use case AES. The CI-based system is compared with two typical software-based algorithms *Reference* and *ttable* [23] provided by the RISC-V cryptography extensions standardization group (detailed in Section 2.5). For this comparison, the AES-specific benchmark script proposed in Section 5.5.1 simulates the three implementation variants of the algorithm on the proposed RTL system with the ISS Spike and the HDL simulator QuestaSim.

The instruction and cycle count of all three implementations is evaluated for the four segments of the test program presented in Section 5.4.2. The timing results are presented in Table 6.6 and illustrated in Figure 6.5. The table shows the instruction count, cycle count, and cycles per instruction of the HDL simulation with QuestaSim. As assumed, the results of the instruction set simulation with Spike regarding instruction count values are identical to the QuestaSim results and, therefore, are not illustrated explicitly. Cycle count cannot be evaluated with Spike, since Spike simulates on an instruction level without cycle-accurate timing information.

The hardware solution was expected to execute faster than the two software solutions. The results confirm these expectations. It took the system 8.37 and 3.15 times as many instructions and 8.02 and 2.53 times as many cycles to execute the *Reference* and *ttable* implementation over the CI-based implementation, respectively. In other words, there is a decrease of 88.05% and 68.26% in instruction count and 87.53% and 60.52% in cycle count when utilizing the CI solution over the software solutions *Reference* and *ttable*.

Table 6.6: Results of the runtime benchmarking program on two software (*Reference*, *ttable*) and one *CI-based* hardware implementation for the use case AES. The three segments of the table represent the simulated instruction count, cycle count, and resulting cycles per instruction, respectively. For the evaluation, the algorithm is split into the four distinct parts of *Encryption Key Scheduling (KSE)*, *Encryption (ENC)*, *Decryption Key Scheduling (KSD)*, and *Decryption (DEC)*.

	Program Parts	Implementation variants		
		Software		Hardware
		Reference	ttable	CI-based
Instr.	KSE	606	606	217
	ENC	3303	1023	237
	KSD	1129	1844	729
	DEC	6847	1001	237
	Sum	**11885**	**4474**	**1420**
Cycles	KSE	713	714	271
	ENC	3626	1043	248
	KSD	2782	1970	1108
	DEC	7903	1020	247
	Sum	**15024**	**4747**	**1874**
Cycles per Instruction	KSE	1.18	1.18	1.25
	ENC	1.10	1.02	1.05
	KSD	2.46	1.07	1.52
	DEC	1.15	1.02	1.04
	Sum	**1.26**	**1.06**	**1.32**

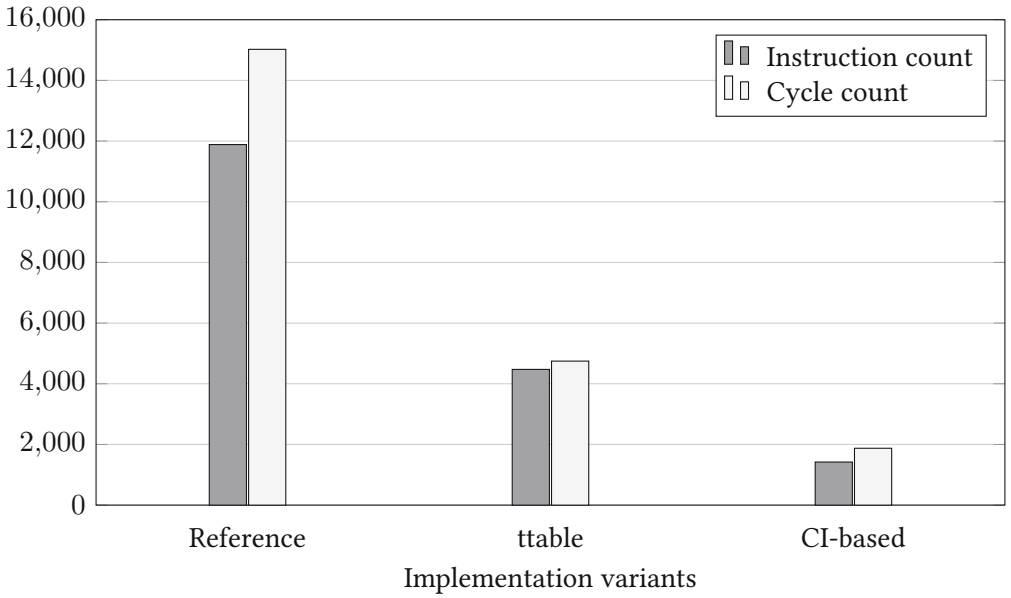


Figure 6.5: Comparison of the two software implementations (*Reference*, *ttable*) and *CI*-based hardware implementation regarding instruction and cycle count for the use case AES.

The program parts of the software solutions are within 1.02 to 1.18 cycles per instruction[4]. The values of the CI hardware solution, however, are higher on average, with 1.04 to 1.52 cycles per instruction. The raised cycle overhead arises from data dependency hazards caused by sequential instruction inside the key scheduling algorithm. The following code snippet illustrates the problem:

```
1 ...
2 aes32esi  T1, T1, T0, 0
3 aes32esi  T1, T1, T0, 1
4 aes32esi  T1, T1, T0, 2
5 aes32esi  T1, T1, T0, 3
6 ...
```

Listing 6.1: Snippet of the AES key scheduling assembly code illustrating the data dependency caused by the recurring occurance of T1.

The previous CI writes on the same register that is the input in the consecutive CI. Therefore, the consecutive instruction has to wait for one cycle until the result from the previous instruction is written back and the register is updated accordingly, resulting in two cycles per instruction in these cases. Fortunately, these constructs can be dissolved in the encryption and decryption algorithms, the respective code parts exhibit a low cycle overhead of 1.04 to 1.05 cycles per instruction. In a practical application, key scheduling is done once, followed by several encryption iterations instead of only one iteration, as done in the test program. Hence, the overall influence of this cycle overhead decreases in practical applications.

