

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Generating Hardware Modules via Binary Translation of RISC-V Binaries

João Miguel Curado Conceição

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Nuno Paulino

Second Supervisor: João Bispo

March 31, 2022

Resumo

Recentemente novos paradigmas como *Edge Computing* trouxeram a necessidade de desenvolver processadores com maior capacidade computacional, energeticamente eficientes e menos dispendiosos. Estes requisitos estão a tornar-se mais difíceis de atingir devido a vários entraves que a indústria de circuitos integrados tem vindo a enfrentar, tal como o fim da Lei de Moore a o fim do Dennard Scaling. Estes problemas têm feito as novas gerações de processos de produção exibirem ganhos de desempenho mais baixos, especialmente no caso do desempenho *single-threaded*, a uma menor diminuição de consumo energético e uma menor redução no custo por transístor. Para mitigar estes problemas várias alternativas foram desenvolvidas, como o desenvolvimento de aceleradores de *hardware* que são *workload specific*, como por exemplo *Neural Processing Units (NPU)* para aceleração de algoritmos de redes neuronais. No entanto estas alternativas requerem o desenvolvimento, validação e produção de *hardware*, que são processos extremamente demorados e dispendiosos, e por tal a produção de aceleradores dedicados para cada aplicação a ser acelerada não é plausível.

Os *Field Programmable Gate Arrays (FPGAs)* permitem o desenvolvimento de *custom hardware* sem necessitarem do dispendioso processo de produção. Esta alternativa tem vindo a ser unicamente utilizada como uma opção rentável quando a produção de *hardware* especializado não se justifica devido ao baixo volume de produção. Estes componentes são raramente utilizados no mercado de consumo, e nunca foram adoptados como aceleradores reconfiguráveis em fluxos tradicionais de desenvolvimento de *software*, dado que estes componentes requerem um profundo conhecimento em desenvolvimento sistemas digitais e um longo processo de validação. Recentemente novos processos de desenvolvimento para estes dispositivos que tentam tornar o uso destes componentes mais semelhante ao desenvolvimento de *software* têm sido apresentados, embora ainda necessitem de algum conhecimento em desenvolvimento de sistemas digitais.

Nesta dissertação foi desenvolvida uma ferramenta que efectua a geração, validação e caracterização automática de aceleradores em *hardware* utilizando aplicações já compiladas, em particular para a arquitectura RISC-V. Esta ferramenta tem como objetivo permitir a geração de aceleradores sem a necessidade de qualquer conhecimento em desenvolvimento de sistemas digitais e sem requerer qualquer alteração do código já desenvolvido.

Abstract

In recent years new paradigms such as Edge Computing have brought forth the need to develop processors that are more powerful, energy efficient and cost effective. These requirements are getting more difficult to meet due to various bottlenecks being faced by the Integrated Circuit (IC) industry, such as the breakdown of Moore's Law and Dennard Scaling. Due to these issues newer process node generations have not benefited from the same rate of performance gains, specially in single-threaded performance, power consumption decrease and reduction of cost per transistor. To mitigate these issues various approaches have been taken, such as the development of workload specific hardware accelerators, such as Graphical Processing Units (GPUs), but these approaches require the development of workload specific accelerators and the validation and manufacturing of hardware, which are lengthy and expensive processes.

Field Programmable Gate Arrays (FPGAs) allow the development of custom hardware accelerators without requiring the expensive manufacturing process of traditional workload specific hardware accelerators. These devices have been mostly used only as a cost effective alternative in low production volume applications. These devices are rarely used in the consumer market, and have never been adopted as re-configurable hardware accelerators in traditional software development flows due to the fact that they require extensive expertise in digital hardware design and require a lengthy validation process. Some alternative workflows that try to bring designing FPGA hardware accelerators closer to software development have been tried, but current offerings still require some digital hardware design expertise and the software's code base to be altered to support these workflows.

In this thesis, a tool-chain that enables the automatic generation and validation of FPGA hardware accelerators from compiled binaries, targeting in particular the RISC-V Instruction Set Architecture (ISA), was developed. This tool-chain allows the generation of custom hardware accelerators without requiring any digital hardware expertise and without requiring changing any of the target software's code base.

Acknowledgements

I would like to thank my supervisor Nuno Paulino and my second supervisor João Bispo for giving me valuable advice and helping me throughout the duration of this thesis.

I would also like to thank my father, because of his unrelenting support and always trying to comprehend my views, my sister for always being there whenever I needed and for teaching me to take life more lightly, and my grandparents, for teaching me what hard work and dedication are. Lastly I would like to thank my mother for never having stopped believing in me until the end.

I would also like to thank all of the authors of the open-source tools and libraries used in this thesis, since their hard work, without any sort of compensation, has allowed me to do this thesis, that would have not been possible if these free tools were not available.

João Miguel Curado Conceição

“We need to build computers for the masses, not the classes”

Jack Tramiel, Founder of Commodore International

Contents

Resumo	i
Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Context	1
1.2 Motivation	3
1.3 Objectives	3
2 State of art	5
2.1 Binary translation	5
2.1.1 QEMU	5
2.2 Generation of custom instructions	6
2.2.1 Instruction Set Extensions in an FPGA Soft Core	6
2.2.2 Efficient Custom Instructions Generator	7
2.2.3 RISC-V ISA Single Instruction Multiple Data instructions extension	8
2.3 Automatic generation of HDL	8
2.3.1 Xilinx Vitis HLS	8
2.3.2 Chisel	9
2.3.3 Calyx compiler infrastructure	10
2.4 RISC-V	11
2.4.1 Xuantie-910	11
2.4.2 Rocket Chip Generator	12
2.4.3 Ibex RISC-V Core	13
3 Proposed approach	15
3.1 Overview	15
3.2 SPeCS Binary Translation Framework	17
3.2.1 Frequent sequence extraction	17
3.2.2 Assembly to Intermediate Language translation	17
3.2.3 Parse Tree	19
3.3 Control and Data Flow Graph	20
3.3.1 Structure	20
3.3.2 Visual representation	21
3.3.3 Generation	22
3.3.4 Optimizations	24
3.4 Module generation	27

3.4.1	Hardware Description Language generation	27
3.4.2	Synthesis	29
3.5	Validation	30
3.5.1	Functional validation	31
3.5.2	Timing verification	34
4	Results	37
4.1	Devices tested	37
4.2	Benchmarks	39
4.3	Results	42
4.3.1	Stage 1	42
4.3.2	Stage 2	43
4.3.3	Stage 3	47
4.3.4	Impact of sequence characteristics on the performance gain obtained . . .	52
5	Conclusion and Future Development	57
5.1	Future Development	58
	References	59

List of Figures

2.1	Overview of the framework proposed by Biswas et al. [1].	7
2.2	Different stages of the Vitis HLS on the example code snippet.	9
2.3	Example of the generation of HW modules using Calyx presented by Rachit Nigam, et al. [2].	11
2.4	Diagram of the pipeline of a Xuantie-910 core presented by the T-Head Division [3].	12
2.5	Rocket Chip Generator processor sub-components presented by Krste Asanovic et al. [4].	13
2.6	Block diagram of the Ibex RISC-V core presented in the Ibex GitHub repository.	14
3.1	Flow graph of the developed tool-chain.	16
3.2	Example of extraction of a frequent sequence.	17
3.3	Simplified parse tree generated from the <i>slt</i> RISC-V RV32I instruction.	19
3.4	DOT representation of the generated CDFG of an example instruction.	22
3.5	DOT representation of the generated CDFG of the RISC-V RV32I <i>slt</i> instruction.	24
3.6	Result of the redundant register elimination pass on the CDFG of the sequence shown in Listing 3.4.	25
3.7	Stages of the register name resolving pass on an example CDFG.	26
3.8	Simplified view of the AST of the generated module for the <i>slt</i> RISC-V RV32I instruction.	28
3.9	Expanded view of the DeclarationBlock branch shown in Figure 3.8.	28
3.10	Expanded view of the AlwaysCombBlock branch shown in Figure 3.8.	29
3.11	Schematic of the synthesized module of the RISC-V RV32I <i>slt</i> instruction.	30
4.1	Generated CDFG of the extracted sequence in the hydro2d benchmark presented in Listing 4.5.	44
4.2	Generated CDFG of the extracted sequence in the avg16 benchmark presented in Listing 4.6.	45
4.3	Average speed up obtained per extracted sequence of a benchmark in the Livermore Loops benchmark set.	47
4.4	Average speed up obtained per extracted sequence of a benchmark in the PolyBench/C benchmark set.	49
4.5	Average speed up obtained per extracted sequence of a benchmark in custom benchmark set.	51
4.6	Speed-up per number of instructions in a frequent sequence in the avg16 custom benchmark.	53
4.7	Speed-up per number of instructions in a frequent sequence in the add8par and add8seq custom benchmarks.	54

4.8 Speed-up per number of instructions in a frequent sequence of the conv2x2 benchmarks. 55

List of Tables

3.1	Summary of the vertices of the tool-chain’s CDFGs.	21
3.2	Summary of the edges of the tool-chain’s CDFGs.	21
3.3	Summary of the visual representation of the vertices of the tool-chain’s CDFGs. . .	21
3.4	Summary of the visual representation of the edges of the tool-chain’s CDFGs. . .	22
4.1	FPGAs chosen.	37
4.2	CPUs chosen.	38
4.3	Clocks Per Instruction of the chosen CPU cores.	38
4.4	Benchmarks used.	40
4.5	Extracted sequences per benchmark.	42
4.6	Extracted sequences averages per benchmark.	43
4.7	Resource utilization and maximum operating frequency of each synthesized generated module from the Livermore Loops benchmark set.	48
4.8	Resource utilization and maximum operating frequency of each synthesized generated module from the PolyBench/C benchmark set.	50
4.9	Resource utilization and maximum operating frequency of each synthesized generated module from the custom benchmark set.	52
4.10	Decrease in maximum operating frequency with the inclusion of a multiplication.	56

Abbreviations

ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BT	Binary Translation
CDFG	Control and Data Flow Graph
CPU	Central Processing Unit
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
GCC	GNU C Compiler
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
IC	Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
IR	Intermediate Representation
IL	Intermediate Language
ISA	Instruction Set Architecture
NPU	Neural Processing Units
RISC	Reduced Instruction Set Computer
SoC	System on Chip
SSA	Single Static Assignment

Chapter 1

Introduction

1.1 Context

In recent years the rise of Internet of Things (IoT) and other distributed devices, and the ever increasing computational needs have brought forth an alternative to the current data center based paradigm [5]. This new paradigm is called Edge Computing, and it allows for lower latency, more data privacy and lower use of the overall network bandwidth in comparison to the current paradigm [6].

Edge computing is defined by the proximity of the computational node to the consumer. This contrasts with the traditional data center based paradigm, where the computational nodes are concentrated in large data centers, that generally are very far from the consumer. Edge computing is done by either using edge servers that are equivalent to typical servers used in data centers or lower power versions of those, that are placed in small scale data centers that are much closer to the consumer than the traditional large scale ones, or by using embedded solutions, where the computational node is as close as possible to the consumer.

This new paradigm, specially in the embedded implementation, has the need for ever more powerful, energy efficient and cost effective processors [7]. These requirements are getting more difficult to meet because of various bottlenecks that the Integrated Circuit (IC) industry has been facing.

The most relevant bottlenecks faced by the industry are the breakdown of Moore's Law, which states that the density of transistors in ICs should double every two years, and the breakdown of Dennard Scaling, which states that as transistors get smaller their power density remains constant. The breakdown of Moore's Law has contributed to each new consecutive process node generation to deliver fewer benefits in terms of performance gain, lower power consumption and cost per transistors [8], and the breakdown of Dennard Scaling has lead to a stagnation of the maximum operating frequency of processors [9] which lead to single-threaded performance in processors to not increase significantly in new process node generations.

Various approaches have been taken to mitigate these issues such as the use of multi-core Central Processing Units (CPUs), to improve the parallel execution of workloads, generally called

multi-threaded workloads. This approach does not improve performance in all types of workloads, because not all application are easily paralellizable via threading due to dependencies between data or tasks [10, 11].

Another approach that has been taken is the use of more specialized Hardware (HW), such as recently the integration of Neural Processing Units (NPU) in some portable device's System on Chip (SoC) used today, such as in smartphones, to accelerate machine learning workloads while being more energy efficient than the CPU also present. In an ideal setting, specialized HW units would be created to accelerate the required workloads while offering better energy efficiency. Due to the fact that the manufacturing of HW devices is a long and expensive procedure, the manufacturing of processors with HW units to accelerate the required workloads is not plausible.

Instead, the use of Field Programmable Gate Array (FPGA) as platforms for future embedded systems on the edge is a possibility. FPGAs are devices where a grid of interconnected logic blocks can be reconfigured (any number of times) in order to implement arbitrary functionality. Although originally intended for prototyping, the increasing density of logic blocks in FPGAs, their increasing operating clock frequencies, and the evolution of compilation tools has powered the adoption of these devices in recent years [12]. The use of FPGAs to implement Application Specific Integrated Circuit (ASIC) is sensible when the production volume is too low to compensate the overhead costs of full ASIC solutions. These devices can come as discrete devices or they can be integrated with CPUs and other processing units in a SoC.

Presently the embedded market is dominated by ARM architecture devices¹, due to the fact that they are available in different processor configurations that specialize in different applications, from high-performance to low-power consumption. These devices have a proprietary Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA).

An open-source freely available alternative to the ARM ISA, called RISC-V, was developed by Prof. Krste Asanović, Yunsup Lee and Andrew Waterman at UC Berkeley in 2010 [13, 14]. Recently this ISA is gaining popularity due to various reasons such as it being open source, free to use, and it allowing the use of only the necessary ISA specification standard instruction set extensions and the use of custom instruction set extensions if needed, that allows manufacturers to add instructions not part of the official RISC-V ISA[3]. This allows RISC-V processors to be extremely flexible since they can be tailored to the manufacturer's needs, by only using the required ISA specification standard instruction set extensions and using custom instruction set extensions if needed, which enables them to be customized to the application's needs.

In summary, conventional single-thread processors can no longer rely on technological benefits of new process nodes to improve performance. This problem is compounded by the issue of power consumption, since embedded and edge systems are constrained regarding energy supply. Under these conditions, a possible solution to support emergent edge applications is increased device heterogeneity via circuit specialization. ASICs benefit from higher performance, and lower energy requirements. However, ASIC design has, so far, been very costly in terms of design, and fabrication. In this regard, the evolution of FPGAs as deployment devices can help mitigate

¹Statista ARM market share report: <https://www.statista.com/statistics/1132112/arm-market-share-targets/>

manufacturing costs, and the increasing popularity of open architectures like the RISC-V have revitalized the embedded ecosystem.

1.2 Motivation

Since there are major manufacturers entering the FPGA market (such as the acquisition of Xilinx Inc. by Advanced Micro Devices, Inc. (AMD) or the acquisition of Altera Corporation by Intel Corporation), FPGAs could become more widespread and therefore more prevalent in Edge Computing, due to the fact that they allow for better energy efficiency when compared with general purpose processors, such as CPUs [15].

This adoption is hindered by the fact that FPGAs designs are designed by relying on Hardware Description Language (HDL). This typically requires expert knowledge on how digital HW operates and is designed, so they are not compatible with traditional software development workflows and require HW design expertise, and they require extensive and lengthy validation that generally takes up most of the development time.

High-Level Synthesis (HLS) workflows have emerged to address this [16], and have been an area of active interest, inclusively by FPGA vendors such as Xilinx². These workflows allow for (subsets of) higher level programming languages such as C/C++ or OpenCL to be translated into HDL automatically which is then synthesized by the conventional underlying circuit generation flow [17, 18]. This makes the development of these modules more akin to software development, but knowledge of hardware design is still beneficial, or even required, for performing solutions. They are also not easily integrated into a traditional software development workflow, and due to their nature already compiled programs can not be accelerated with this approach, unless the source code is available and is modified to use it.

For embedded applications SoCs that integrate a CPU+FPGA are more appropriate than using discrete FPGAs, because of their higher energy efficiency, better performance and possibly lower overall cost, that comes from tighter integration. Presently most of these SoCs contain one or more ARM ISA based CPUs (including multi-cores), such as the market leader Xilinx UltraScale+ family of devices, although there are designs that use instead a RISC-V CPU such as the Microsemi PolarFire SoC family of devices³. Since the RISC-V ISA allows for custom instruction extensions, in a setting in which a CPU is integrated in an SoC with a FPGA, it allows hardware modules on the FPGA to be possibly used as custom instruction set extensions.

1.3 Objectives

The issues mentioned in 1.2 proposes the following problem:

²Xilinx Vitis HLS website: <https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html>

³Microsemi PolarFire SoC website: <https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga>

What are the performance benefits achievable by the automated generation of HW accelerators for the RISC-V architecture using already compiled binaries?

The goal of this thesis is to develop a solution to the previously mentioned problem. In order to accomplish that goal, a tool-chain was developed that is able to:

- Detect hot-spots in RISC-V binaries, such as loops or frequent blocks of instructions, in compiled code, either by statically analyzing the software's code, or by analyzing the stream of instructions from the execution of the software and extract the relevant sequences of instructions;
- Automatically generate HW accelerators and applying optimizations in order to generate better performing and synthesizable modules;
- Validate and characterize the generated accelerators, so that the tool-chain generates fully functional accelerators with known characteristics, such as resource utilization and critical path delay, and the user can choose which accelerators to use.

With the proposed solution, it is hoped to achieve an alternative flow to current offerings, that is able to generate HW accelerators without the need of any manual HW design, without requiring the lengthy validation process of FPGA development, and in a way that is transparent to the software programmer, allowing it to be seamlessly integrated in traditional software development workflows. In this work, we will consider system architectures where the custom hardware is implemented as an embedded FPGA into a RISC-V pipeline, and therefore the performance gains come from saved clock cycles by creating custom instructions without manual development effort. We evaluate, as a function of custom instruction size (number of instructions in the sequence), how the performance varies, also taking into account the influence that the new unit has on operating frequency.

Chapter 2

State of art

2.1 Binary translation

Binary Translation (BT) is a technique in which software binaries that were compiled for a given ISA, generally called source instruction set, are recompiled to be deployed in another one, generally called target instruction set [19]. This translation can be either static, in which the binaries are translated by analyzing the software's code without executing it, and dynamic, in which the software is executed and the stream of the executed instructions is analyzed, and it can be performed in software or in HW[20].

2.1.1 QEMU

The QEMU machine emulator¹, in which the internal architecture is described by Fabrice Bellard in [21], is a dynamic binary translator that supports the vast majority of current ISA has either source or target instruction sets, and is able to emulate all of the subsystems of a computing system (CPU, input and output devices, user interface, etc). This emulator is currently one of the most popular, due to its modularity and open-source nature.

The CPU emulation subsystem, is based on the principle of splitting the individual source instruction set's instructions into simpler instructions called micro operations. These micro operations are hand written C functions that represent the individual components of an instruction, such as the registers used and the operation performed. The registers of the source instruction set CPU are mapped to host instruction set registers, so they are rapidly accessed.

An example of the translation of a PowerPC instruction to be run in a x86 platform is given by the author. In Listing 2.1 the example instruction is shown, which is the *addi* instruction that represents the addition of a register with an immediate value.

¹QEMU website: <https://www.qemu.org/>

```
addi r1,r1,-16 // r1 = r1 - 16
```

Listing 2.1: Original PowerPC *addi* instruction.

In Listing 2.2, the micro operations generated from the example instruction by the tool’s binary translator are shown.

```
movl_T0_r1 // T0 = r1
addl_T0_im -16 // T0 = T0 - 16
movl_r1_T0 // r1 = T0
```

Listing 2.2: Micro operations of the PowerPC *addi* instruction generated by QEMU’s binary translator.

Each of the generated micro operations corresponds to a C function. As an example the first micro operation corresponds to the C function shown in Listing 2.3.

```
void op_movl_T0_r1(void)
{
    T0 = env->regs[1];
}
```

Listing 2.3: C function of the *movl_T0_r1* micro operation.

2.2 Generation of custom instructions

In this section some relevant frameworks for generating custom instructions are presented.

2.2.1 Instruction Set Extensions in an FPGA Soft Core

In the work of Partha Biswas et al [1], a unnamed framework for generating instruction set extensions is presented. The framework presented is able to generate custom instruction set extensions that allow for an average 1.41x speedup while consuming only up to 60% the energy.

The framework receives as an input a high-level application in C and it profiles the application’s source code and detects the sequences of instructions that present the largest delay. These sequences are then used to automatically generate custom instructions, by generating graphs that represent the sequence’s algorithm. After the custom extensions are generated, corresponding HW modules are generated using pre-designed component libraries, and these modules are then synthesized and an interface to the soft-core is also generated. The input application is also modified by the framework so that it can exploit the generated modules. In Figure 2.1 an overview of the various stages of the proposed framework is shown.

The MicroBlaze soft-core architecture is used as an example, and in the experiments performed the code size of the benchmarks used is lower due to the new instructions generated replacing multiple instructions in the original source code. In most of the benchmarks a significant speedup of around 1.4x is achieved with 40% energy savings when compared to an equal soft-core that does not use the custom instructions. The soft-cores with the custom instruction set extensions utilize on average 17% of the slices available in the FPGA used, and the soft-core without instruction set extension utilizes 11% of the available slices.

The authors claim that this approach could possibly be used for other ISAs and FPGA architectures.

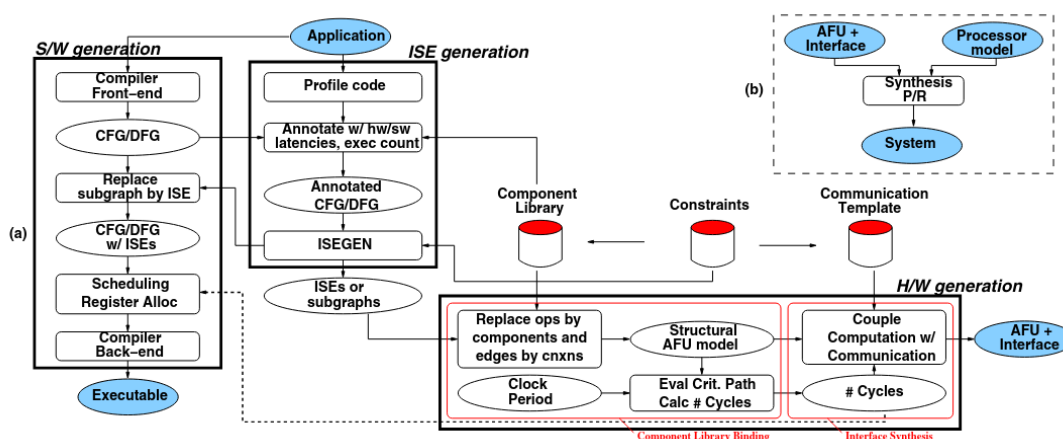


Figure 2.1: Overview of the framework proposed by Biswas et al. [1].

2.2.2 Efficient Custom Instructions Generator

In a paper published by Huynh Phung Huynh, Yun Liang and Tulika Mitra [22], an algorithm for the automatic generation of custom instructions is presented. This algorithm advocates a top-down approach, in which the system requirements guide the entire design flow in order to guarantee that the requirements are always met. The main requirement is generally the deadline to perform the required task, and this algorithm evaluates if there is a possible way to generate a custom instruction that allows the processor to meet the deadlines. If such is not possible this algorithm will output custom instructions that will speed up the task as much as possible. One noteworthy characteristic of this algorithm is that if required, it will successively generate more custom instructions in order to further improve performance. In the benchmarks performed, this algorithm generated custom instructions that showed a large speedup when compared to the same system without the custom instructions, while generating these custom instructions under 10 seconds in most of tests performed.

2.2.3 RISC-V ISA Single Instruction Multiple Data instructions extension

In the paper published by Philippos Papaphilippou, Paul H. J. Kelly and Wayne Luk [23], the creation of custom instruction extensions for the RISC-V ISA targeting Single Instruction Multiple Data instructions on a custom soft-core processor is presented. The instructions of this custom instruction set extension are hand-designed, and they were developed in such a way that they can be implemented as variations of the base type of instruction type defined in the RISC-V ISA specification of base instruction set extensions. The proposed instruction extensions are benchmarked and comparison of the custom soft-core with the custom instruction extensions with a commercial ARM A53 core based CPU is also performed.

The soft-core with the developed instruction extensions showed a large performance gain from using these instructions (from 4.1 times to 12.1 times depending on the benchmark), while offering around 0.4 times the performance of commercial ARM A53 core. The authors claim that if application specific custom accelerators are used, the performance gains achieved could be as large as 49 times on some workloads.

2.3 Automatic generation of HDL

In order to implement the generated custom instruction set extensions, where the system they will be deployed in is a SoC containing a CPU and a FPGA, it is necessary to synthesise HW modules that are able to compute the logical operations they represent. As mentioned in Section 1.2, the manual design of these modules is extremely time consuming, so in order to make the generation of the custom instruction set extensions truly automatic, the automatic generation of HDL for synthesising HW modules is necessary.

In this section various frameworks that allow the automatic generation of HDL from either a high-level language or a more abstract description of the logic to be implemented are presented.

2.3.1 Xilinx Vitis HLS

Xilinx Vitis HLS² is a software infrastructure that aims to integrate the use of an HLS workflow in the Xilinx ecosystem of devices and software. Xilinx does not publish papers on the inner workings of this workflow, but recently some of its code base has been released as open source software. It uses the LLVM compiler infrastructure which is a popular open-source compiler infrastructure, that is designed to be independent of the source language and the target instruction set used, making it an universal and portable compiler. Xilinx's HLS implementation uses C/C++ or OpenCL as source languages and Verilog or VHDL as target languages. The main advantages of this workflow are that, as mentioned in Section 1.2, lower expertise of HW design required, and as described in a paper by Declan O'Loughlin et al. [24], the more efficient utilization of FPGA resources when compared to using traditional HDL workflows.

²Vitis Model Composer website: <https://www.xilinx.com/products/design-tools/vitis/vitis-model-composer.html>

This HLS workflow simplifies the generation of HDL by synthesizing loops and other repeating parts of code as finite state machines, and serial operations as a pipeline, where each stage of the pipeline can be composed of one or more parallel operations.

In the Vitis HLS User Guide³, some examples of the different phases of Vitis are presented.

In Figure 2.2 the different stages of Vitis HLS performed on the example code snippet shown in Listing 2.4 is shown. The Scheduling Phase representation is equivalent to the HDL generated by Vitis HLS. In the following phases the design will be progressively optimized for the target FPGA architecture.

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```

Listing 2.4: Example C program given in the Vitis HLS User Guide.

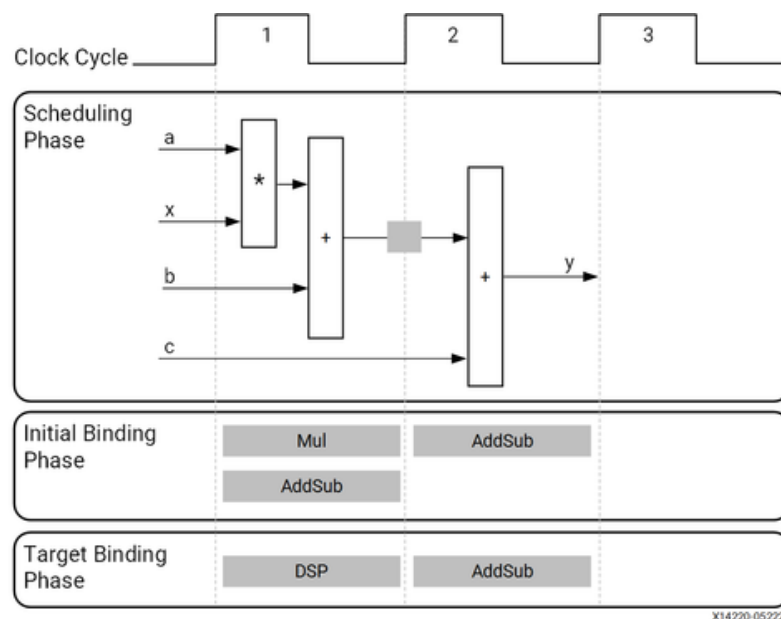


Figure 2.2: Different stages of the Vitis HLS on the example code snippet.

2.3.2 Chisel

In the work of Jonathan Bachrach et al. [25], a new hardware construction language called Chisel is introduced. This language aims to abstract HW as much as possible by introducing concepts such as object oriented design and functional programming. This language is implemented on

³Vitis High-Level Synthesis User Guide (UG1399): <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Basics-of-High-Level-Synthesis>

top of Scala, and the tool chain developed by the authors is able to generate C++ cycle-accurate simulators, similar to Verilator C++ test benches, and is able to generate low-level Verilog that can be used in standard FPGA or ASIC flows.

In the paper the authors took an already developed 3-stage 32-bit RISC processor that was written in Verilog, and converted it to Chisel. The implementation Chisel showed about a 3 times reduction in code size. The authors also designed a 64-bit Fused-Multiply-Add unit in Chisel and Verilog, and after the mapping process in a 65nm process ASIC flow, the total area occupied by both implementations was roughly equal.

In 2018 Google LLC used Chisel to develop its Google Edge TPU devices. In a presentation given by the engineers that worked on the project, their experience of using Chisel instead of a traditional HDL language was discussed⁴. The overall opinion was mixed, with the engineers criticizing the learning curve of Chisel and the difficulties that they had in the verification steps, and commending the productivity gains that Chisel provides in the design steps.

2.3.3 Calyx compiler infrastructure

In the work of Rachit Nigam, et al. [2] a new compiler infrastructure named Calyx⁵ is presented, that aims to be an open-source alternative to current HLS workflows, such as the Xilinx Vitis HLS mentioned in 2.3.1. In the benchmarks performed by the authors Calyx generated accelerators with better performance while utilizing slightly more resources than the Xilinx's HLS software infrastructure.

This compiler infrastructure makes use of the Calyx IR, that aims to combine an hardware-like structural language with software-like control flow. This contrasts traditional HLS workflows, that use high-level programming languages such as C/C++ that are more appropriate to software development since they do not contemplate the inherent parallelism that FPGAs allow. This IR is also designed to automate the generation of the control signals needed, and simplify the declaration of HDL modules (called groups in Calyx IR), simplify their sequential and/or parallel placement in the processing pipeline and the memory accesses that are performed.

Calyx also includes various optimization passes, that further optimize the generated circuit to better utilize the resources available on the FPGA while offering better performance. In Figure 2.3 the optimization passes performed by Calyx are shown.

⁴Experiences Building Edge TPU with Chisel: <https://www.youtube.com/watch?v=x85342Cny8c>

⁵Calyx Github repository: <https://github.com/cucapra/calyx/>

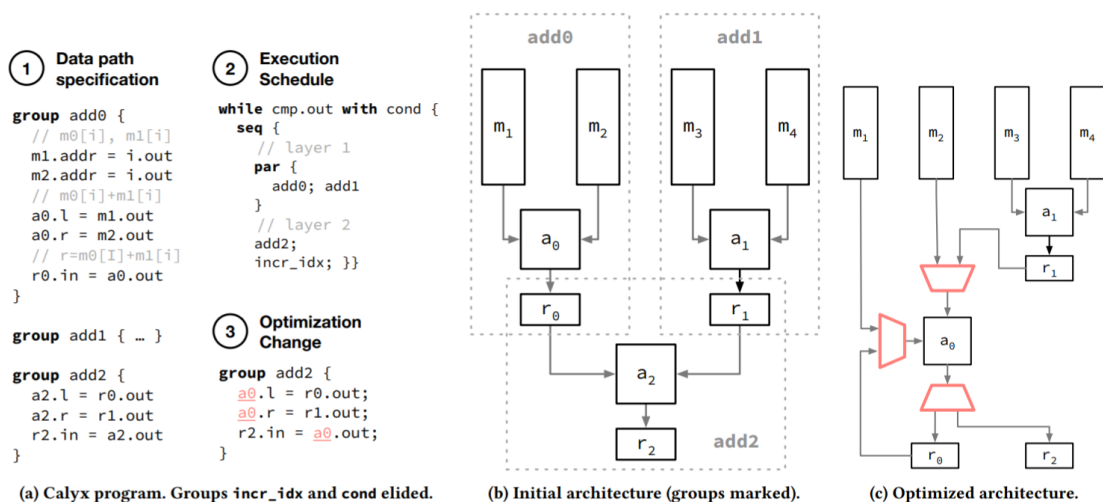


Figure 2.3: Example of the generation of HW modules using Calyx presented by Rachit Nigam, et al. [2].

2.4 RISC-V

As mentioned in Section 1.1 the RISC-V ISA aims to be an open source alternative to current commercial offerings such as the ubiquitous ARM ISA. Due to RISC-V being still extremely recent, with development starting only in 2010 as a research project at UC Berkeley, not many RISC-V CPUs commercial offerings are available, with currently Sifive⁶ and T-Head⁷ being the major providers of core IPs. Most of the currently available CPUs are designed and used in academic settings.

Currently the development of the RISC-V ISA is done by the RISC-V Foundation, that was established in 2015 by the original designers and 36 industry and academic partners, such as IBM, NVIDIA, Microchip and Lattice. Since it's establishment more major industry partners have become members such as Huawei, Analog Devices, Cadence and NXP.

In this Section relevant developments in the RISC-V ecosystem will be presented.