A comparison of the software solutions further shows an improvement of the optimized *ttable* implementation over the conventional *Reference* implementation. The similarity of the encryption key scheduling (KSE) of both software solutions arises because the identical code is used for this part of the algorithm. A further observation on the cycles per instruction shows that the *ttable* implementation is relatively efficient in terms of cycles per instruction (1.02 to 1.07) in contrast to the *Reference* implementation.

[4]An exception to this range is the decryption key scheduling KSD with 2.46 cycles per instruction. For the following considerations, it is assumed to be an outlier, attributable to the inefficient decryption key scheduling of the *Reference* implementation in contrast to the *ttable* implementation.

6.2.2 Hardware overhead

As proposed in Section 5.5.2, the resulting system for the use case AES has been synthesized on a selected Xilinx FPGA to evaluate the resource utilization. For the overhead analysis, two configurations of the system are synthesized. The first configuration represents the pure software approach without CIs, consisting of the CV32E40X main core and instruction and data memory. The second, CI-based configuration is extended to also include the XIF wrapper with the integrated AES module.

The utilization results of the CI-based approach are divided into two parts, one resembling the main core and memory components and a second one arising due to the XIF wrapper and AES module. The resulting utilization values are illustrated in Table 6.7, complemented by the total utilization and percentage overhead of the CI hardware compared to the total utilization. The software-only configuration is not explicitly stated in the table since its utilization values closely resemble those of the main core and memory part of the second configuration, differing only by an additional LUT but one FF less.

The results show that the XIF wrapper with integrated AES module has only a marginal utilization overhead, the major part of the system utilization arises from the main core and memory. However, this data only provides insight into the specific use case AES, and not for generic applications. A more complex use case will likely implicate a more complex XIF wrapper, particularly due to the integrated CI module, therefore resulting in more resource overhead.

Table 6.7: Lookup Table (LUT), Flip Flop (FF), and Block RAM (BRAM) utilization data for the synthesized system generated by FRANCIS-V for the specific use case AES. The values are stated for the two parts of the system consisting of *main core and memory* and *XIF wrapper with CI AES module*, respectively, as well as for the overall system (*sum*). Also stated is the overhead of the XIF wrapper in relation to the total utilization values.

System components	**Utilization**		
	LUT	**FF**	**BRAM**
Core, Memory	3731	2333	16
XIF Wrapper, AES Unit	32	66	0
Sum	3763	2399	16
CI Overhead [%]	0.85	2.75	0

6.3 XIF timing properties

The CI interface XIF, in contrast to systems using in-pipeline integration, potentially holds a timing overhead for communication between the main core and external coprocessor or wrapper. To assess this overhead and to gain further insight on the behavior, features, and restrictions of XIF, we perform a simulation of the internal signals of the main core, coprocessor, and XIF, and analyze the results.

To get a typical exemplary transfer between the main core and the wrapper over XIF, the use case CRC is simulated with QuestaSim. The resulting internal signals of this simulation are shown in Figure 6.6[5]. The 4-stage core consists of the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Writeback (WB) stages. In the first cycle, the core's IF stage fetches the instruction (opcode *0x02F8086B*). The input data (*0x6162*) was already fetched by the core due to previous instructions (not illustrated in the figure). In the second cycle, the decoded instruction is not recognized by the subsequent ID stage and, therefore, issued to the wrapper over XIF together with the input operand and some additional data. The wrapper decodes and accepts the instruction in the same cycle. Afterwards, the instruction is signaled to be committed by the main core in cycle three, thus no longer speculative. In the same cycle, the wrapper executes the instruction and generates the output (*0x74FF*), indicated by the result valid signal. The main core accepts the result in cycle four and writes the result back to its internal register file.

For a process-internal instruction, the core's four stages implicate a latency of four cycles due to IF, ID, EX, and WB for instructions managed by the main core, respectively. A CI is fetched (IF), issued (ID), and committed (EX) by the main core, and in the meantime decoded (ID), executed (EX), and written back (WB) by the wrapper and subsequently by the main core. Therefore, a CI in these cases is also performed in four cycles. In addition, multiple CIs enter the main core's pipeline one cycle at a time. Thus, the same throughput of one instruction per cycle is reached, which is also achievable by processor-internal instructions. Therefore, the wrapper and CV32E40X combination inside FRANCIS-V utilizing XIF does not come with additional latency or throughput overhead under these ideal assumptions.

However, it has to be noted that these considerations are based on ideal assumptions of a hazard-free program execution. In practice, as seen in Section 6.2.1, in most cases a throughput of exactly one instruction per cycle is not reached due to pipeline hazards (e.g., load data or jump hazards [19]). Using XIF and CIs may still influence the occurrence of these hazards and thus indirectly influence latency and throughput,e.g., due to the inefficient compiler scheduling described in Section 6.2.1. In addition, the framework does not use all of XIF's features and is limited to single-cycle instructions. In the case

[5]The figure represents a simplified version of the core's and XIF's internal signals to illustrate the general execution process. Details such as additional signals of the Issue interface are omitted.