2.4.1 Xuantie-910

The Xuantie-910 CPU was developed by the T-Head Division of Alibaba Cloud [3]. This CPU is a multi-cluster Symmetric Multi Processing with cache coherence. Each cluster contains up to 4 cores that individually have a 12-stage deep pipeline. These cores are super-scalar, multi-issue and support out-of-order execution. This processor implements the RV64GCV base instruction set extensions and makes use of many custom instruction set extensions, such as arithmetic operation, bit manipulation, load and store, and cache operations. This processor showed single-core

⁶Sifive website: <https://www.sifive.com/>

⁷T-Head website: <https://www.t-head.cn/?lang=en>

performance equivalent to that of an ARM Cortex-A73, a micro-architecture released in 2016 that when released was used in a wide variety of high-performance mobile device processors.

This processor was developed to be used in the server space, specially by Alibaba’s cloud platform, so it can be considered a direct competitor to the other types of high-performance processors used in the server space. Alibaba has publicly committed to further developments on high-performance RISC-V processors, and recently many of Alibaba’s RISC-V CPU cores were made open-source⁸.

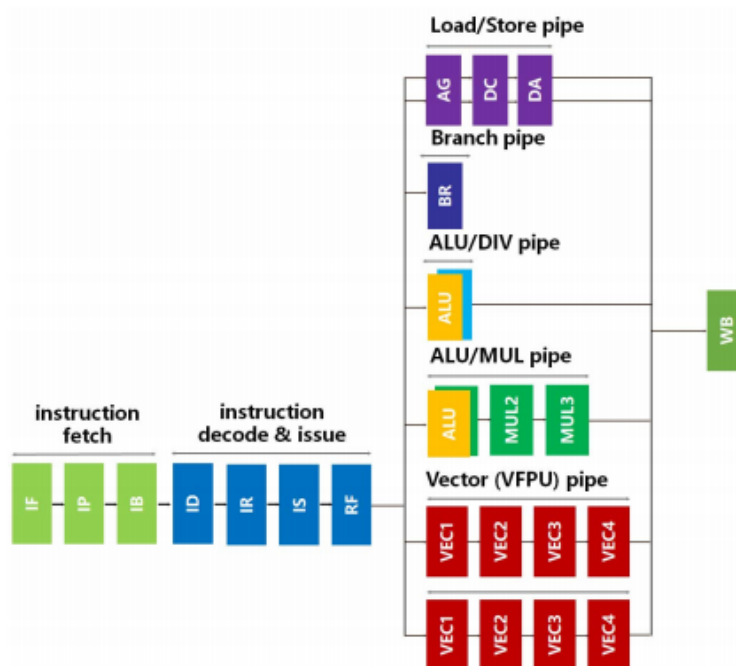


Figure 2.4: Diagram of the pipeline of a Xuantie-910 core presented by the T-Head Division [3].

2.4.2 Rocket Chip Generator

In the work of Krste Asanovic et al. [4], the open-source Rocket Chip Generator⁹ is presented. This tool generates RISC-V based general-purpose processors, and it supports the integration of custom accelerators in the form of instructions set extensions or co-processors. Both in-order and out-of-order processor cores can be generated, and all major RISC-V standard instruction set extensions are supported. One noteworthy aspect of this tool is that it allows the customization of any part of the generated processor, so that the generated processor meets the requirements of the user.

This tool divides the generation of processors into sub-components, shown in Figure 2.5, such as the generation of cores and caches, which also allows each of the sub-components of the generated processor to be individually validated.

⁸T-Head semiconductor GitHub repository: <https://github.com/T-head-Semi>

⁹Rocket Chip Generator Github repository: <https://github.com/chipsalliance/rocket-chip>

Due to the tool being written in Chisel, presented in Section 2.3.2, the benefits provided by this language such as the automatic generation of C++ cycle accurate simulators and the automatic generation of low-level Verilog are present. This tool has been used to tape out multiple non-commercial processors that are able to boot Linux.

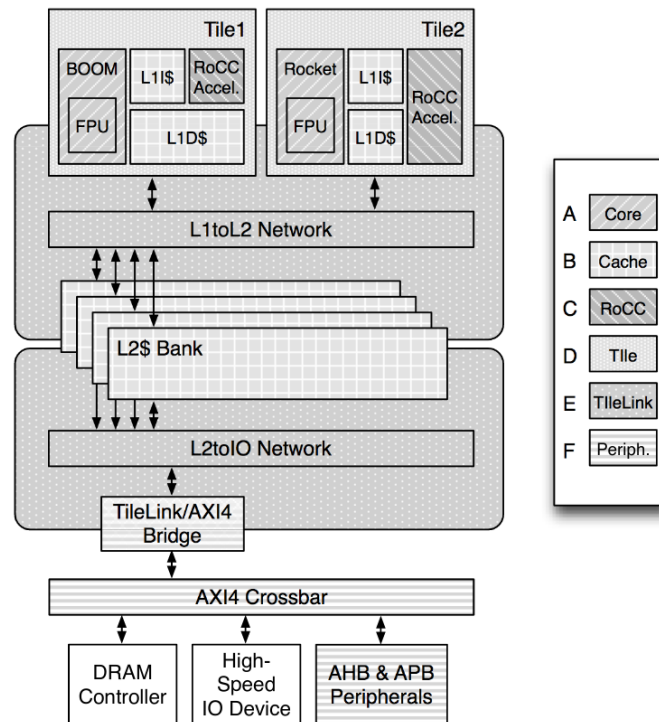


Figure 2.5: Rocket Chip Generator processor sub-components presented by Krste Asanovic et al. [4].

2.4.3 Ibex RISC-V Core

Ibex is an heavily parameterizable open-source production-quality 32 bit RISC-V CPU core¹⁰. This core is written in SystemVerilog and it is mainly suited for embedded control applications. It has been used in CPUs that have been taped-out due to this core having been extensively verified and validated. Depending on configuration this core can support the RV32I (Integer) or RV32E (Embedded), RV32M (Integer Multiplication and Division), RV32C (Compressed Instructions) and RV32B (Bit Manipulation) standard extension sets.

This core can be configured in many ways to meet the needs of the user, with four main configurations proposed by the developers: micro, small, maxperf and maxperf-pmp-bmfull. The different possible core configurations differ by which of the previously mentioned standard extension sets they support and the inclusion of certain features such as an HW multiplier (in configurations that support the RV32M standard extension) and the inclusion of more stages in the pipeline.

¹⁰Ibex RISC-V core GitHub repository: <https://github.com/lowRISC/ibex>

In Figure 2.6 the block diagram of the small configuration proposed by the developers is shown.

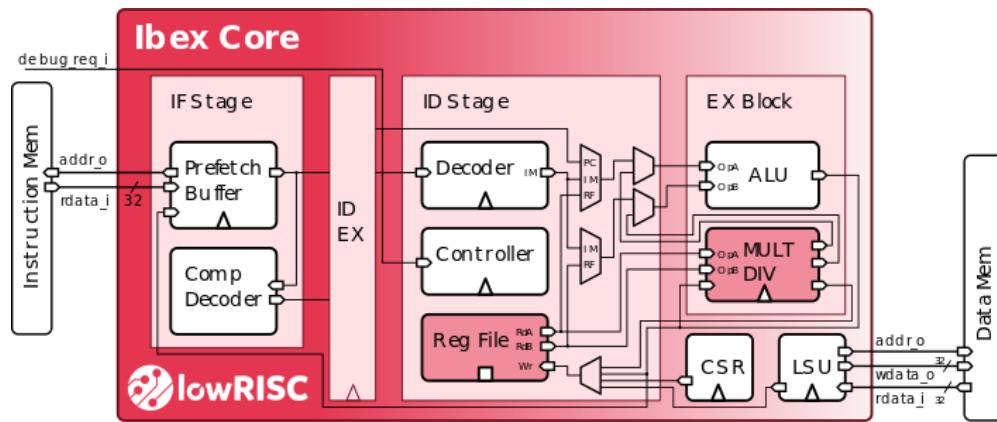


Figure 2.6: Block diagram of the Ibex RISC-V core presented in the Ibex GitHub repository.

Chapter 3

Proposed approach

3.1 Overview

As mentioned in Section 1.3, the goal of this thesis is to explore the performance gains that can be obtained by the automatic generation, validation and characterization of HW accelerators for RISC-V processors from already compiled code. In order to achieve this the developed tool chain needs to be able to automatically generate HDL from frequent segments extracted from RISC-V binaries, and it needs to automatically validate the generated module, in order to avoid the lengthy process of manual validation mentioned in Section 1.2.

In order to accomplish the stated goal, a tool-chain that is integrated with the Binary Translation Framework being developed by SPeCS¹ was developed. The flow graph of the stages of the developed tool-chain is shown in Figure 3.1.

The flow graph of the developed tool-chain can be divided in three main stages:

- **Stage 1:** Frequent sequence extraction and ISA abstraction:
 - The binary file is read and frequent sequences of instructions are detected and extracted;
 - The detected sequences are translated to an ISA-independent Intermediate Language (IL);
 - A parse tree (representation of the syntax of an instruction's Intermediate Language) is generated for each instruction in the sequence.

- **Stage 2:** Optimizations and HDL generation:
 - A Control and Data Flow Graph is generated by combining the generated parse trees;
 - Optimization passes are performed on the generated graph;
 - An HDL Abstract Syntax Tree is generated from the optimized graph, with Verilog and SystemVerilog currently supported.

¹SPeCS Binary Translation Framework Github repository: <https://github.com/specs-feup/specs-hw>

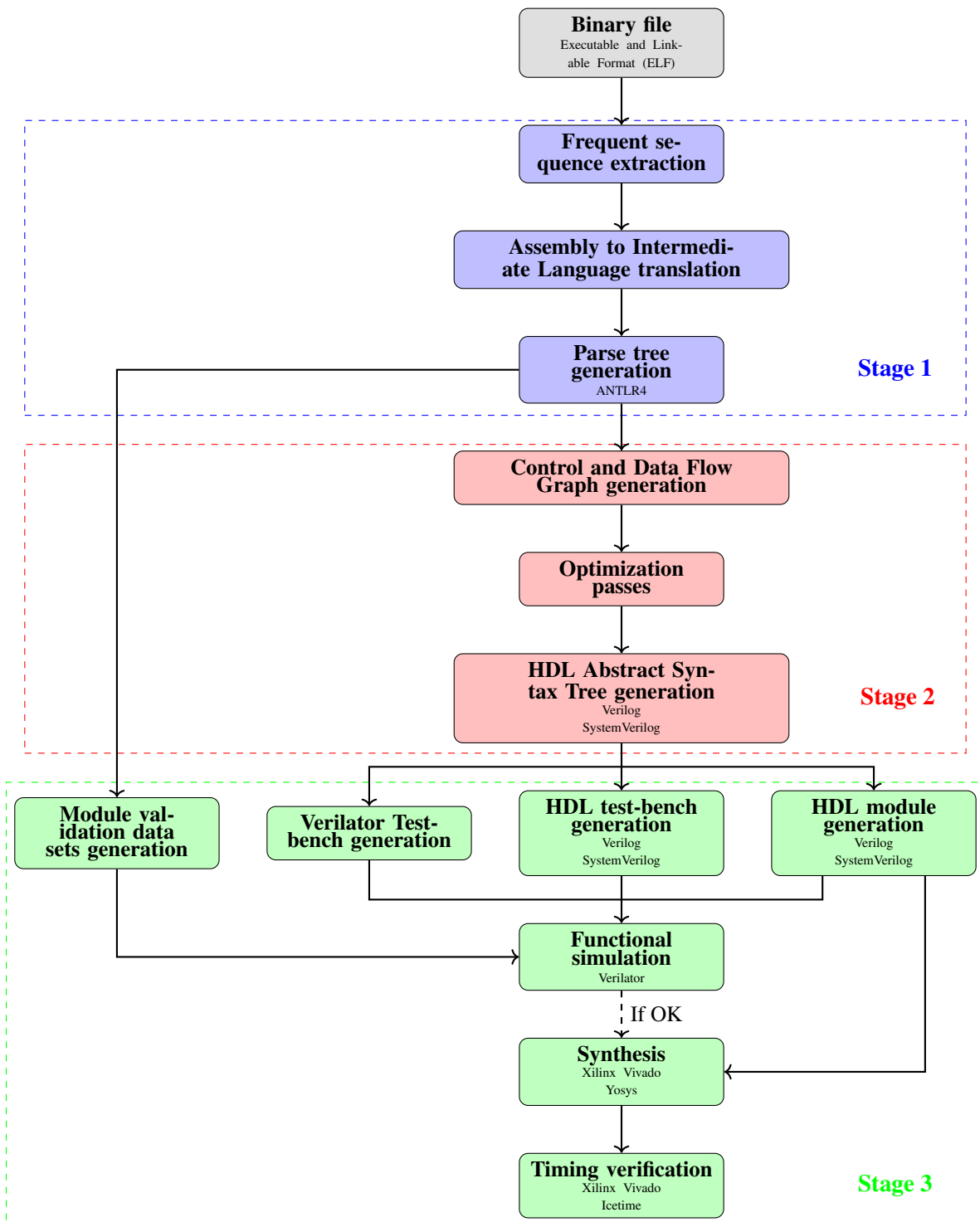


Figure 3.1: Flow graph of the developed tool-chain.

- **Stage 3:** Automatic validation, synthesis and characterization:
 - The test-benches and data sets required to validate the generated modules are generated;
 - The generated modules are functionally validated and synthesized;
 - A timing verification is performed on the synthesized modules.

The tool-chain was developed to be as modular as possible, so that each of the tool-chain's sub-components can be modified or new ones can be developed without requiring the modification of the code base of the remaining sub-components.

3.2 SPeCS Binary Translation Framework

The SPeCS Binary Translation Framework is an ISA independent framework, that aims to allow the development of tools that target already compiled binaries [26].

3.2.1 Frequent sequence extraction

One of the key features of this framework is that it is able to detect frequent sequences of instructions, called segments, in already compiled code. The framework is able to detect static segments (frequent sequences of instructions present in the binary file) and trace segments (frequent sequences of instructions that occur during the run-time of the program).

```

add t0, a1, t2
addi t3, t2, 10
mul t0, a2, a3
sll t1, t0, 2
add t1, a1, a2
addi t4, a2, 52

```

→

```

add ra, rb, rc
addi rd, rc, imm

```

Figure 3.2: Example of extraction of a frequent sequence.

In Figure 3.2 an example of the extraction of a frequent sequence is shown. The sequence detection pass not only takes into account the sequence of operations contained in the sequence but also the pattern of use of registers in the sequence, since the data-flow of the extracted sequence will be the same if the pattern of use of registers is the same. In the example shown, both of the detected sequences perform the same sequence of instructions and have the same pattern of register use.

3.2.2 Assembly to Intermediate Language translation

In this framework an IL is used to represent any instruction from any ISA, with ARM, MicroBlaze and RISC-V ISAs being currently supported. The syntax of the framework's IL is similar to that of C, but without certain features such as variable types and loops. For each instruction of

the supported ISAs, it is only required to define the IL representation of each instruction when initially implementing the instruction in the framework.

The framework represents each ISA as a set of properties such as encoding, instruction names and types, properties of each of the ISA's registers and a definition of the logical and arithmetic behavior of each instruction using the framework's IL.

In this thesis the RV32I (Base Integer Instruction Set), RV32M (Standard Extension for Integer Multiplication and Division) and RV32F (Standard Extension for Single-Precision Floating-Point) standard extensions were added to the framework. Due to the fact that the equivalent 64 bit standard extensions (the RV64I, RV64M and RV64F standard extensions) use the same op-codes and only introduce some 64 bit specific instructions, the IL representations of the 32 bit standard extensions added to the framework can be used for these extensions as well, with only the new instructions added by the 64 bit alternatives not being supported.

As an example of the framework's IL representation of a RISC-V RV32I standard extension instruction let's consider the *slt* (Set Less Than) instruction. This instruction compares the signed values of two registers (*rs1* and *rs2*) and 1 is written to *rd* if $rs1 < rs2$, otherwise 0 is written to *rd*. In Listing 3.1 the framework's IL representation of this instruction is shown.

```

if(signed(RA) < signed(RB)) {
    RD = 1;
} else{
    RD = 0;
};
// RA=rs1, RB=rs2, RD=rd

```

Listing 3.1: Example of the IL representation of the RISC-V *slt* instruction.

The framework's IL supports instructions with an arbitrary number of registers, such as in the case of the *fmadd.s* (Float Multiply and Add)RV32F standard extension instruction. This instruction multiplies *rs1* with *rs2* and adds the resulting value to *rs3*. This results is then stored in *rd*. In Listing 3.2 the framework's IL representation of this instruction is shown.

```

RD = (float(RA) * float(RB)) + float(RC);
// RA=rs1, RB=rs2, RC=rs3, RD=rd

```

Listing 3.2: Example of the IL representation of the RISC-V *fmadd.s* instruction.

The framework's IL also supports instructions that perform multiple operations such as in the case of the *jalr* (Jump and Link Register) RV32I standard extension instruction. This instruction stores the value of the address of the next instruction in *rd* and it jumps to the address described by $ra + imm$, with *imm* being an immediate value passed in the instruction. In Listing 3.3 the framework's IL representation of this instruction is shown.

```

RD = $pc + 4;
$pc = RA + sext (IMM);
//$pc=program counter, RA=rsl, IMM=imm field, RD = rd

```

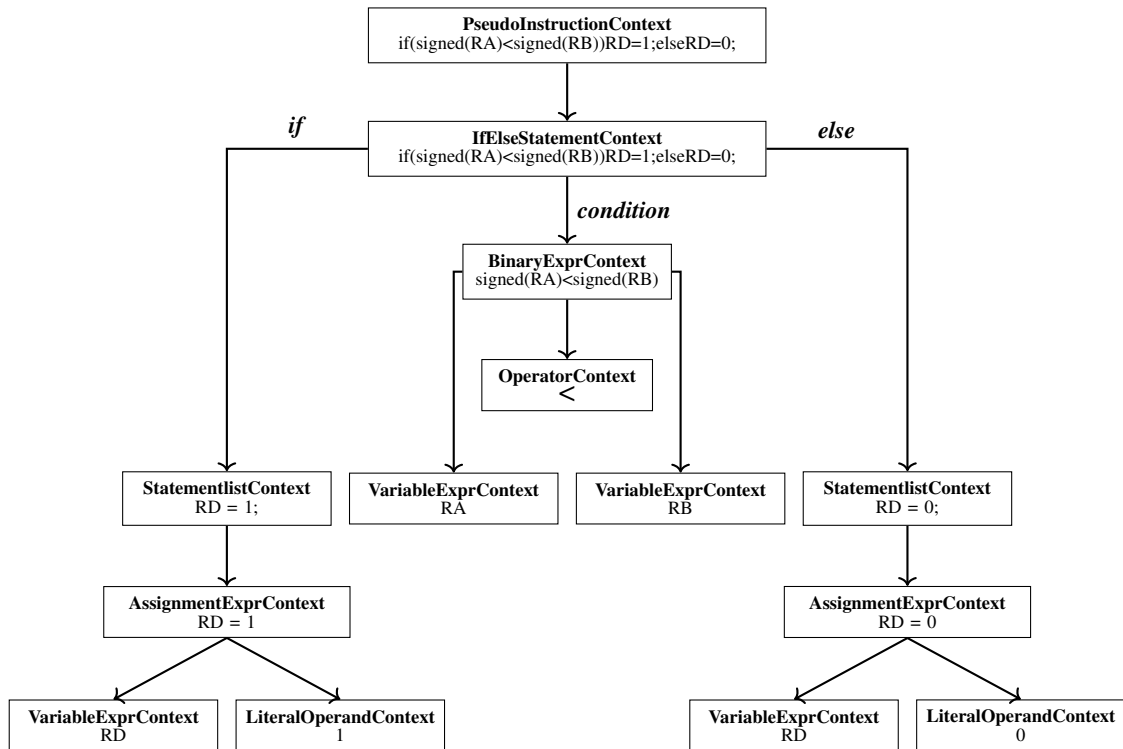
Listing 3.3: Example of the IL representation of the RISC-V *jalr* instruction.

3.2.3 Parse Tree

After the translation of the instructions of the extracted frequent sequence to the framework's IL, a parse tree is generated for each of these instructions. A parse tree is a representation of the syntax of a program according to the structure of the programming language, in this case the framework's IL. The generation of parse trees is done using the ANTLR4² parser generator.

The parse trees generated by the framework have two types of nodes, statements and expressions. Statements represent actions that will be executed without returning a value, such as conditional statements. Expressions are dependent on whether the node is a terminal node or a non-terminal node. If the expression node is a terminal node then it represents either data (variables or literals) or operators, and if the expression node is a non-terminal node then it represents operations that will be evaluated (calculated) and a value returned.

The simplified parse tree generated by the framework of the *slt* instruction, mentioned in Section 3.2.2, can be seen in Figure 3.3.

Figure 3.3: Simplified parse tree generated from the *slt* RISC-V RV32I instruction.

²ANTLR website: <https://www.antlr.org/>

Therefore, the use of IL (internally called pseudo-instruction) allows for specifying the arithmetic and logical behaviour of an instruction using any number of statements, as a function of the registers encoded in the instruction, and also implicit registers such as the PC. In a way, it is similar to approaches like QEMU, which emulate the instruction using C/C++ code. However, the distinction is that a separate language is used for the IL, which can be easily extended and separates the concerns between defining the instruction behaviour and evaluating its execution.

3.3 Control and Data Flow Graph

In most compiler flows, after the generation of the program's parse tree, this tree is transformed into a Control Flow Graph (CFG). This type of representation focuses on representing the control flow of the program, and they enable the simplification of some common operations performed by compilers such as the detection of loops and performing Single Static Assignment (SSA) passes, in which variables are given an unique identifier when they are assigned in order to simplify optimizations that the compiler might perform.

In the developed tool-chain, as in many HLS frameworks, the parse tree is transformed instead into a Control and Data Flow Graph (CDFG). A CDFG is a type of graph that combines CFGs with Data Flow Graphs (DFGs). DFGs focus on representing the data dependency of the program, which is useful for generating HDL since this type of representation enables performing optimizations that are relevant to better utilize the benefits that using FPGAs provide. One common optimization performed is the parallelization of operations, that is done in order to exploit the inherent parallelism acceleration of FPGAs.

In order to generate and interact with the tool-chain's CDFGs in a standardized manner, the open-source JGraphT³ library was used, since it includes most of the required base methods for generating and using graphs.

3.3.1 Structure

The structure of the generated CDFGs is similar to that of CFGs, with some additions that allow the simplification of the HDL generation steps that follow.

The CDFG is composed of subgraphs, that describe the control flow of the extracted frequent sequence, and these subgraphs are in turn composed of nodes, that describe the data flow of the sequence. A summary of the vertices of the tool-chain's CDFG is shown in Table 3.1 and a summary of the edges is shown in Table 3.2.

³JGraphT website: <https://jgrapht.org/>

Table 3.1: Summary of the vertices of the tool-chain's CDFGs.

Vertex type	Sub-type	Description
Subgraph	Control Flow	Same purpose of control nodes in a CFG
	Data Flow	Same as a DFG
Node	Control Flow	Indicates whether the control flow should be split or merged
	Operation	Represents an operation such as arithmetic operations and assignments
	Data	Represents immediate values or registers

Table 3.2: Summary of the edges of the tool-chain's CDFGs.







Edge type	Description
Control Flow	Used to indicate what conditional path the target subgraph is part of
Operand	Used to indicate the order of operands in operations that have two operands
Regular	Used for vertices in which for the target vertex the order of the sources does not matter, such as unary operations

3.3.2 Visual representation

During the development of the tool-chain, in order to allow for easier debugging and validation of the generated CDFGs, a DOT graph description language exporter was developed. The exported graphs can be viewed using a DOT viewer such as Graphviz Online⁴.

The visual representation of the structure of the generated CDFGs, presented in Section 3.3.1 is shown in Table 3.3 and Table 3.4.

Table 3.3: Summary of the visual representation of the vertices of the tool-chain's CDFGs.

Vertex type	Sub-type	Visual representation	
Subgraph	Control Flow		
	Data Flow		
Node	Operation		
	Data		
	Control Flow	Split	
		Merge	

⁴Graphviz Online viewer: <https://dreampuf.github.io/GraphvizOnline/>

Table 3.4: Summary of the visual representation of the edges of the tool-chain's CDFGs.

Edge type		Visual representation
Operand	Left	—□
	Right	—■
Regular		—→
Control Flow	True	◆→
	False	◇→

In Figure 3.4 the DOT representation of the generated CDFG of an example instruction is shown.

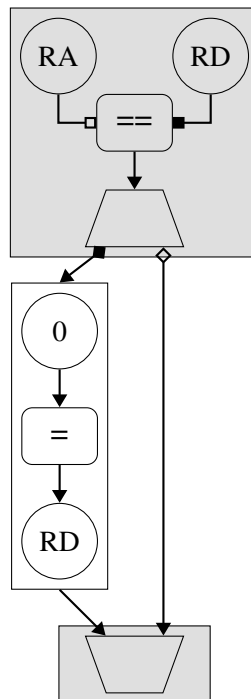


Figure 3.4: DOT representation of the generated CDFG of an example instruction.

3.3.3 Generation

The generation of a frequent sequence's CDFG in the developed tool chain is performed in two consecutive steps:

1. A CDFG is generated from each of the parse trees of the instructions present in the extracted frequent sequence;
2. The individual generated instruction CDFGs are merged into a CDFG that represents the entire sequence.

The generation of an instruction's CDFG from its parse tree is performed by visiting the tree and generating the appropriate CDFG components depending on what parse tree node is being visited.

The generation of the vertex types presented in Section 3.3.1 is related to what type of parse tree node is currently being visited. If the node being visited is a statement node then a subgraph is generated, and if the node being visited is an expression node that is a terminal node then a node is generated.

The type of subgraph that is generated depends on whether the visited statement is a conditional statement or not. If the statement being visited is a conditional statement (If or IfElse) then a Control Flow Subgraph with a Control Flow Split node is generated, otherwise a Data Flow Subgraph is generated. When visiting a conditional statement, first the statement's condition expression is used to generate a Data Flow Subgraph that is then inserted into the generated Control Flow Subgraph. Then the conditional paths are visited and the resulting generated subgraphs are connected to the statement's Control Flow Subgraph using the appropriate Control Flow edges. Finally a Control Flow Subgraph with a Control Flow Merge node is added to indicate that the control flow of the sequence has merged.

The generation of Data Flow Subgraphs is done by first visiting the terminal nodes of the parse tree's statement being visited and generating the appropriate Nodes according to the parse tree's nodes being visited. If the parse tree node being visited represents data (variables or literals) a Data Node is generated, and if the parse tree node being visited represents an operator, then an Operation Node is generated. The CDFG Nodes generated from the parse tree's terminal nodes are connected according to the non-terminal expression nodes of the statement being visited, since these expression nodes describe the operation being performed.

In Figure 3.5 the DOT representation exported by the tool-chain of the generated CDFG of the RISC-V *slt* instruction, mentioned in Section 3.2.2, is shown.

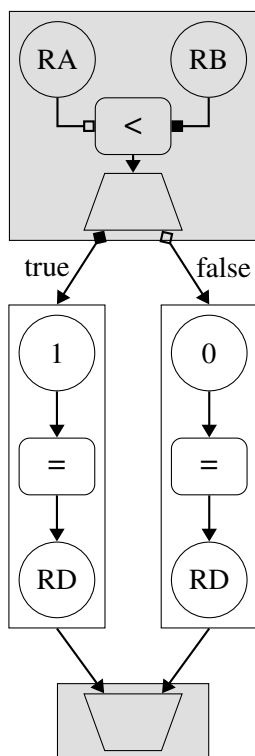


Figure 3.5: DOT representation of the generated CDFG of the RISC-V RV32I *slt* instruction.

3.3.4 Optimizations

In order to generate better performing and more efficient HDL, some optimization passes are done after the generation of the frequent sequence’s CDFG. These passes aim to reduce the amount of resources required for the HDL generation step and to ensure that the generated modules comply with HDL design best practices.

3.3.4.1 Redundant register elimination

This pass eliminates redundant registers present in the CDFG in order to minimize the resources that will be required during the HDL generation steps. In a sequence of instructions only the last assignment of a register is propagated to outside the sequence, so in registers that are written multiple times only the last write is propagated. This makes all other assignments temporary, so the value that was written in these assignments does not need to be stored and can be directly propagated to the next operation. The registers written in temporary assignments are considered redundant registers because these registers can be suppressed without affecting the overall behavior of the program.

In Listing 3.4 an example of a sequence that contains a redundant register is shown. In this sequence register *t0* is assigned twice. The first assignment is only used to temporarily store the result of $t1 + t2$, and in the last instruction this temporary value is overwritten. If this sequence is viewed as a black box, the first assignment of *t0* is not propagated to outside the sequence, so it

```

add t0, t1, t2
addi t0, t0, 10

```

Listing 3.4: Example of a sequence of instructions containing a redundant register.

will not be used by other instructions and it can be suppressed. In Figure 3.6 the original sequence's CDFG and the CDFG after this pass is applied are shown.

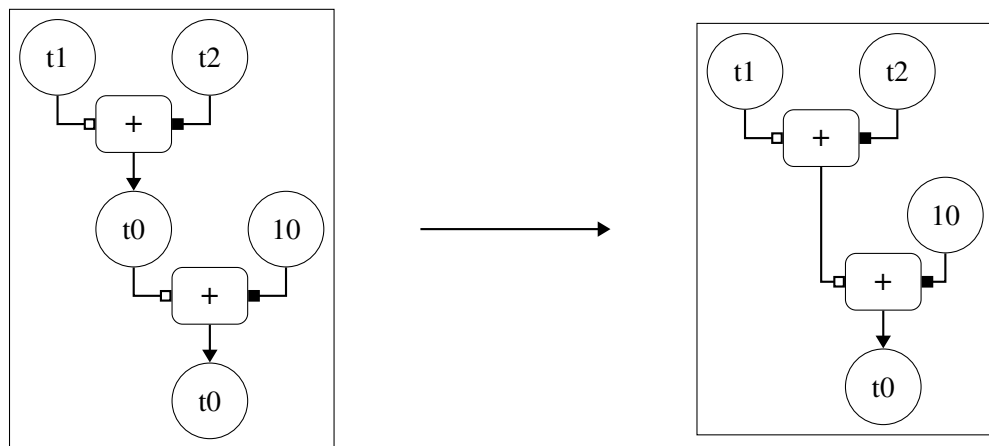


Figure 3.6: Result of the redundant register elimination pass on the CDFG of the sequence shown in Listing 3.4.

3.3.4.2 Register name resolving

In this context a typical SSA pass cannot be applied because in some cases SSA passes can lead to variables assigning themselves. This is problematic because in combinational HW design signals are not allowed to assign themselves, which is called a combinatorial loop. In order to solve this issue a modified SSA pass was developed.

This pass is performed in the following stages:

1. Replaces the generic IL names given to the CDFG registers with the names of the actual registers of the instructions in the sequence;
2. It performs a regular SSA pass on the CDFG, where all registers that are assigned are given an unique identifier. This step facilitates the generation of HDL because all registers can be mapped into unique signals;
3. It transforms if statements into if-else statements if a register was assigned within the if statement and resolves any unique identifier inconsistencies that may arise from this. This step solves the issues with typical SSA passes and ensures that all conditional statements have proper coverage and that the unique identifier of registers after conditional is not ambiguous.