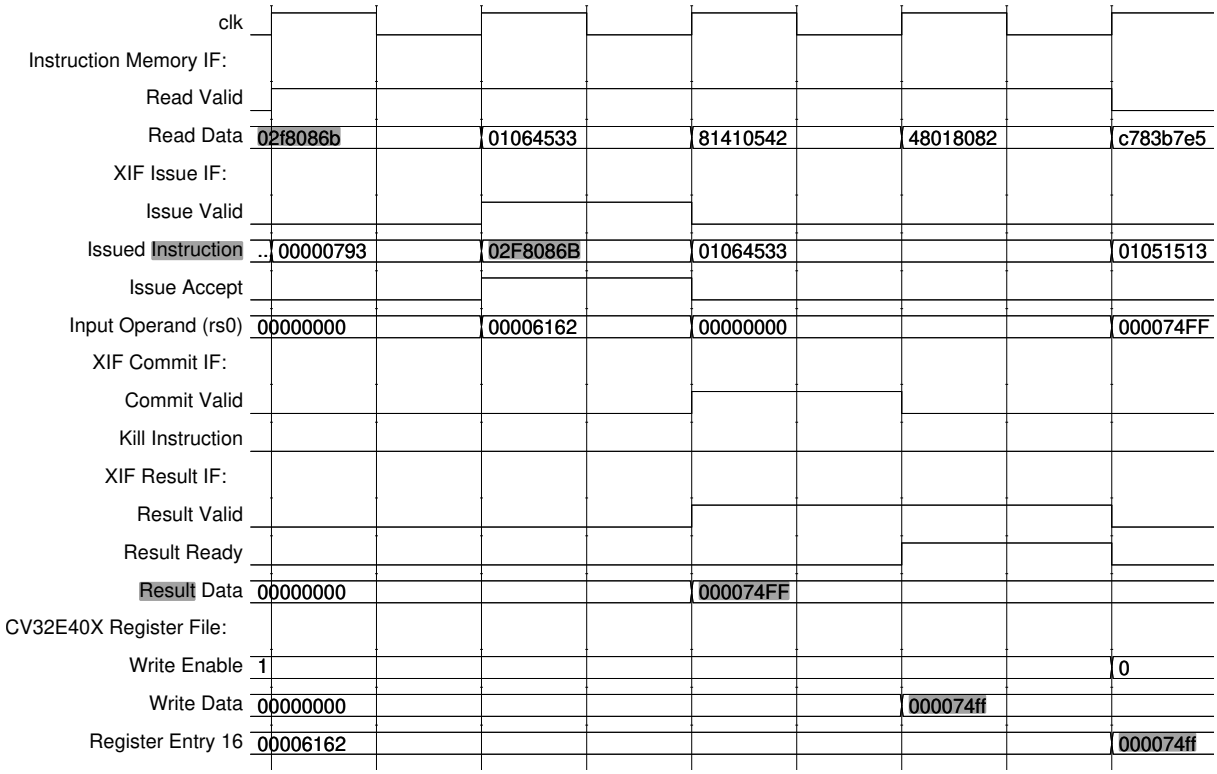


Figure 6.6: Simulation of a typical XIF transfer for the use case CRC. In the first cycle, the main core fetches the instruction *0x2F8086B*. It issues the instruction to the wrapper in the second cycle together with the prefetched input data. Yet in this cycle, the wrapper decodes and accepts the issued instruction. In cycle three, the main core commits the instruction while the coprocessor executes it and generates the output *0x74FF* which is collected and stored by the main core in the subsequent fourth cycle.

of multi-cycle instructions or more advanced features of XIF, such as load/store operations, additional stalls due to multiple execute-cycles and memory transactions would occur. Hence, the assumption of one cycle per instruction may not hold anymore in these cases.

Chapter 7

Discussion and future work

In this chapter, we discuss the main results, limitations, and findings of this thesis. Based on these considerations, we suggest future work regarding research topics and potential extensions of FRANCIS-V.

7.1 CI development process analysis

The first research question of this thesis reads as follows:

Which are the most challenging process steps in a typical Custom Instruction development process regarding time and effort?

To identify these challenges, we define the process steps in Section 4.1 based on various assumptions and identify potential automation approaches. Among these steps, based on the results of the conducted expert interview in Section 6.1, the most time-consuming challenges are identified to be the combined steps of CI integration with around 14 workdays, verification with around 12 workdays, and CI implementation with 11 workdays for the AES use case. Among these, FRANCIS-V's proposed integration methodology contributes to the integration problem. Additionally, the framework offers verification tools. The estimations of only 25% savings due to these tools, however, indicate that they are not expected to fully automate verification. Formal verification methods are identified as an alternative promising approach that also might raise the verification quality but are out of the scope of this thesis. Moreover, CI logic generation is not covered by the framework. However, FRANCIS-V is compatible with prospective tools that generated HLS-based implementations.

CI identification is not included in the expert estimation and thus does not appear in the time estimation results. Rather, the CIs are assumed to be already predetermined. However, depending on the complexity of the application, it is reasonable to assume that this development task holds a particular development time and effort. The identified approaches of automated profiling and performance estimation tools for various CI configurations likely simplify the selection of the most appropriate CIs for the designer. In addition, such approaches would provide the designer with metrics of the examined configurations, thus allowing for a more evidence-based CI selection in contrast to the conventional manual selection approach that solely depends on the designer's experience.

One potential point of criticism is the focus of the analysis on the integration-based approaches, in particular with XIF as a suggested solution. This emphasis can be justified with the focus of this thesis being on integration methodologies. However, a grouping or abstraction of the integration segments may facilitate a more generalized analysis. An inclusion of more details, on the other hand, such as splitting CI identification into substeps, adding synthesis and implementation, or better visualizing iterations within the process may have led to a more accurate representation of the development process' details. However, these details may have made it more cumbersome to grasp the essence of each step.

7.2 Framework limitations and applications

The framework FRANCIS-V approaches the second research question:

How can the manual effort of integrating Custom Instructions with RISC-V cores be reduced using the CORE-V eXtension Interface?

Besides several features proposed in the previous chapters, FRANCIS-V entails certain limitations that can be categorized as follows:

Supported instructions: FRANCIS-V supports R-type single-cycle instructions to keep the underlying CI hardware combinational. It does not support multi-cycle instructions nor immediate, load/store, or branch and jump instructions. Those restrictions ensure a constrained wrapper complexity. Branch and jump instructions, in addition, are not supported by the framework's underlying CI interface XIF.

Assumed inputs: It is assumed that the designer provides preselected CIs and an appropriate hardware implementation and source code to the framework. FRANCIS-V features no hardware generation or CI identification methods. Further, as detailed in Section 4.1, the framework only supports compilation of adapted source code where the designer already placed the CIs in form of their opcodes at designated

code locations. Thus, the compiler has no semantic information about the instructions and only places the CIs at these locations. It cannot autonomously decide on further, potentially suitable code locations. The fundamental code compilation is supported by FRANCIS-V to aid the designer, more complex or autonomous compiler configurations, however, are not within the scope of this thesis.