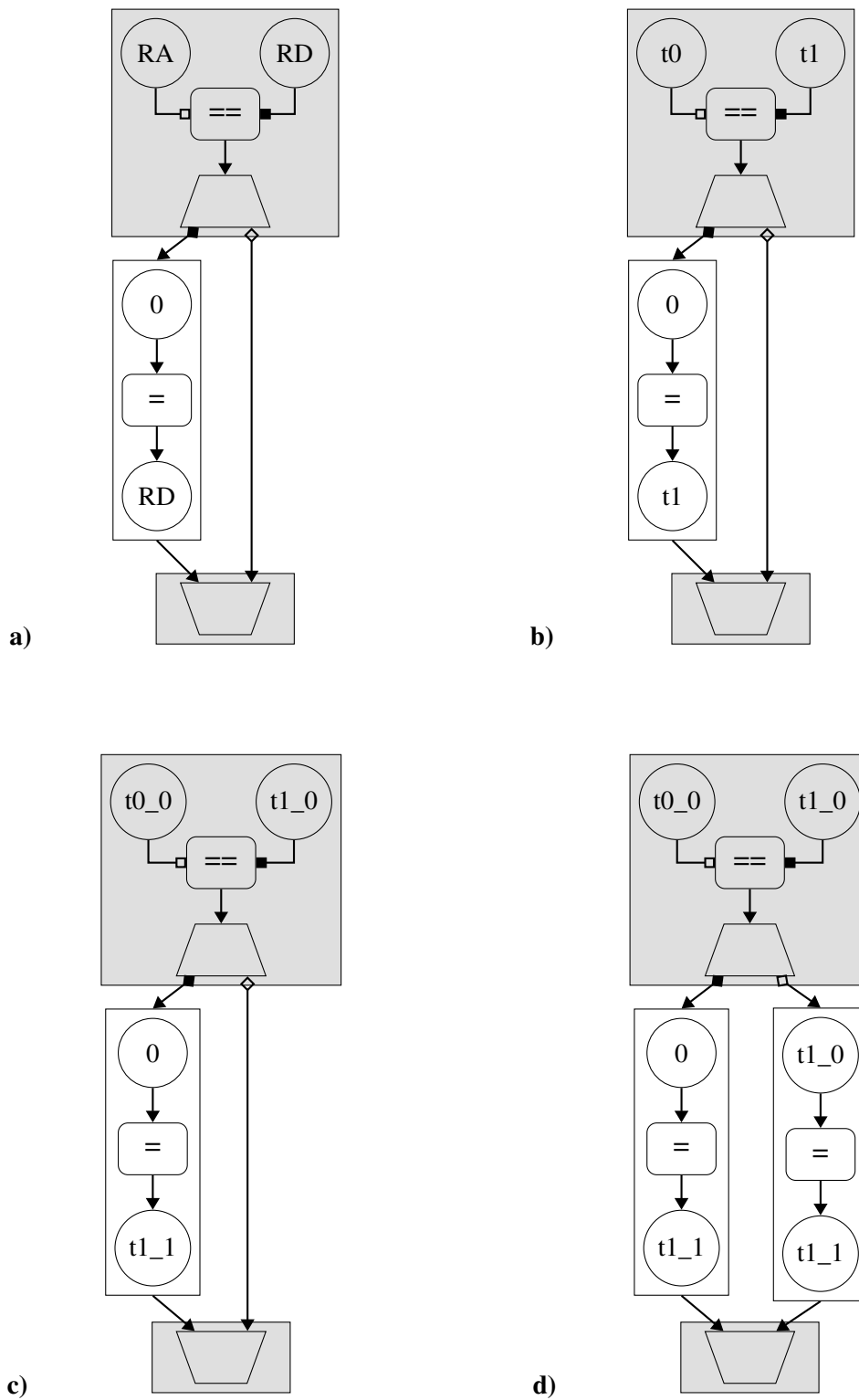


Figure 3.7: Stages of the register name resolving pass on an example CDFG.

In Figure 3.7 the different stages in this pass on an example instruction CDFG are shown (the example CDFG does not represent an actual RISC-V instruction). In a) the CDFG generated by

the tool-chain is shown. In b) the first stage of this pass is performed. This stage replaces all of the generic IL register names with the names of registers in the extracted sequence, in this example RA was register *t0* and RD was register *t1*. In c) the second stage is performed, in which unique identifiers are given to the registers when they are assigned. The register's unique identifier is shown by the value after the underscore in the register name. In this example, *t1* is assigned in the true conditional path so it's unique identifier is incremented from *t1_0* to *t1_1*. In d) the third stage is performed, where if statements are transformed into if-else statements if a register was assigned within the if statement's true path, and in this example *t1* was assigned. This stage is required due to the unique identifier of register *t1* not being consistent after the second stage of this pass. If the true path is taken then the unique identifier of *t1* is *t1_1* and if the false path is taken then the unique identifier of *t1* is *t1_0*. This poses a problem, since in the HDL generation step, if an operation after the example CDFG uses *t1* as an operand, the signal that will be used can not be determined since it can either be the signal that represents *t1_0* or *t1_1*. After the third stage of this pass is performed and the if statement is transformed into an if-else statement, the unique identifier of *t1* is consistent, since if either of the paths are taken the resulting unique identifier of *t1* will always be *t1_1*.

3.4 Module generation

After the CDFG is generated and the optimization passes are applied, the HDL generation step can be performed. Currently the tool-chain only generates fully combinational modules, and sequences that contain division operations or floating point operations are not considered. This is due to the fact that division operations and floating point operations have an extremely high resource cost in FPGAs, and more than likely the generated module would be extremely slow in comparison to dedicated processing units that modern CPUs generally contain, such as the ability to perform division in a single cycle and dedicated Floating Point Units. Sequences that contain memory accesses are also not used to generate modules, due to the fact that multiple concurrent read and write operations can happen if the sequence contains multiple memory accesses. This would require the development of a complex memory interface, which is outside the scope of this thesis.

3.4.1 Hardware Description Language generation

The generation of HDL is done by using the frequent sequence's CDFG to generate either a Verilog Abstract Syntax Tree (AST) or a SystemVerilog AST, with SystemVerilog being targeted in the current implementation. Both of these HDLs can be targeted, without requiring the development of two separate AST generators, due to both of these languages having a similar syntax, with SystemVerilog only implementing some extensions to Verilog.

The first step is the generation of the module's ports and the generation of a single *always_comb* block. To generate the module's ports the inputs and outputs of the CDFG are retrieved and the inputs are transformed to input wire ports with the appropriate bit-widths and the outputs are transformed to output reg ports with the appropriate bit-widths. A single *always_comb* block can

be generated since the generated modules are currently fully combinational, and this block will contain all of the module's statements.

The AST is generated from visiting the frequent sequence's CDFG from inputs to outputs and generating the tree nodes according to what CDFG node was visited. A node can only be visited if all of the nodes before it have been already visited. This is necessary in order to ensure that when generating an operation node, the signals of the operation's operands are already generated and can be retrieved. After visiting an Operation Node an output signal is generated to be used in further operations that require the output of the operation node being generated.

If a node is visited and not all of the nodes before it have been already visited then the CDFG is visited from the next input. This is repeated until the CDFG has been visited from all inputs, and in this manner it can be assured that all possible vertices and edges of the CDFG have been visited.

In Figure 3.8 a simplified view of the SystemVerilog AST generated from the CDFG of the *slt* RISC-V RV32I instruction, shown in Figure 3.5, is shown.

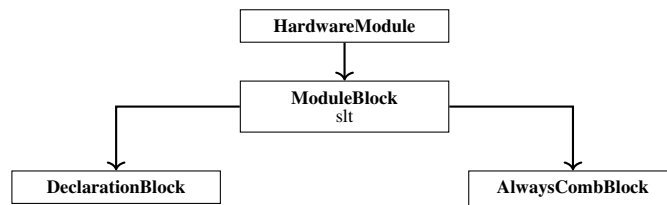


Figure 3.8: Simplified view of the AST of the generated module for the *slt* RISC-V RV32I instruction.

The generated SystemVerilog AST contains a ModuleBlock node that represents the declaration of a module. This node has two children, a DeclarationBlock that contains the module's ports and signals declarations and an AlwaysCombBlock that represents the *always_comb* block that will contain all of the module's logic. An expanded view of the DeclarationBlock branch is shown in Figure 3.9 and an expanded view of the AlwaysCombBlock branch is shown in Figure 3.10.

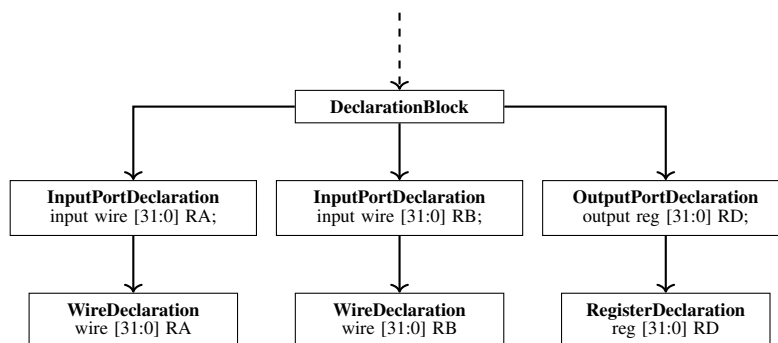


Figure 3.9: Expanded view of the DeclarationBlock branch shown in Figure 3.8.

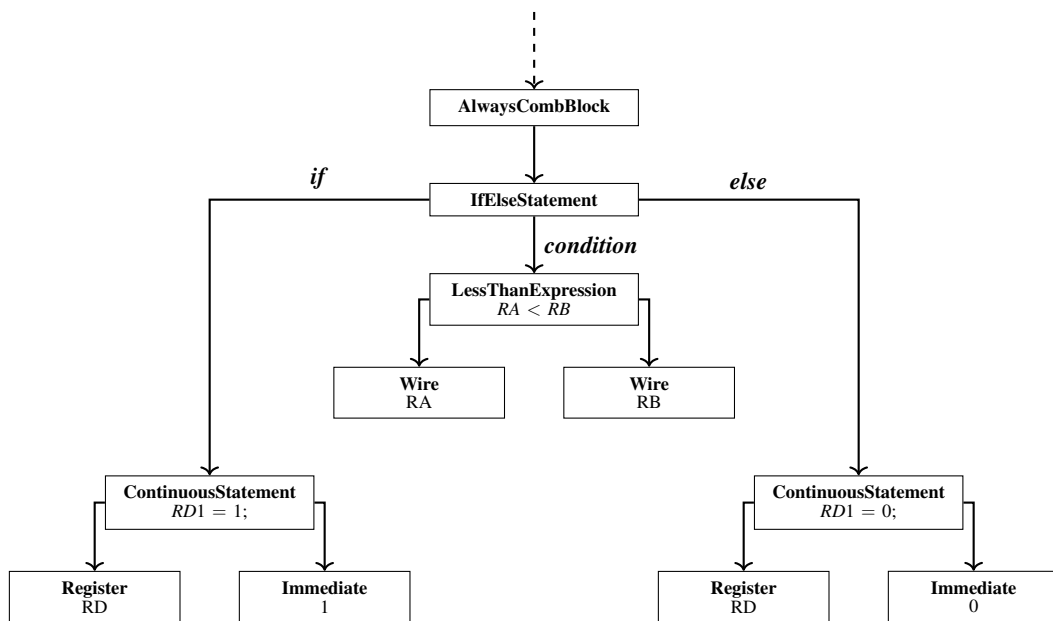


Figure 3.10: Expanded view of the AlwaysCombBlock branch shown in Figure 3.8.

After the SystemVerilog AST is generated the corresponding HDL files are emitted. In Listing 3.5 the emitted SystemVerilog file of the example AST of *slt* instruction is shown.

```

module slt(RA, RB, RD);

input wire [31 : 0] RA;
input wire [31 : 0] RB;
output reg [31 : 0] RD;

always_comb begin : comb_0
    if(RA < RB)
        RD = 32'd1;
    else
        RD = 32'd0;
end

endmodule

```

Listing 3.5: Generated SystemVerilog module of the RISC-V RV32I *slt* instruction.

3.4.2 Synthesis

The modules generated by the tool-chain are all synthesizable, and the tool-chain natively supports two synthesis tools: Xilinx Vivado⁵ and Yosys⁶, or if required by the user the emitted HDL can

⁵Xilinx Vivado website: <https://www.xilinx.com/products/design-tools/vivado.html>

⁶Yosys Github repository: <https://github.com/YosysHQ/yosys>

also be synthesized using an external synthesis tool. If Xilinx Vivado is used, the tool-chain can be configured to target any of Xilinx's FPGA offerings.

In Figure 3.11 the schematic of the synthesized module of the RISC-V *slt* instruction, shown in 3.4.1, is shown. To synthesize this example the Yosys synthesis tool was used, due to this tool having the option of emitting a visual representation of the schematic of the synthesized module.

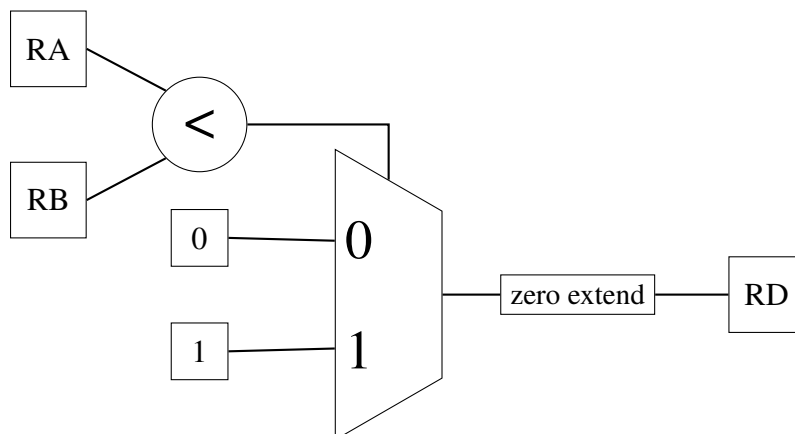


Figure 3.11: Schematic of the synthesized module of the RISC-V RV32I *slt* instruction.

When a module is synthesized using Vivado, the out-of-context option is used. This option ensures that Vivado does not introduce any buffers in the input or output ports of the module. Vivado performs this action due to the fact that it expects the top-level module to be connected to the FPGA pins. The buffers inserted by Vivado invalidate any delay analysis that might be performed, since the inserted buffers have a non-constant large propagation delay (about an order of magnitude larger than the module's maximum propagation delay).

3.5 Validation

The development of HDL modules requires extensive validation in order to ensure that the developed module meets the design requirements. The design requirements can be of functional nature, where the developed module needs to meet exactly the expected behavior, and/or of timing nature, where the different path delays of the module need to be within a certain range defined by the design requirements. In embedded applications other requirements such as power consumption also need to be considered.

In the developed tool-chain the modules are functionally validated and a timing verification is performed in order to determine the maximum frequency of operation of the generated modules. Due to the fact that the generated modules are fully combinational and generally contain a small number of operations, no power consumption measurements were performed due to the fact that this would not give an actual representation of the power consumption of the generated modules, since the power consumption is dependent on factors such as the operating frequency of the FPGA and how the module is implemented in the implementation step of FPGA design.

3.5.1 Functional validation

The functional validation of the generated modules is done using a Monte Carlo process, where a random input data set is generated and the module's outputs are compared to a ground truth data set. This is required because it would not be feasible to validate all possible input combinations. The required testing coverage can be set by defining the number of iterations to perform.

The simulator chosen is the open-source Verilator⁷ simulator, because of its speed in comparison to other offers and its widespread adoption by major manufacturers, such as Intel, AMD and ARM. Due to the fact that Verilator is a cycle accurate simulator and not a timing accurate simulator, this simulator is not appropriate to be used for the timing verification step.

The developed tool chain generates for each generated module a Verilog or a SystemVerilog test-bench (with SystemVerilog currently being targeted), a wrapper Verilator C++ test-bench and generates the required validation data sets.

This validation step is performed in the following way:

1. The tool-chain generates a SystemVerilog test-bench according to the ports of the generated module;
2. The tool-chain generates the required validation data sets by calculating the expected module outputs for each of the generated module inputs values;
3. The tool-chain generates the C++ Verilator test-bench and compiles it;
4. Verilator is used to execute the compiled test-bench. This step goes as follows:
 - (a) Verilator signals the SystemVerilog test-bench to load the next value in the validation data sets and to feed it to the module under test;
 - (b) Verilator then signals the SystemVerilog test-bench to evaluate if the module's outputs matches the output data set value;
 - (c) The SystemVerilog test-bench signals if the validation was successful;
 - (d) These steps are repeated until all of the values in the validation data sets are evaluated.
5. Verilator returns to the tool-chain if the functional validation was successful or not.

3.5.1.1 Automatic test-bench generation

In order to functionally validate the generated modules, appropriate test-benches need to be developed. Traditionally test-benches are hand made for the modules to be validated. In the developed tool-chain a custom SystemVerilog test-bench is automatically generated for each of the generated modules. The generated test-benches do not comply with standard test-bench practices, but this was required in order to perform the functional validation step without requiring the development of a C++ emitter, because Verilator requires the use of C++ test-benches and it does not support all

⁷Verilator website: <https://www.veripool.org/verilator/>

of the SystemVerilog tasks that are generally used for validation such as the `$assert()` task. This approach also has the side effect of the generated test-benches being independent of the simulation tool used, so these test-benches can be used in other simulation tools with only requiring the development of a simple wrapper test-bench.

In Listing 3.6 the generated SystemVerilog test-bench for the module of the RISC-V RV32I `slt` instruction, shown in Section 3.4.1, is shown.

```

module slt_tb(verify, verifyResults);

    // Declarations block: Ports
    input wire verify;
    output reg verifyResults;

    // Declarations block: Wires
    wire [31 : 0] moduleOutputs;

    // Declarations block: Registers
    reg [31 : 0] index;
    reg [63 : 0] inputs [31 : 0];
    reg [31 : 0] outputs [31 : 0];

    initial begin
        index = 32'd0;
        $readmemh("input.mem", inputs);
        $readmemh("output.mem", outputs);
        verifyResults = 1'd0;
    end

    always_ff @ ( posedge verify ) begin : block0
        index <= index + 32'd1;
    end

    always_ff @ ( negedge verify ) begin : block1
        if(moduleOutputs != outputs[index])
            verifyResults <= 1'd0;
        else
            verifyResults <= 1'd1;
    end

    slt slt_2022 (
        .RA(inputs[index][31:0]),
        .RB(inputs[index][63:32]),
        .RD(moduleOutputs[31:0])
    );

endmodule

```

Listing 3.6: Generated SystemVerilog test-bench for the generated module of the `slt` instruction.

The generation of Verilator C++ test-benches is done using a template, since these test-benches are only used as wrapper test-benches for the SystemVerilog test-benches. The developed template is shown in Listing 3.7. During the test-bench generation the `<TESTBENCHNAME>` field is re-

placed with file name of the generated SystemVerilog test-bench, and the <NUMBEROFSAMPLES> is replaced with the size of the generated validation data set.

```

#include <stdlib.h>
#include <iostream>
#include <verilated.h>
#include <verilated_vcd_c.h>

#include "V<TESTBENCHNAME>.h"

#define VALIDATION_SAMPLES <NUMBEROFSAMPLES>
#define VERIFICATION_FAIL 0
#define VERIFICATION_OK 1

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

    V<TESTBENCHNAME> *tb = new V<TESTBENCHNAME>;

    for(int i = VALIDATION_SAMPLES; i > 0; i--){

        tb->verify = 1;

        for(int w = 0; w < 100; w++){
            tb->eval();

        }

        tb->verify = 0;
        tb->eval();

        if(tb->verifyResults == VERIFICATION_FAIL) {
            delete tb;
            std::cout << "FAILED\n";
            exit(EXIT_SUCCESS);
        }
    }

    delete tb;
    std::cout << "PASSED\n";

    exit(EXIT_SUCCESS);
}

```

Listing 3.7: Verilator C++ test-bench template.

3.5.1.2 Automatic validation data generation

The automatic generation of validation data is done using a process similar to some Computer Algebra System implementations. In these implementations, an AST that represents the equation to be solved is generated and is visited sequentially, from inputs to outputs and the values of the intermediate tree nodes are calculated according to the node's inputs. In the developed approach, since a parse tree is generated for each of the sequence's instructions, each of the instruction's outputs are calculated and are used to calculate the next instruction's outputs. The developed

approach also differs due to the instruction's parse tree containing conditional nodes that need to be resolved, since the path taken in visiting the tree will change according to if the node's condition expression is met or not.

The first step is the generation of a random data set for each of the module's inputs. The generated input data set is then used to calculate the outputs of the first instruction in the sequence. The calculated outputs are used to calculate the next instruction's outputs, and this is repeated for the remaining instructions in the sequence. When the last instruction's outputs are calculated, these values are saved as the ground-truth data set to be used for validating the generated module's outputs for the generated random input data set. This process is repeated until the required data set size is generated.

The calculation of each of the instructions outputs is done in the same way as the Computer Algebra System implementations mentioned previously. In the developed approach when a conditional statement is visited, first the statement's condition expression is evaluated, and the true or false path will be visited according to if the condition is met or not.

After the generation of the data sets, these are then emitted as hexadecimal *.mem* files that will be used by the module's test-bench. As shown in Listing 3.6, the generated test-benches require an input data set (*input.mem*) for setting the module's inputs, and a output data set (*output.mem*) that serves as a ground truth for validating if the outputs of the module are correct.

3.5.2 Timing verification

To determine the possible performance gain obtained from using the generated module the maximum propagation delay of the generated module must be determined. Due to the fact that the propagation delay is affected by multiple factors, such as the FPGA's manufacturing process and its architecture, or the way the synthesis step is performed by the FPGA flow used this verification must be done on a per-device basis.

In order to allow a broad set of target devices and due to the fact that manufacture's Integrated Development Environments (IDEs) only support the manufacture's own devices, the developed tool-chain supports two different tools:

- Xilinx Vivado which is a commercial IDE from Xilinx, that fully encompasses all of the steps in FPGA design. This tool is available either for free, with a limited subset of Xilinx's FPGA devices and with some parts of the IDE restricted, or a commercial license can be purchased, that fully enables the IDE. In order to run Vivado without requiring the use of its Graphical User Interface, this tool is run in batch mode, where a TCL script is given, which is used to execute the required tasks, in this case importing the necessary HDL files, synthesizing the design and outputting timing reports. The TCL scripts used are generated using a template so that the target FPGA can be customized. The developed template is shown in Listing 3.8. During the TCL script generation the `<MODULE_NAME>` field is replaced with the name of the generated module to be tested and the `<TARGET_DEVICE>`

field is replaced with the manufacturer reference of the target FPGA device;

```
set outputDir ../reports
set moduleName <MODULE_NAME>

read_verilog [glob ../hdl/*.sv]

set device2test <TARGET_DEVICE>
synth_design -top $moduleName -name
    $moduleName -part $device2test -mode
    out_of_context
report_timing -file $outputDir/
    timing_$device2test.rpt

exit
```

Listing 3.8: Vivado TCL script template.

- Icetime which is part of IceStorm⁸, an open-source tool-chain that aims to reverse engineer Lattice's iCE40 (40nm) family of devices and provide, in conjunction with the Yosys synthesis tool and the nextpnr⁹ place and route tool, a similar flow to that of Lattice's own Diamond IDE¹⁰. The tool-chain's implementation of this tool allows the target device from the supported device family to be chosen as desired. The timing verification results given by this tool are not exactly the same as what would be given by the manufacturer's IDE, but this tool was chosen mainly due to the fact that YosysHQ, the developer of this tool, also provides similar tools for other device families from other manufacturers. These other tools were not implemented because most of them are not currently finalized, but they can be implemented in exactly the same way Icetime was implemented when they are completed, with little to no modifications required for the interface developed for this tool.

The timing reports generated by these tools are then read and parsed by the developed tool-chain so that the maximum propagation delay of the generated module can be determined.

⁸IceStorm Github repository: <https://github.com/YosysHQ/icestorm>

⁹nextpnr Github repository: <https://github.com/YosysHQ/nextpnr>

¹⁰Lattice Diamond IDE website: <https://www.latticesemi.com/LatticeDiamond>

Chapter 4

Results

To determine the validity of the developed tool-chain for the problem stated in Section 1.3, the results produced by the different stages of the tool-chain, shown in Section 3.1, and the performance gains of using the proposed approach will be shown in this section.

4.1 Devices tested

In order to have results that better represent the performance gain that could be obtained with current FPGA architectures, two Xilinx devices manufactured with different feature sizes were chosen, shown in Table 4.1. The devices chosen are the highest performing devices in each of the feature sizes targeted that were available in the free version of Vivado. In all tests done all optimizations available in Vivado that target improving the critical path delay were enabled.

Table 4.1: FPGAs chosen.

Device family	Device	Feature size	DSP slices	Logic Cells (K)
Xilinx Artix Ultrascale+	xcau25p-ffvb676	16nm	1200	308
Xilinx Kintex 7	xc7k70tfbv484-1	28nm	240	65.6

In both of the device families the Digital Signal Processing (DSP) slices contain: a pre-adder, a 25×18 multiplier, an adder and an accumulator. Note that due to the fact that the multipliers in the DSP slices do not match the size of the operands in the RISC-V RV32M standard extension, which would require 32×32 multipliers, each multiplication in the generated module will require the use of multiple DSP slices, which will possibly lead to lower obtained performance gains, due to the distributed placement of multipliers in the FPGA fabric leading to longer signal paths.

In order to obtain more realistic performance gain results, two RISC-V CPUs that are manufactured with the same feature size of the chosen FPGAs were used for comparison. The chosen CPUs are described in Table 4.2.

Table 4.2: CPUs chosen.

CPU	Cores	Feature size	Frequency
Colin Schmidt et al.	8 RV64GC Rocket cores	16nm	1.44GHz
StarFive JH7100	2 Sifive U74 cores + 1 Sifive E24 core	28nm	1GHz

The 16nm CPU shown in Table 4.2 was recently presented in the work of Colin Schmidt et al. [27], and it has the particularity of having been designed using Chisel, mentioned in Section 2.3.2, and its cores generated using the open-source Rocket Chip Generator, mentioned in Section 2.4.2. This CPU will be paired with the chosen Xilinx Artix Ultrascale+ FPGA.

The StarFive JH7100¹ CPU is a commercial offering of StarFive Tech, and it uses commercially available cores designed by Sifive. This CPU has 2 types of processing cores, the U74 core² that is a high performance core and the E24 core³ that is a high efficiency core that mainly targets real time and control applications. Due to its higher performance, for all tests performed the U74 core was chosen. This CPU will be paired with the chosen Xilinx Kintex 7 FPGA.

The performance of the cores of the CPUs shown in Table 4.2 is presented in Table 4.3.

Table 4.3: Clocks Per Instruction of the chosen CPU cores.

Instruction	Clocks Per Instruction	
	RV64GC Rocket	Sifive U74
Multiplication	1	3
Remaining arithmetic operations	1	1

In order to determine the performance gains that the generated HW accelerators provide, the way in which the FPGA and CPU are integrated needs to be chosen. There are multiple levels of integration such as the FPGA being a discrete device and communicating with the CPU using a bus (such as PCIe), the FPGA and the CPU being integrated into an SoC in where the FPGA communicates with the CPU over a bus (such as AXI), or the FPGA being integrated into the CPU's pipeline.

The chosen configuration was to have the FPGA integrated into the CPU's pipeline, since in this manner the communication overheads are minimized, which would drastically reduce the performance gains that can be obtained, and the FPGA can be treated as a computational unit in the CPU's pipeline (such as the CPU's Arithmetic Logic Unit or the Floating Point Unit).

Since the extracted sequences do not contain any branch/jump instructions or memory accesses, the expected number of clocks required to process a sequence of instructions is given by Equation 4.1, where $ClockPerInstruction[i]$ is the Clocks Per Instructions of the i -th instruction in the sequence. This value can be determined, in the case of the CPU cores considered, by using Table 4.3.

¹StarFive JH7100 Github repository: https://github.com/starfive-tech/JH7100_Docs

²Sifive U74 Core Complex Manual: https://sifive.cdn.prismic.io/sifive/ad5577a0-9a00-45c9-a5d0-424a3d586060_u74_core_complex_manual_21G3.pdf

³Sifive E24 Core Complex Manual: https://sifive.cdn.prismic.io/sifive/dc408861-94ce-4d82-a704-caddec98609d_e24_core_complex_manual_21G3.pdf

$$N_{clocks} = \sum_{i=1}^{N_{instructions}} ClocksPerInstruction[i] \quad (4.1)$$

In CPU cores that exhibit single cycle delays in all instructions, such as in the RV64GC Rocket core, the number of clocks required to process a sequence of instructions is given by Equation 4.2.

$$N_{clocks} = N_{instructions} \quad (4.2)$$

Since the clock frequencies of the CPUs are known, the amount of time necessary to process a sequence of instructions is given by Equation 4.3.

$$T_{delayCPU} = \frac{1}{f_{CPU}} \times N_{clocks} \quad (4.3)$$

The expected speed-up of the extracted sequence that can be obtained by the generated module, is given by Equation 4.4, where $\max(T_{delay_{module}})$ is the maximum propagation delay of the generated module given by the synthesis tool used.

$$SpeedUp = \frac{T_{delayCPU}}{\max(T_{delay_{module}})} \quad (4.4)$$

4.2 Benchmarks

To have a more broad set of results, multiple sets of benchmarks were chosen, that target different applications:

- **Livermore Loops:** classical benchmark set that targets parallel computers. This benchmark was presented in the work of Francis H. McMahon [28]. Originally this benchmark was written in FORTRAN, but since it's release it has been ported to numerous languages⁴.
- **PolyBench/C:** set of benchmarks that targets validating optimizations performed by compilers. It was developed in 2010 by Louis-Noel Pochet at the Ohio State University⁵.
- **Custom developed benchmark set:** custom developed benchmark set that aims to better demonstrate the limitations and merits of the developed tool-chain and to better show the impact that factors such as the number of instructions in a frequent sequence or the sequence's inherent parallelism has on the performance gains of the proposed approach.

The kernels of the benchmark sets used are shown in Table 4.4.

⁴Livermore Loops C kernel: <https://www.netlib.org/benchmark/livemorec>

⁵PolyBench benchmark set website : <https://web.cse.ohio-state.edu/pouchet.2/software/polybench/>

Table 4.4: Benchmarks used.

Set	Benchmark	Description
Livermore Loops	hydro2d	2-D explicit hydrodynamics fragment
	matmul	Matrices multiplication
	pic1d	1-D particle in a cell
	cholesky	Incomplete Cholesky conjugate gradient
	statefrag	Equation of state fragment
	hydro2dimpl	2-D implicit hydrodynamics fragment
PolyBench/C	gemm	Matrix-multiply $C=\alpha.A.B+\beta.C$
	nussinov	Nucleic acid structure prediction
	syr2k	Symmetric rank-2k operations
	ludcmp	LU decomposition
	lu	LU decomposition
	Custom	conv2x2
add8par		Parallel addition of 8 integers
add8seq		Sequential addition of 8 integers
avg16		Average of 16 integers calculator

The custom set of benchmarks was developed in order to isolate and try to determine which factors have an impact on the performance gains that can be obtained by using the proposed approach. The developed kernels were written in C, and the `register` keyword was used in all declared variables, in order to hint the compiler to not use memory operations.

The custom benchmark set contains the following benchmarks:

- **avg16 - Average of 16 integers calculator:** This benchmark contains 15 additions and a bit shift, and it aims to show the acceleration provided by the developed tool-chain on sequences that contain a large amount of instructions, that are all simple operations. The kernel of this benchmark is shown in Listing 4.1.

```
int avg16(int v0, int v1, int v2, int v3, int v4, int v5, int v6, int v7, int v8,
int v9, int v10, int v11, int v12, int v13, int v14, int v15){

    register int v_quad0 = v0 + v1 + v2 + v3;
    register int v_quad1 = v4 + v5 + v6 + v7;
    register int v_quad2 = v8 + v9 + v10 + v11;
    register int v_quad3 = v12 + v13 + v14 + v15;

    register int v_final = v_quad0 + v_quad1 + v_quad2 + v_quad3;

    return v_final >> 4;
}
```

Listing 4.1: avg16 benchmark kernel.

- **conv2x2 - Parallel convolution of two 2x2 matrices:** This benchmark contains 4 parallel multiplications and 3 additions, and it aims to show the limitations of the developed tool-chain, in sequences that are both inherently parallel and contain multiple complex operations, in this case multiplications. The kernel of this benchmark is shown in Listing 4.2.

```
int conv2x2(int a0, int a1, int a2, int a3, int b0, int b1, int b2, int b3){

    register int mul0 = a0 * b0;
    register int mul1 = a1 * b1;
    register int mul2 = a2 * b2;
    register int mul3 = a3 * b3;

    register int add0 = mul0 + mul1;
    register int add1 = mul2 + mul3;
    register int add_final = add0 + add1;

    return add_final;
}
```

Listing 4.2: conv2x2 benchmark kernel.

- **add8par - Parallel addition of 8 integers:** This benchmark contains 7 additions in a binary tree structure, and it is meant to be compared with the add8seq benchmark in order to demonstrate the sensitivity of the developed tool-chain to the inherent parallelism of the extracted sequence. This benchmark is not a realistic representation of actual binaries. The kernel of this benchmark is shown in Listing 4.3.

```
int sum8_par(int v0, int v1, int v2, int v3, int v4, int v5, int v6, int v7){

    register int add0 = v0 + v1;
    register int add1 = v2 + v3;
    register int add2 = v4 + v5;
    register int add3 = v6 + v7;

    register int add_0_1 = add0 + add1;
    register int add_2_3 = add2 + add3;

    return add_0_1 + add_2_3;
}
```

Listing 4.3: add8par benchmark kernel.

- **add8seq - Sequential addition of 8 integers:** This benchmark contains 7 sequential additions, and it is meant to be compared with the add8par benchmark in order to demonstrate the sensitivity of the developed tool-chain to the inherent parallelism of the extracted sequence. This benchmark is also meant to demonstrate the optimizations performed by compilers, mainly the serialization of operations. This is meant to be a more realistic representation of binaries. The kernel of this benchmark is shown in Listing 4.4.

```
int sum8_seq(int v0, int v1, int v2, int v3, int v4, int v5, int v6, int v7) {
    return v0 + v1 + v2 + v3 + v4 + v5 + v6 + v7;
}
```

Listing 4.4: add8seq benchmark kernel.

All of the benchmark sets used were compiled using the RISC-V GNU Compiler Toolchain⁶, which is a RISC-V C and C++ cross-compiler. This tool-chain offers the same tools as the regular Linux GNU tool-chain, such as GCC, G++, GDB and objdump.

4.3 Results

In this section the results obtained in each of the stages of the developed tool-chain, mentioned in Section 3.1, are shown.

4.3.1 Stage 1

In this section the results obtained from Stage 1 of the developed tool-chain, in which frequent sequences are extracted from a compiled binary and translated to the IL of the SPeCS Binary Translation Framework, and parse trees are generated from this representation.