Verification: Verification is supported by FRANCIS-V by providing a testbench and debug capabilities of the source code within the HDL simulation. However, additional features such as automated test case generation or formal verification methods are not supported. Rather, the designer has to develop the source code according to their testing requirements.

Simulation: In an otherwise open-source framework, QuestaSim is the only commercial tool. In Section 5.3, we justify this simulator choice based on the enhanced and intuitive features of QuestaSim and argue that a change to open-source solutions is a potential future step.

Synthesis: The framework constrains itself on simulation, and does not offer tools for synthesis and implementation on FPGAs or ASICs. However, the generated system designs are fully synthesizeable, apart from two architecture-dependant clock gate and RAM modules.

Potential solutions and extensions to these limitations are discussed in Section 7.6.

As presented in Section 4.2, FRANCIS-V supports a broad range of potential applications despite the limitation to R-type single-cycle instructions. One prime example is security applications. The possibility of realizing AES or CRC is already featured in the thesis, further potential cryptography algorithms, such as SHA (Secure Hash Algorithm), are imaginable. Use cases proposed by work priorly featured in this thesis for PQC, NFC or for vector instructions (similar to those of Vicuna or Spatz) represent further possible applications.

Moreover, the various applications proposed by Cui et al. [59] are all interesting candidates for FRANCIS-V. The authors, besides the already mentioned security and PQC use cases, also propose candidates such as floating-point computation, signal processing, or Artificial Intelligence (AI) applications. All use cases are only restricted by the single-cycle limitation requiring a combinational implementation. To avoid this limitation, however, complex instructions can be split into multiple CIs.

7.3 Design time reduction

The third research question specifies the main purpose of FRANCIS-V and its integration methodology by defining the main metric of interest:

How much design time can a designer save with the developed framework depending on their experience level and use case?

The methodology (Section 4.3) and results (Section 6.1) for evaluating this time saving raises three major points of discussion. First, the granularity of the partitioned tasks to be estimated is rather coarse. We justify this partitioning by the time limitation of the conducted interview and the deliberate restriction of too detailed information, e.g., to prevent an accidental solution proposal that could possibly limit the participant's imagination. Secondly, the selected group of participants is relatively small and, regarding company and experience level diversity, homogeneous. Nonetheless, the group consists of experts in their diverse respective fields to provide an assessment as accurate as possible. Third, expert estimations inherently involve particular accuracy limitations and biases.

Despite a particular uncertainty due to these stated limitations, the results of the estimation provide a trend and show that FRANCIS-V does significantly reduce the required design time by supporting the designer in certain tasks of the otherwise manual design process. The quantified results suggest around 7 to 24 workdays or 13% to 41% of design time reduction when utilizing FRANCIS-V, depending on the assumed scenario, with the realistic scenario yielding around 16 days or 34%.

The distinct estimations of the various tasks of the process facilitate an analysis of the influencing factors of these values, as discussed in Section 6.1. The dependency on the specific use case and on the designer experience are of particular interest. To recall previous considerations, the percentage time reduction increases with simpler use cases and, according to the simplified assumptions, does not depend on the designer experience. The use case dependency provides an argument for automated, HLS-based CI hardware generation to raise the potential time savings for complex applications. Without this addition, FRANCIS-V is most efficient when used for applications with limited complexity and prototypical use cases. Nevertheless, more complex systems are supported, with the generated system at least providing a starting point for subsequent optimizations to tailor more specific system requirements.

The designer experience not influencing the relative reduction value likely follows the simplified modeling with the same experience factor on all influenced tasks. The real influence of designer experience is most certainly way more complicated and dependent on the knowledge of the particular designer, e.g., regarding XIF or RISC-V.

A further aspect is the reduction of required knowledge to design CI-based systems when utilizing FRANCIS-V. After becoming familiar with the framework, a designer requires less knowledge about its underlying concepts, in particular about XIF, since FRANCIS-V fully manages XIF-related integration and implementation tasks. Not learning XIF, consequently, reduces the training period for both inexperienced and experienced designers. Inexperienced designers may additionally profit from the provided system solution since they otherwise may produce a less efficient design than the framework does. Thus, the system's quality regarding its metrics may increase. Furthermore, FRANCIS-V is a promising solution when a designer is in need of a proof-of-concept system design or for quickly exploring different C source code and CI logic configurations. On the other hand, while still saving time in more complex application designs, the relative design time efficiency decreases in these cases. Experienced designers familiar with XIF, in addition, might even benefit from implementing the system manually and optimizing it to their specific needs, in particular when they are faced with tight system requirements.

7.4 System metrics comparison

Besides reducing the design time with FRANCIS-V, it is of natural interest that the resulting CI system exhibits a baseline of comparable runtime and utilization metrics to be competitive in contrast to similar approaches. The respective evaluation results are presented in Section 6.2.1 and Section 6.2.2. To summarize these results, a reduction in cycle count of up to around 88% is achieved with utilization overhead of only 0.85% and 2.75% for LUTs and FFs for the AES use case.

Since these metrics strongly depend on the application and underlying main core, a comparison to other papers can only give rough trends, but is still of interest for a coarse classification of the system. Therefore, we compare FRANCIS-V's AES-based metrics with three selected frameworks OpenASIP, SCAIE-V, and TIGRA. Hepola et al., the authors of OpenASIP, also utilize AES as a test case and report a runtime reduction of 40% and an area overhead of 1.5% [6]. The SCAIE-V authors Damian et al. evaluate an AES SBox instruction on the 4-stage core VexRiscv. Out of 1111 LUTs and 671 FFs[1], 1.26% and 0.45%

[1]In contrast, the 4-stage processor CV32E40X and XIF wrapper within this thesis utilizes 3763 LUTs and 2399 FFs. Therefore, albeit both cores realize four stages, the utilization overhead only holds limited comparability. For larger cores, the their influence on the utilization is higher, thus, the percentage overhead is likely to go down.

are attributed to the CIs, respectively [28]. The authors of TIGRA, Green et al., report a cycle count of 51 cycles for the AES encryption, which amounts to a reduction of 91.5% compared to their stated AES-128 software reference completing at 600 cycles[2] [33].