In Table 4.5 the sequences extracted by the tool-chain for each of the benchmarks presented in Section 4.2 is shown.

Table 4.5: Extracted sequences per benchmark.

Set	Benchmark	Sequence size		Number of sequences
		Average	Max	
Livermore Loops	hydro2d	2.18	3	11
	matmul	2.2	3	5
	pic1d	2.2	3	5
	cholesky	2.25	3	5
	statefrag	2.25	3	4
	hydro2dimpl	2.3	3	6
PolyBench/C	gemm	2.5	3	11
	nussinov	2.67	4	9
	syr2k	2.86	4	10
	ludcmp	2.64	4	19
	lu	2.46	4	15
Custom	conv2x2	3.5	7	6
	add8par	3.5	7	6
	add8seq	3.5	7	6
	avg16	8.5	17	16

In Table 4.6 the average number of sequences and their size per extracted per benchmark is shown.

⁶RISC-V GNU Compiler Toolchain Github repository: <https://github.com/riscv-collab/riscv-gnu-toolchain>

Table 4.6: Extracted sequences averages per benchmark.

Set	Average sequence size	Average number of sequences
Livermore Loops	3	6
PolyBench/C	3.8	12.8
Custom	9.5	8.5

In Listing 4.5 an example of a sequence extracted from the hydro2d benchmark of the Livermore Loops benchmark set is shown.

```

addi    s10, a0, 0x0
addi    a0, s11, 0x0
add     a1, a5, a4

```

Listing 4.5: Extracted sequence from the hydro2d benchmark.

In Listing 4.6 the largest extracted sequence detected in the benchmark sets used is shown. This sequence was detected in the avg16 benchmark, that is part of the custom benchmark set. This sequence contains 17 instructions, that are mostly addition operations.

```

add     a5, t4, t3
add     a5, t1, a5
add     s4, a0, a5
add     a5, a1, a2
add     a5, a3, a5
add     s3, a4, a5
add     a5, s2, s1
add     a5, t2, a5
add     s2, t0, a5
add     a5, t6, t5
add     a5, a7, a5
add     s1, a6, a5
add     a5, s4, s3
add     a5, s2, a5
add     s1, s1, a5
srai    a5, s1, 0x4
addi    a0, a5, 0x0

```

Listing 4.6: Largest extracted sequence in the avg16 benchmark.

4.3.2 Stage 2

In this stage of the tool-chain, the extracted frequent sequences parse trees are transformed into CDFGs and HDL modules, currently SystemVerilog, are generated. In order to demonstrate the generated CDFGs and SystemVerilog modules, the extracted sequences presented in Section 4.3.1 will be used.

In Figure 4.1 the CDFG generated from the frequent sequence presented in Listing 4.5 is shown. Since the extracted sequence is composed of 3 independent additions, the resulting CDFG will contain a single Data Flow Subgraph.

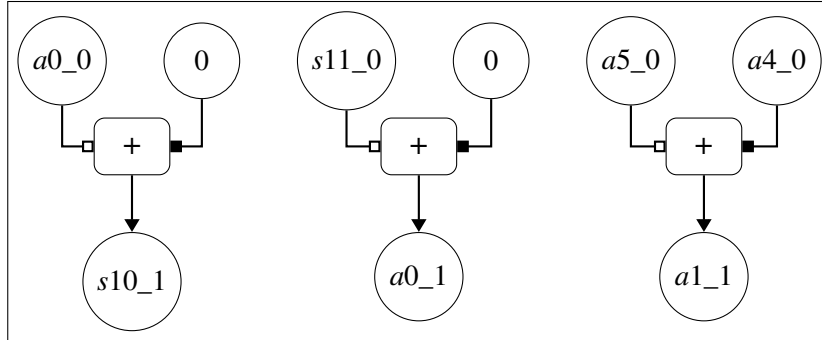


Figure 4.1: Generated CDFG of the extracted sequence in the hydro2d benchmark presented in Listing 4.5.

In Listing 4.7 the SystemVerilog module generated from the CDFG shown in Figure 4.1 is presented. The generated module contains two additions with a zero immediate value, that the synthesis tool used, in this case Xilinx Vivado, will replace with assignments, as per example $s10_1 = a0_0 + 32'd0$ will be transformed to $s10_1 = a0_0$.

```

module Segment_2848929098(a0_0, a4_0, a5_0, s11_0, a0_1, a1_1, s10_1);

    // Declarations block: Ports
    input wire [31 : 0] a0_0;
    input wire [31 : 0] a4_0;
    input wire [31 : 0] a5_0;
    input wire [31 : 0] s11_0;
    output reg [31 : 0] a0_1;
    output reg [31 : 0] a1_1;
    output reg [31 : 0] s10_1;

    always_comb begin : comb_0
        s10_1 = a0_0 + 32'd0;
        a0_1 = s11_0 + 32'd0;
        a1_1 = a5_0 + a4_0;
    end
endmodule //Segment_2848929098

```

Listing 4.7: Generated SystemVerilog module of the extracted sequence in the hydro2d benchmark presented in Listing 4.5.

In Figure 4.2 the CDFG generated from the frequent sequence presented in Listing 4.6 is shown. Since the extracted sequence does not contain any instructions that alter the control flow of the kernel, the generated CDFG contains a single Data Flow Subgraph with all of the operations inside it. In this representation the serialization optimization performed by the compiler used

can be seen, since each of the sums of 4 values in the benchmark was compiled as 3 sequential operations. This optimization is done by the compiler in order to reduce the number of registers used.

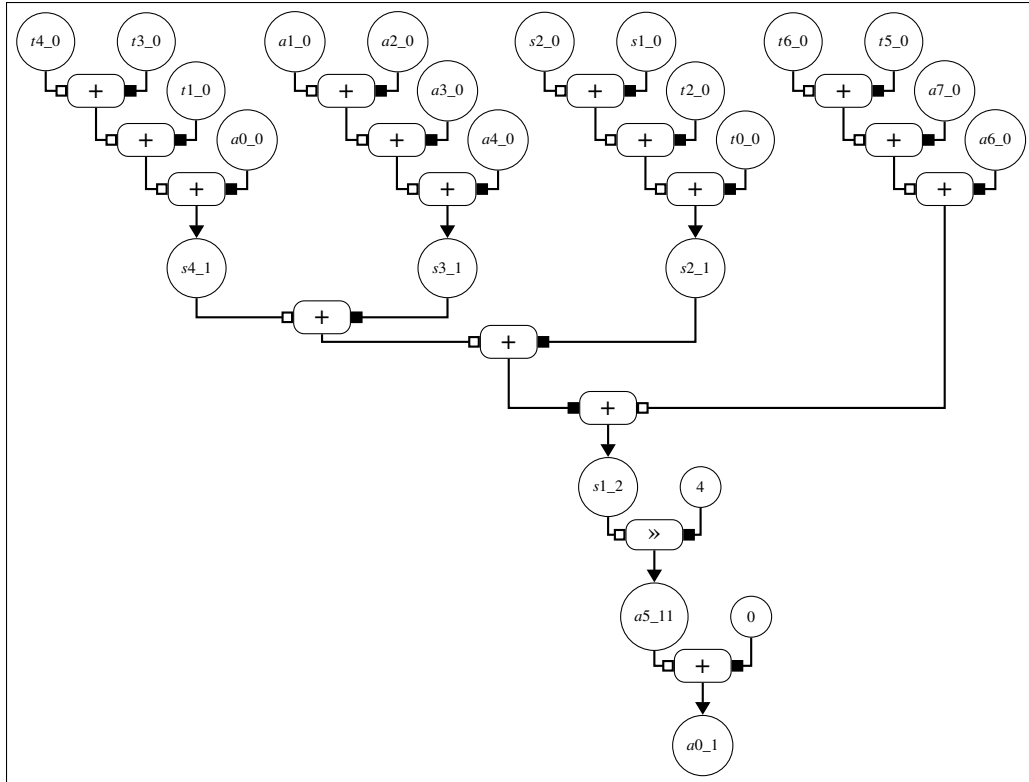


Figure 4.2: Generated CDFG of the extracted sequence in the avg16 benchmark presented in Listing 4.6.

In Listing 4.8 the generated SystemVerilog module of sequence's CDFG shown in Figure 4.2 is shown.

```

module segment_avg16_size_17(a0_0, a1_0, a2_0, a3_0, a4_0, a6_0, a7_0, s1_0, s2_0,
    t0_0, t1_0, t2_0, t3_0, t4_0, t5_0, t6_0, a0_1, a5_11, s1_2, s2_1, s3_1, s4_1);

    // Declarations block: Ports
    input wire [31 : 0] a0_0;
    input wire [31 : 0] a1_0;
    input wire [31 : 0] a2_0;
    input wire [31 : 0] a3_0;
    input wire [31 : 0] a4_0;
    input wire [31 : 0] a6_0;
    input wire [31 : 0] a7_0;
    input wire [31 : 0] s1_0;
    input wire [31 : 0] s2_0;
    input wire [31 : 0] t0_0;
    input wire [31 : 0] t1_0;
    input wire [31 : 0] t2_0;
    input wire [31 : 0] t3_0;
    input wire [31 : 0] t4_0;
    input wire [31 : 0] t5_0;
    input wire [31 : 0] t6_0;
    output reg [31 : 0] a0_1;
    output reg [31 : 0] a5_11;
    output reg [31 : 0] s1_2;
    output reg [31 : 0] s2_1;
    output reg [31 : 0] s3_1;
    output reg [31 : 0] s4_1;

    // Declarations block: Wires
    wire [31 : 0] add_0;
    wire [31 : 0] add_1;
    wire [31 : 0] add_2;
    wire [31 : 0] add_3;
    wire [31 : 0] add_4;
    wire [31 : 0] add_5;
    wire [31 : 0] add_6;
    wire [31 : 0] add_7;
    wire [31 : 0] add_8;
    wire [31 : 0] add_9;
    wire [31 : 0] add_10;

    always_comb begin : comb_0
        add_0 = t4_0 + t3_0;
        add_1 = t1_0 + add_0;
        s4_1 = a0_0 + add_1;
        add_2 = a1_0 + a2_0;
        add_3 = a3_0 + add_2;
        s3_1 = a4_0 + add_3;
        add_4 = s2_0 + s1_0;
        add_5 = t2_0 + add_4;
        s2_1 = t0_0 + add_5;
        add_6 = t6_0 + t5_0;
        add_7 = a7_0 + add_6;
        add_8 = a6_0 + add_7;
        add_9 = s4_1 + s3_1;
        add_10 = s2_1 + add_9;
        s1_2 = add_8 + add_10;
        a5_11 = s1_2 >> 32'd4;
        a0_1 = a5_11 + 32'd0;
    end

```

Listing 4.8: Generated SystemVerilog module of the largest extracted sequence in the avg16 benchmark.

4.3.3 Stage 3

In this stage the generated modules are functionally validated, and if this validation step is successful, the modules are synthesized and a timing analysis is performed on the synthesized modules.

In this section the speed-up results obtained in each of the benchmark sets used and the factors that impact the performance gain that can be obtained by the proposed approach will be analyzed.

4.3.3.1 Livermore Loops benchmark set

The performance gains obtained in each of the benchmarks in the Livermore Loops benchmark set is shown in Figure 4.3. The error bars in the graph show the maximum and minimum speed-up obtained in that particular benchmark.

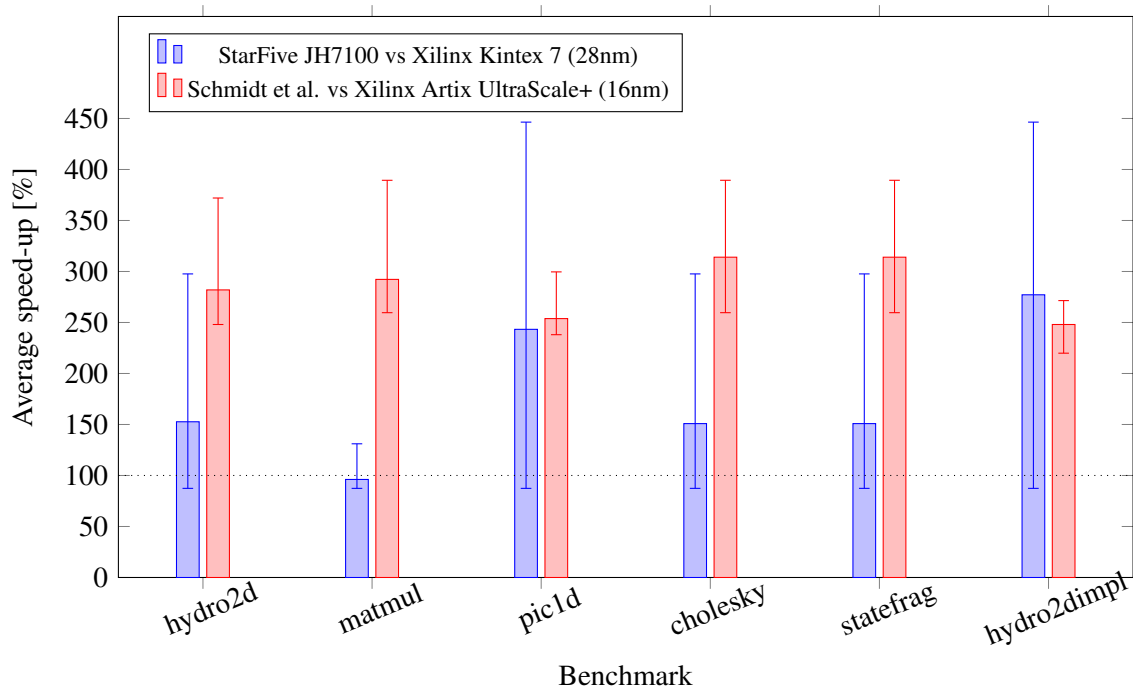


Figure 4.3: Average speed up obtained per extracted sequence of a benchmark in the Livermore Loops benchmark set.

In this benchmark set, most of the extracted frequent sequences were of either two sequential additions or of an addition followed by a multiplication. The 28nm devices on average benefited less from this approach than the 16nm devices. This might be due to the 16nm FPGA having a better architecture and its smaller feature size decreases the maximum path delay more than the increase in CPU frequency from the 28nm CPU to the 16nm CPU. Overall all benchmarks showed a performance gain of more than 100%, with the 28nm devices showing an average of 175% and the 16nm devices an average speed-up of 275%.

In Table 4.7 the resource utilization and maximum operating frequency of the synthesized generated modules of each extracted sequence of the Livermore Loops benchmark set is shown. The

modules that have a N/A maximum operating frequency, the synthesis tool synthesized modules that have an infinite maximum operating frequency, since the extracted sequences contained only *mv* pseudo operations, that are synthesized as direct connections between the input and output ports of the module.

Table 4.7: Resource utilization and maximum operating frequency of each synthesized generated module from the Livermore Loops benchmark set.

Benchmark	Sequence size	Resources		Maximum frequency(MHz)	
		Logic Cells	DSP slices	16nm	28nm
hydro2d	2	2	0	1869	437
		64	0	1786	437
		0	0	N/A	1488
		2	0	1869	437
		0	0	N/A	1488
		32	0	1786	437
		0	0	N/A	1488
		32	0	1786	437
	32	0	1786	437	
	3	32	0	1786	437
32		0	1786	437	
matmul	2	1	0	2110	437
		6	0	1869	437
		1	0	1869	437
		2	0	1869	437
	3	2	0	1869	437
pic1d	2	32	0	1786	437
		0	0	N/A	1488
		2	0	1869	437
		0	0	N/A	1488
	3	0	0	N/A	1488
statefrag	2	0	0	N/A	1488
		1	0	1869	437
		1	0	2110	437
	3	2	0	1869	437
hydro2dimpl	2	64	0	1786	437
		0	0	N/A	1488
		0	0	N/A	1488
		33	0	1786	437
	3	0	0	N/A	1488
cholesky	2	0	0	N/A	1488
		1	0	2110	437
		1	0	1869	437
	3	2	0	1869	437

Legend: N/A - Not Applicable

4.3.3.2 PolyBench/C benchmark set

The performance gains obtained in each of the benchmarks in the PolyBench/C benchmark set is shown in Figure 4.4. The error bars in the graph show the maximum and minimum speed-up obtained in that particular benchmark.