With 0.85% LUT and 2.75% FF overhead, FRANCIS-V's system achieves a similar magnitudes than SCAIE-V does, with a lower LUT but higher FF. OpenASIP only reports area instead of utilization, so a direct comparison has to be interpreted with caution. However, in terms of magnitude, the overhead is also comparable. Furthermore, FRANCIS-V outperforms OpenASIP with a cycle time reduction of 60.52% to 87.53%, depending on the compared software algorithm. Regarding AES encryption only, FRANCIS-V achieves a runtime reduction of 76.22% or 93.16%, which is also comparable to TIGRA's reported number.

In addition to these metrics, additional parameters such as change in critical path and power consumption would be of interest. Furthermore, more evaluations regarding the second use case CRC and additional use cases would have complemented the thesis. However, these ambitions are out of scope for a single master thesis. Still, the evaluated metrics and chosen use cases are comparable to similar work on this topic, especially regarding the popular test case AES. The comparisons indicate the competitiveness of FRANCIS-V's generated designs.

7.5 XIF benefits and limitations

XIF is chosen as the CI interface for FRANCIS-V and, therefore, strongly influences the capabilities and limitations of the framework. One advantage of XIF is its scalable specification. The six subinterfaces support complex structures and CI designs, and the structure and purpose of XIF's signals appear elaborate. In addition, since not all of the interfaces are mandatory, XIF also supports designs with reduced features and complexity, e.g., as FRANCIS-V only supports R-type instructions. Still, as a disadvantage of XIF, the induction period to grasp and apply XIF was found to be rather high during this thesis.

Furthermore, as proposed in Section 6.3, a benefit of XIF is the absence of additional latency due to XIF when utilized with the CV32E40X, assuming a hazard-free instruction sequence. Another advantage of XIF is the support of a large subset of instruction types. However, control transfer instructions like jumps and branches are not supported, albeit these features seem to have almost no interface and framework support outside of SCAIE-V and Codasip Studio [28].

[2]Since the authors do not give an exact cycle count of a reference implementation, we choose 600 cycles as reference.

A potential disadvantage of XIF that also impacts the applicability of FRANCIS-V in its current form is its present lack of usage within the RISC-V ecosystem. While there are several coprocessors and accelerators with XIF support (as listed in Section 3.3), as of writing this thesis, only the CV32E40X core from CORE-V themselves supports XIF. The initial expectation for XIF at the time of starting this project was for it to become common within the RISC-V ecosystem in the near future. This expectation, among others, emerged from the XIF specification being formulated for generic processors and not a specific core, and from articles about two different cores in development with XIF on their intended feature's list [60], [61]. Though, one year later at the time of writing this paragraph, both the IBEX core and the AIRISC seem not to support XIF yet [62], [63].

However, we anticipate that XIF will increase in popularity as time progresses. The present lack of utilization does not undermine the generic design of XIF, in particular when considering that the interface is still in its pre-release phase and not finally released yet. Due to CORE-V eXtension Interfaces (XIFs) promising idea and elaborate structure, it is more than probable that XIF will become more common over time once it had time to settle. After its major release, we envision XIF to experience a rise of support in RISC-V cores, coprocessors, and also design tools supporting the interface specification. The various proprietary solutions and advances in the field of CI integration certainly show the need and potential benefits of a RISC-V standard interface.

Even a potential lack of XIF-compatible RISC-V cores would not reduce the value of this thesis. In this case, FRANCIS-V still facilitates CI design for the feature-rich and promising CV32E40X core. In addition, the framework is applicable for potential adaptions to feature further CI interfaces. In this scenario, the insights on CI integration proposed in this thesis still hold.

7.6 Future work

Although this thesis proposes a fully developed CI integration framework and a comprehensive discussion about its contributions, the presented limitations leave room for future improvements of FRANCIS-V and for additional contributions to the current state of the art. FRANCIS-V is developed as one component in a large prospective setting of CI development tools for CI identification, generation, integration, and verification. We structure suggestions on future work in three aspects regarding automation of the overall *development process*, additions to *FRANCIS-V*, and potential advancements for *RISC-V*.

Development process automation

A major insight of this thesis is the influence of the CI hardware implementation on the overall design time, which is also evident from the estimation results. To overcome FRANCIS-V's need for existing CI logic, one promising approach would be to implement an HLS-based tool for automatic hardware generation based on the C application, resulting in CI logic that is applicable to the input of FRANCIS-V. In addition, CI identification methods for evaluating and selecting the optimized CIs for the specific application based on certain requirements have the potential to significantly improve the resulting system's quality. Both these concepts would be directly compatible with FRANCIS-V.

Framework improvements

An extension of additional instruction types supported by FRANCIS-V mainly has implications on the design of its XIF wrapper. Supporting immediate instructions for computing would be the simplest addition, with minor changes to the CI module instantiation and decoder. Load and store instructions would additionally require extending the wrapper with the Memory and Memory Result subinterfaces of XIF and result in a more complex control logic, instruction FIFO, and a local register file. Adding these instructions would allow for simpler and more compact assembler code [64]. Custom jump and branch instructions would require a different CI interface since XIF does not support those kinds of instructions. Multi-cycle instructions are a more complex addition to the wrapper that, among others, would require a considerable extension of the control logic.

A rather straightforward improvement to the software functionality group of the framework would include an automated generation of unused and legal opcodes for each CI to be implemented, which would eliminate the minor and currently manual opcode selection task. An even more significant extension would be an automated integration of the CIs and their behavior into the compiler itself, as mentioned in Section 4.2, to remove the need of manually exchanging the respective CI code parts with their assembler representations.

Another open discussion point is the support of simulation and compilation tools. FRANCIS-V would benefit from further verification features such as additional simulator support like the ISS Spike or an open-source alternative to QuestaSim. Further verification support would also be conceivable, e.g., automated source code generation for different test cases or formal verification techniques.

Advancing the RISC-V ecosystem

One key finding of this thesis is that the RISC-V ecosystem would benefit from a standard CI interface. Therefore, with this thesis, we aim to encourage future work on XIF and similar CI interfaces or work on the extension of RISC-V processors and tools to support XIF. RISC-V would benefit from more attention to standard CI solutions in the long run.

Similarly, an advancement of open-source tools, in particular open-source simulators, would benefit a growing RISC-V community. During this thesis, we experienced that, although powerful open-source alternatives exist, their commercial counterparts understandably often outperform them regarding usability and intuitiveness.

Chapter 8

Conclusion

Based on the identified lack of CI design tools in the RISC-V ecosystem, this thesis proposes the framework FRANCIS-V for facilitating the CI integration process using XIF. The defined research questions aim at an analysis of the CI development process to identify its primary automation challenges and potential solutions, with a focus on integration with interfacing of external CI-based coprocessors. They further define the metric of interest for evaluating FRANCIS-V as design time reduction due to utilizing the framework within the CI development process.

We answer the first question analyzing the typical development process. We identified the most demanding process steps as CI integration, logic generation, and verification. The influence of integration justifies the thesis' focus, while logic generation with HLS or similar methods is recognized as a promising future approach. We further identified CI identification techniques as a potential addition for a more optimized system design.

To approach the manual CI integration effort, we propose an integration methodology based on XIF, realized within FRANCIS-V. A predefined RISC-V system as well as compilation, simulation, and verification features complement this methodology. The framework's capabilities are demonstrated with two use cases, AES and CRC. The system designs generated by the framework achieve a substantial cycle count decrease of up to 87.53% and a low resource overhead of 0.85% and 2.75% for LUTs and FFs, respectively, rendering FRANCIS-V competitive in contrast to related work. A secondary finding of the timing analysis is that XIF entails no latency overhead when assuming a hazard-free program execution, positioning XIF as promising in tightly integrated embedded applications.

For inferring the design time saved by FRANCIS-V, we performed expert estimation with a selected group of participants and analyzed the results with varying assumptions and scenarios. The typical scenario yields 16 workdays or 34% of design time reduced due to FRANCIS-V. After analyzing the detailed results, we showed that, despite being more efficient for simpler applications regarding design time reduction, the framework is applicable for both simple and complex use cases. Both inexperienced and experienced designers benefit from FRANCIS-V by not having to learn about XIF and by reusing the proposed system and tools. Inexperienced designers, furthermore, may save additional time when using FRANCIS-V for design steps where they lack experience.

FRANCIS-V is limited to single-cycle instructions. It also requires the designer to provide the CIs, application source code, and CI logic. Further limitations include only commercial simulators and no synthesis tools being native to FRANCIS-V. Still, the generated systems are fully synthesizeable. The conducted expert interview has a particular uncertainty due to a small participant group but still indicates trends that allow for an analysis and coarse quantification of the required design time. Another point of discussion is that XIF currently only supports a single processor. However, after a prospective full release of XIF, we envision that more RISC-V processors and tools will facilitate the interface over time and assess that XIF is a promising candidate for a future CI standard.

The limitations of this thesis highlight the potential for further work. In particular, extensions further automating the overall development process, such as HLS logic generation or CI identification techniques, represent promising approaches directly compatible with FRANCIS-V's integration methodology. Another potential future approach is extending the framework with further instruction and tool support.

We conclude the thesis's main contribution to be FRANCIS-V and its CI interfacing methodology, constituting significant improvements towards an automated CI development process. We further contribute an analysis of the typical process and highlight potential enhancements and future research on this topic. Finally, we identify the current lack of a standard RISC-V CI interface and emphasize the importance of efforts concerning this matter, especially regarding the promising candidate XIF.

Bibliography

[1] M. Imai, Y. Takeuchi, K. Sakanushi, and N. Ishiura, "Advantage and possibility of application-domain specific instruction-set processor (ASIP)," *IPSJ Transactions on System LSI Design Methodology*, vol. 3, pp. 161–178, Aug. 2010. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjtsldm/3/0/3_0_161/_article. [Accessed: Aug. 2023]

[2] K. Keutzer, S. Malik, and A. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 84–90. [Online]. Available: https://ieeexplore.ieee.org/document/1106752. [Accessed: Aug. 2023]

[3] M. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *Fourteenth International Conference on VLSI Design*, 2001, pp. 76–81. [Online]. Available: https://ieeexplore.ieee.org/document/902643. [Accessed: Aug. 2023]

[4] A. Waterman and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, document version 20191213," RISC-V International, Dec. 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf. [Accessed: Aug. 2023]

[5] "Extending RISC-V ISA with a custom instruction set extension," Codasip, 2019. [Online]. Available: https://www.design-reuse.com/articles/46237/extending-risc-v-isa-with-a-custom-instruction-set-extension.html. [Accessed: Feb. 2023]

[6] K. Hepola, J. Multanen, and P. Jaaskelainen, "OpenASIP 2.0: Co-design toolset for RISC-V application-specific instruction-set processors," in *IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors*, 2022, pp. 161–165. [Online]. Available: https://ieeexplore.ieee.org/document/9912050. [Accessed: Aug. 2023]

[7] "CORE-V eXtension Interface," OpenHW Group, 2021. [Online]. Available: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif. [Accessed: Jan. 2023]

[8] "RISC-V homepage," RISC-V International. [Online]. Available: https://riscv.org/. [Accessed: Jan. 2023]

[9] "Together for RISc-V Technology and ApplicatioNs," CORDIS EU research results, NXP Semiconductors Germany GmbH, 2022. [Online]. Available: https://cordis.europa.eu/project/id/101095947. [Accessed: Jul. 2023]

[10] "RISC-V specifications," RISC-V International. [Online]. Available: https://riscv.org/technical/specifications/. [Accessed: Jan. 2023]

[11] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V instruction set manual, volume II: Privileged architecture, document version 20211203," Dec. 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf. [Accessed: Aug. 2023]

[12] "RISC-V cores and SoC overview," RISC-V International, 2019, GitHub repository. [Online]. Available: https://github.com/riscvarchive/riscv-cores-list. [Accessed: Jan. 2023]

[13] J. Scheel, "RISC-V recently ratified extensions," RISC-V International. [Online]. Available: https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions. [Accessed: Jul. 2023]

[14] A. Zeh, A. Glew, B. Spinney, B. Marshall, D. Page, D. Atkins, K. Dockser, M.-J. O. Saarinen, N. Menhorn, L. P. Deutsch, R. Newell, and C. Wolf, "RISC-V cryptography extensions volume I: Scalar & entropy source instructions, document version v1.0.1," RISC-V International, Feb. 2022. [Online]. Available: https://github.com/riscv/riscv-crypto/releases/download/v1.0.1-scalar/riscv-crypto-spec-scalar-v1.0.1.pdf. [Accessed: Aug. 2023]

[15] "Functional verification project for the CORE-V family of RISC-V cores." OpenHW Group, 2019, GitHub repository. [Online]. Available: https://github.com/openhwgroup/core-v-verif. [Accessed: Jan. 2023]

[16] "CORE-V family of RISC-V cores," OpenHW Group, 2020, GitHub repository. [Online]. Available: https://github.com/openhwgroup/core-v-cores. [Accessed: Jan. 2023]

[17] "OpenHW Group CV32E40P User Manual," OpenHW Group, 2020. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual. [Accessed: Jan. 2023]

[18] "CORE-V verification environment," OpenHW Group, 2021. [Online]. Available: https://docs.openhwgroup.org/projects/core-v-verif. [Accessed: Jan. 2023]

[19] "CV32E40X user manual," OpenHW Group, 2020. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40x-user-manual. [Accessed: Jan. 2023]

[20] "OBI: Open Bus Interface specification," Silicon Labs, 2020, GitHub repository. [Online]. Available: https://github.com/openhwgroup/obi/. [Accessed: Jan. 2023]

[21] "CORE-V eXtension Interface," OpenHW Group, 2021, GitHub repository. [Online]. Available: https://github.com/openhwgroup/core-v-xif. [Accessed: Jan. 2023]

[22] J. Daemen and V. Rijmen, *The design of Rijndael: The Advanced Encryption Standard (AES)*, 2nd ed. Heidelberg, Germany: Springer, 2020. [Online]. Available: https://link.springer.com/book/10.1007/978-3-662-60769-5. [Accessed: Aug. 2023]

[23] "RISC-V cryptography extension," RISC-V International, 2019, GitHub repository. [Online]. Available: https://github.com/riscv/riscv-crypto. [Accessed: Jan. 2023]

[24] T. Lin, "Adding custom instructions to RISC-V to boost performance while reducing power and code density," Andes Technology, 2019. [Online]. Available: https://www.allaboutcircuits.com/industry-articles/add-instructions-risc-v-boost-performance-reduce-power-and-code-density/. [Accessed: Jan. 2023]

[25] B. Hu, Y. Chen, and X. Zeng, "An agile instruction set extension method based on the RISC-V processor," in *IEEE 4th International Conference on Electronics Technology*, 2021, pp. 342–346. [Online]. Available: https://ieeexplore.ieee.org/document/9450911. [Accessed: Aug. 2023]

[26] A. Waage, P. G. Kjeldsberg, and H. Fegran, "Secure implementation of a RISC-V AES accelerator," M.S. thesis, Department of Electronic Systems, Norwegian University of Science and Technology, Trondheim, Norway, 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3023096. [Accessed: Aug. 2023]

[27] M. Poncino and D. Pala, "Design and programming of a coprocessor for a RISC-V architecture," M.S. thesis, Collegio di Ingegneria Informatica, del Cinema e Meccatronica, Politecnico di Torino, Turin, Italy, 2017. [Online]. Available: https://webthesis.biblio.polito.it/6589. [Accessed: Aug. 2023]

[28] M. Damian, J. Oppermann, C. Spang, and A. Koch, "SCAIE-V: an open-source SCAlable interface for ISA extensions for RISC-V processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 169–174. [Online]. Available: https://dl.acm.org/doi/10.1145/3489517.3530432. [Accessed: Aug. 2023]

[29] "PicoRV32 - A size-optimized RISC-V CPU: Pico Co-Processor Interface (PCPI)," YosysHQ, 2015. [Online]. Available: https://github.com/YosysHQ/picorv32#pico-co-processor-interface-pcpi. [Accessed: Feb. 2023]

[30] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html. [Accessed: Aug. 2023]

[31] J. Nazarenus and D. Swierzy, "CCOPI: Implementing a custom coprocessor interface for VexRiscv," RheinMain University of Applied Sciences, 2018. [Online]. Available: https://github.com/jens-na/VexRiscv-CCOPI/blob/master/paper/ccopi_paper.pdf. [Accessed: Jan. 2023]

[32] "Hummingbirdv2 E203 core: Nuclei instruction co-unit extension (NICE)," Nuclei System Technology, 2020. [Online]. Available: https://doc.nucleisys.com/hbirdv2/core/core.html#nice. [Accessed: Feb. 2023]

[33] B. Green, D. Todd, J. C. Calhoun, and M. C. Smith, "TIGRA: A Tightly Integrated Generic RISC-V Accelerator interface," in *IEEE International Conference on Cluster Computing*, 2021, pp. 779–782. [Online]. Available: https://ieeexplore.ieee.org/document/9556053. [Accessed: Aug. 2023]

[34] M. Imfeld, "Floating-point unit subystem," PULP Platform, 2022, GitHub repository. [Online]. Available: https://github.com/pulp-platform/fpu_ss. [Accessed: Feb. 2023]

[35] F. Babbaro, G. Masera, and S. Brennsteiner, "A RISC-V based accelerator for NFC signal processing," M.S. thesis, Politecnico di Torino, Turin, Italy, 2022. [Online]. Available: http://webthesis.biblio.polito.it/id/eprint/25487. [Accessed: Aug. 2023]

[36] M. Platzer and P. Puschner, "Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation," in *33rd Euromicro Conference on Real-Time Systems*, 2021, pp. 1:1–1:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13932/. [Accessed: Aug. 2023]

[37] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, "Spatz: A compact vector processing unit for high-performance and energy-efficient shared-L1 clusters," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9. [Online]. Available: https://doi.org/10.48550/arXiv.2207.07970. [Accessed: Aug. 2023]

[38] J. Lee, W. Kim, S. Kim, and J.-H. Kim, "Post-quantum cryptography coprocessor for RISC-V CPU core," in *International Conference on Electronics, Information, and Communication*, 2022, pp. 1–2. [Online]. Available: https://ieeexplore.ieee.org/document/9748834. [Accessed: Aug. 2023]

[39] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," in *Reconfigurable Computing: Architectures, Tools, and Applications*, ser. Lecture Notes in Computer Science, R. Woods, K. Compton, C. Bouganis, and P. C. Diniz, Eds., vol. 4943. 4th International Workshop on Applied Reconfigurable Computing, 2008, pp. 209–220. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-78610-8_21. [Accessed: Aug. 2023]

[40] "SCAIE-V," TU Darmstadt, 2022, GitHub repository. [Online]. Available: https://github.com/esa-tu-darmstadt/SCAIE-V. [Accessed: Sep. 2023]

[41] M. Platzer, "Vicuna - a RISC-V Zve32x vector coprocessor," 2021, GitHub repository. [Online]. Available: https://github.com/vproc/vicuna. [Accessed: Jan. 2023]

[42] K. Karuri, M. A. A. Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for ASIP design," in *Proceedings of the 42nd annual Design Automation Conference*, 2005, pp. 329–344. [Online]. Available: https://dl.acm.org/doi/10.1145/1065579.1065666. [Accessed: Aug. 2023]

[43] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem — overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, May 2008. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/1347375.1347389. [Accessed: Sep. 2023]

[44] M. Jørgensen, "A review of studies on expert estimation of software development effort," *Journal of Systems and Software*, vol. 70, no. 1–2, pp. 37–60, Feb. 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121202001565. [Accessed: Aug. 2023]

[45] C. Bazeghi, F. J. Mesa-Martinez, and J. Renau, "System and procesor design effort estimation," in *VLSI-SoC: Advanced Topics on Systems on a Chip*, ser. IFIP International Federation for Information Processing. Boston, MA, USA: Springer, 2009, vol. 291, pp. 1–21. [Online]. Available: https://doi.org/10.1007/978-0-387-89558-1_14. [Accessed: Aug. 2023]

[46] "GCC, the GNU Compiler Collection," Free Software Foundation. [Online]. Available: https://gcc.gnu.org/. [Accessed: May 2023]

[47] M. Poorhosseini, W. Nebel, and K. Gruttner, "A compiler comparison in the risc-v ecosystem," in *International Conference on Omni-layer Intelligent Systems*, 2020, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9191411. [Accessed: Aug. 2023]

[48] "The LLVM compiler infrastructure," LLVM Foundation. [Online]. Available: https://llvm.org/. [Accessed: Jun. 2023]

[49] "Questa advanced simulator," Siemens. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/. [Accessed: May 2023]

[50] W. Snyder, "Welcome to Verilator," Veripool, 2023. [Online]. Available: https://www.veripool.org/verilator/. [Accessed: Jun. 2023]

[51] S. Williams, "Icarus verilog," 2023. [Online]. Available: https://steveicarus.github.io/iverilog/. [Accessed: Jun. 2023]

[52] J. S. Sobolewski, "Cyclic redundancy check," in *Encyclopedia of Computer Science*, 4th ed., A. Ralston, E. D. Reilly, and D. Hemmendinger, Eds. Chichester, U.K.: John Wiley and Sons, 2003, p. 476–479. [Online]. Available: https://dl.acm.org/doi/10.5555/1074100.1074303. [Accessed: Aug. 2023]

[53] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *International Conference on Dependable Systems and Networks*. IEEE, 2004, pp. 145–154. [Online]. Available: https://ieeexplore.ieee.org/document/1311885. [Accessed: Aug. 2023]

[54] M. Büsch, "Generator for CRC HDL code," Michael Büsch - Software Engineering, 2023. [Online]. Available: https://bues.ch/cms/hacking/crcgen. [Accessed: Jul. 2023]

[55] "CRC-16-CCITT," AutomationWiki, 2018. [Online]. Available: http://automationwiki.com/index.php?title=CRC-16-CCITT. [Accessed: Feb. 2023]

[56] S. Rao, D. Mahto, and D. Khan, "A survey on Advanced Encryption Standard," *International Journal of Science and Research*, vol. 391, no. 1, pp. 711–724, Jan. 2017. [Online]. Available: https://www.ijsr.net/archive/v6i1/ART20164149.pdf. [Accessed: Aug. 2023]

[57] "Spike RISC-V ISA simulator," RISC-V International, 2011, GitHub repository. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim. [Accessed: Jun. 2023]

[58] "Zynq 7000 SoC," Advanced Micro Devices, 2023. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. [Accessed: Aug. 2023]

[59] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, Feb. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10049118. [Accessed: Sep. 2023]

[60] L. Benini, "Ibex: Tightly-coupled accelerators and ISA extensions," IIS-Projects ETH Zurich, 2021. [Online]. Available: https://iis-projects.ee.ethz.ch/index.php?title=Ibex:_Tightly-Coupled_Accelerators_and_ISA_Extensions. [Accessed: Aug. 2023]

[61] A. Stanitzki, "Edge AI on low-footprint RISC-V," Fraunhofer IMS, 2022. [Online]. Available: https://riscv.org/blog/2022/01/edge-ai-on-low-footprint-risc-v-alexander-stanitzki-fraunhofer-ims/. [Accessed: Aug. 2023]

[62] "Ibex RISC-V core," lowRISC, 2017, GitHub repository. [Online]. Available: https://github.com/lowRISC/ibex. [Accessed: Aug. 2023]

[63] A. Utz, "AIRISC - The RISC-V processor for embedded AI," Fraunhofer IMS, 2021, GitHub repository. [Online]. Available: https://github.com/Fraunhofer-IMS/airisc_core_complex. [Accessed: Aug. 2023]

[64] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed. San Francisco, CA, United States: Morgan Kaufmann, 2020.