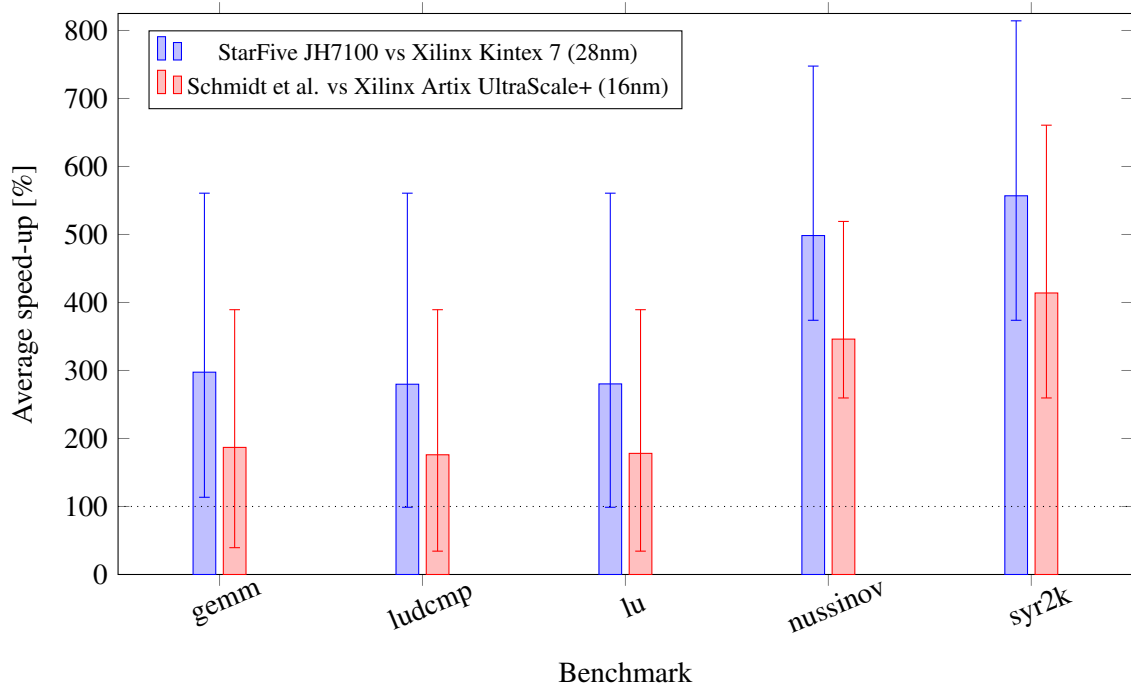


Figure 4.4: Average speed up obtained per extracted sequence of a benchmark in the PolyBench/C benchmark set.

The composition of the extracted sequences in this benchmark set did not follow a particular pattern. One noteworthy aspect that is not shown in the figure is that some of the benchmarks showed an infinite speed-up. This is due to the fact that those particular frequent sequences were fully composed of *addi* instruction with the immediate operand set to zero. This is equivalent to sequence with only *mv* pseudo-instructions, so those frequent sequences are performing a register mapping function, and in Xilinx Vivado the additions of a register with 0 are replaced with a direct connection between the input and output signals. In a real application the speed-up would not be infinite, but it would be limited by the clock speed of the processor, since if it wrote all values in parallel to the register file, this would take a single clock cycle. All of the benchmarks performed showed an average performance gain of over 150%, with the 28nm devices showing an average speed-up of 350% and the 16nm devices an average speed-up of 250%. In the *nussinov* and *syr2k* benchmarks the 28nm devices showed a maximum speed up of over 700%, when the extracted sequence contained 4 instructions.

In Table 4.8 the resource utilization and maximum operating frequency of the synthesized generated modules of each extracted sequence of the Livermore Loops benchmark set is shown. The modules that have a N/A maximum operating frequency, the synthesis tool synthesized modules that have an infinite maximum operating frequency, since the extracted sequences contained only *mv* pseudo operations, that are synthesized as direct connections between the input and output ports of the module.

Table 4.8: Resource utilization and maximum operating frequency of each synthesized generated module from the PolyBench/C benchmark set.

Benchmark	Sequence size	Resources		Maximum frequency(MHz)	
		Logic Cells	DSP slices	16nm	28nm
gemm	2	2	0	1869	437
		16	3	284	136
	3	5	0	1869	437
		17	3	284	136
ludcmp	2	17	3	284	136
		2	0	2004	448
		64	0	1786	437
		0	0	N/A	1488
		3	0	1869	437
		0	0	N/A	1488
		47	3	247	122
		2	0	1869	437
	3	1	0	1869	437
		3	0	1869	437
		0	0	N/A	1488
		18	3	284	136
		49	3	247	122
		17	3	284	136
		6	0	1869	437
		50	3	247	122
	4	0	0	N/A	1488
		49	3	247	122
2		0	1869	437	
17		3	284	136	
1		0	1869	437	
0		0	N/A	1488	
3		0	1869	437	
2		0	2004	448	
lu	2	0	0	N/A	1488
		3	0	1869	437
		47	3	247	122
		1	0	1869	437
		0	0	N/A	1488
		2	0	2004	448
	3	0	0	N/A	1488
		3	0	1869	437
		18	3	284	136
		49	3	247	122
		50	3	247	122
		0	0	N/A	1488
nussinov	2	1	0	1869	437
		5	0	1869	437
		0	0	N/A	1488
		2	0	1869	437
	3	3	0	1869	437
		3	0	1869	437
		4	0	1869	437
		4	0	1869	437
syr2k	2	2	0	1869	437
		0	0	N/A	1488
		1	0	1869	437
		4	0	2004	448
	3	0	0	N/A	1488
		3	0	1869	437
		5	0	1869	437
		0	0	N/A	1488
	4	0	0	N/A	1488
		0	0	N/A	1488
5	0	0	N/A	1488	

Legend: N/A - Not Applicable

4.3.3.3 Custom benchmark set

The performance gains obtained in each of the benchmarks in the custom benchmark set is shown in Figure 4.5. The error bars in the graph show the maximum and minimum speed-up obtained in that particular benchmark.

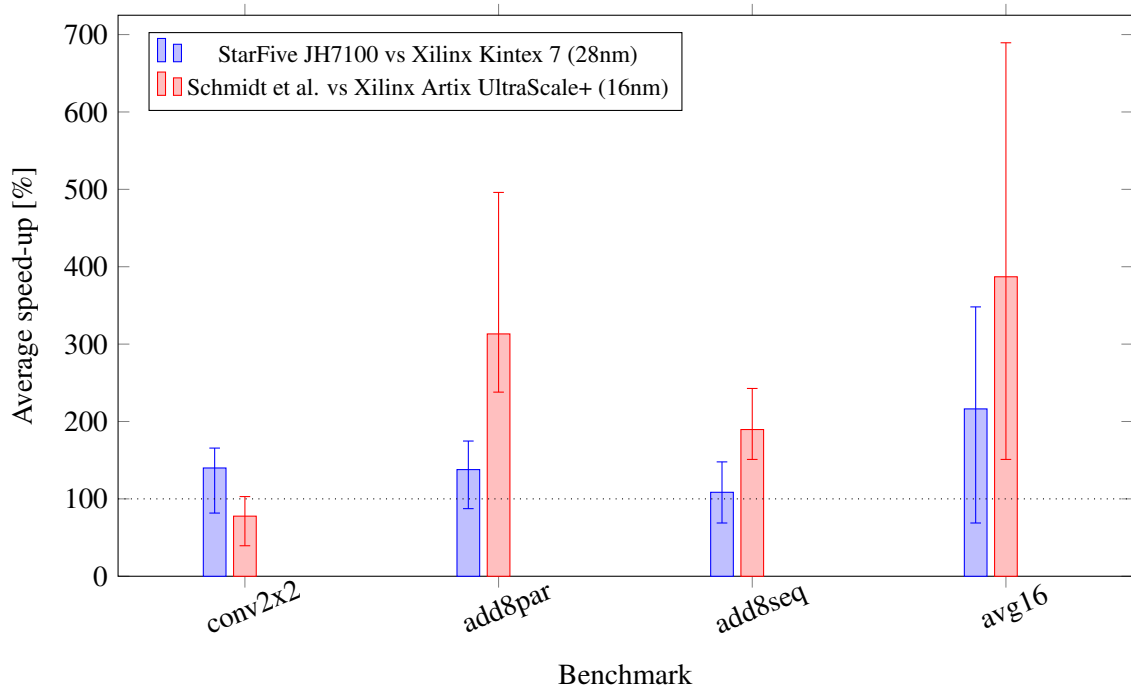


Figure 4.5: Average speed up obtained per extracted sequence of a benchmark in custom benchmark set.

In all benchmarks except for the conv2x2 benchmark, the 16nm devices obtained a larger performance gain than the 28nm devices. The factors that contribute to the results obtained in this benchmark set are explained in detail in Section 4.3.4.

In Table 4.9 the resource utilization and maximum operating frequency of the synthesized generated modules of each extracted sequence of the Livermore Loops benchmark set is shown.

Table 4.9: Resource utilization and maximum operating frequency of each synthesized generated module from the custom benchmark set.

Benchmark	Sequence size	Resources		Maximum frequency(MHz)	
		Logic Cells	DSP slices	16nm	28nm
conv2x2	2	1	0	2004	448
	3	1	0	2004	448
	4	1	0	2004	448
	5	1	0	2004	448
	6	1	0	2004	448
	7	1	0	2004	448
	8	1	0	2004	448
	9	1	0	2004	448
	10	46	9	284	136
	11	61	12	284	136
	12	93	12	247	118
	13	125	12	247	118
	14	125	12	205	102
	15	125	12	205	102
	16	125	12	205	102
	add8par	2	64	0	1786
3		96	0	1786	437
4		128	0	1786	437
5		128	0	687	264
6		128	0	687	264
7		160	0	489	205
add8seq	2	32	0	1087	344
	3	92	0	903	316
	4	64	0	551	232
	5	124	0	499	218
	6	96	0	551	230
	7	156	0	499	211
avg16	2	32	0	1087	344
	3	64	0	663	264
	4	124	0	903	316
	5	124	0	903	316
	6	156	0	663	264
	7	216	0	903	316
	8	216	0	903	316
	9	250	0	680	264
	10	308	0	903	316
	11	308	0	903	316
	12	343	0	680	264
	13	400	0	590	235
	14	400	0	492	212
	15	372	0	266	132
	16	372	0	345	166
	17	372	0	345	166

4.3.4 Impact of sequence characteristics on the performance gain obtained

To determine the factors that contribute to the performance gain that can be obtained by the proposed approach, the custom benchmark set will be considered.

4.3.4.1 Sequence size

To determine if the number of instructions in a sequence has a measurable impact on the performance gain obtained from the developed tool-chain, the speed-up obtained per number of instructions in a sequence was calculated for the avg16 benchmark, since this benchmark showed the largest extracted segment size of all benchmarks used. In Figure 4.6 the performance gain per number of instructions is shown.

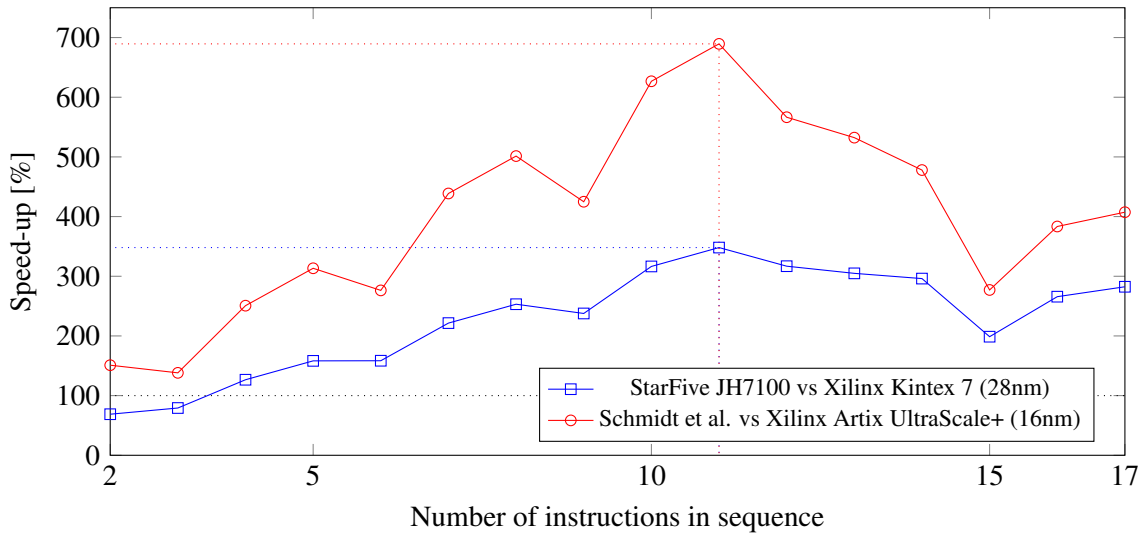


Figure 4.6: Speed-up per number of instructions in a frequent sequence in the avg16 custom benchmark.

The avg16 benchmark showed a large performance gain from using this approach until the sequence contained 11 instructions. The 16nm devices showed a maximum performance gain of almost 700%, and the 28nm devices a maximum performance gain of about 350%. The decrease in the performance gain obtained in larger sequences is due to the synthesis tool used, Xilinx Vivado, starting to utilize more complex primitives, such as CARRY8 primitives in the case of the 16nm device, that incur a larger propagation delay. This leads to a decrease in the performance gains obtained.

The results obtained indicate that larger extracted sequences tend to benefit from the proposed approach.

4.3.4.2 Inherent parallelism of a sequence

In order to determine the impact that the inherent parallelism of the extracted sequences has on the performance gains, the add8par and add8seq benchmarks were considered since these benchmarks compute the same algorithm, in this case the addition of 8 integers. The performance gain per number of instructions in an extracted sequence is shown in Figure 4.7.

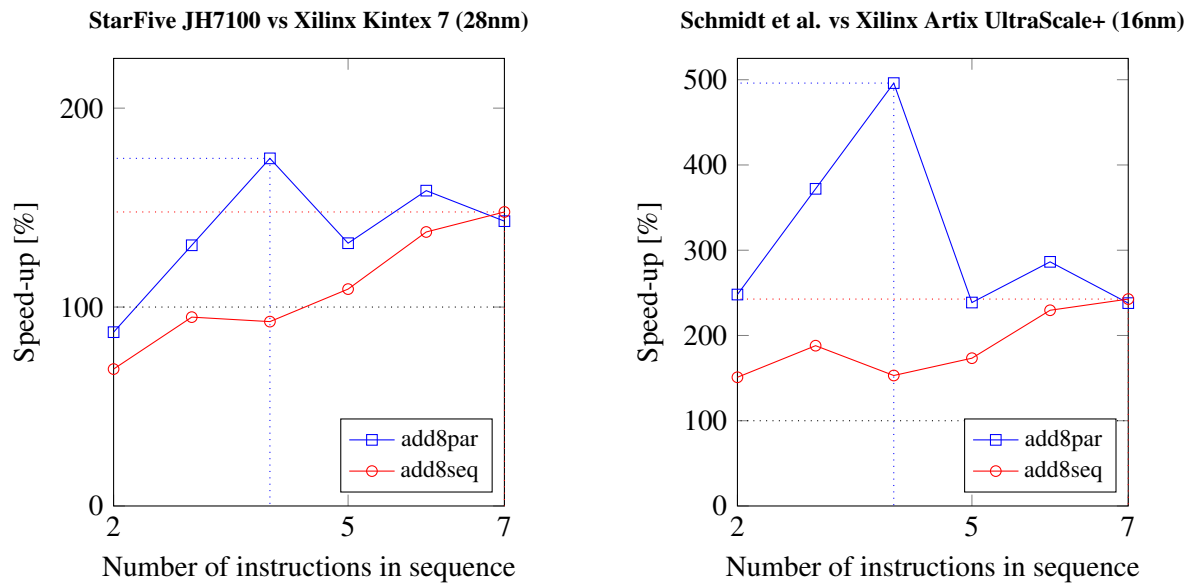


Figure 4.7: Speed-up per number of instructions in a frequent sequence in the add8par and add8seq custom benchmarks.

In almost all sequence sizes the add8par benchmark showed measurably larger performance gains than the add8seq benchmark. In the add8par benchmark, the 16nm devices showed a maximum speed-up of almost 500% and the 28nm devices showed a maximum speed-up of about 185%. In the add8seq benchmark, the 16nm devices showed a maximum speed-up of about 250% and the 28nm devices showed a maximum speed-up of about 150%. The results obtained indicate that sequences that are more inherently parallel tend to benefit more from the proposed approach, due to the parallelism acceleration that FPGAs provide.

Both benchmarks showed about the same performance gain in the largest sequence size, due to, as mentioned in Section 4.3.4.1, the synthesis tool used starting to utilize more complex primitives which negatively impact the propagation delay of the module.

4.3.4.3 Complexity of operations in a sequence

To determine the impact that the complexity of operations in a frequent sequence has on the performance gains that can be obtained, the conv2x2 benchmark was considered, since half of the operations in this benchmark are multiplications, that as mentioned in Section 3.1, may have a measurable impact on the performance gain that can be obtained. The speed-up obtained per number of instructions in an extracted sequence in this benchmark is shown in Figure 4.8.

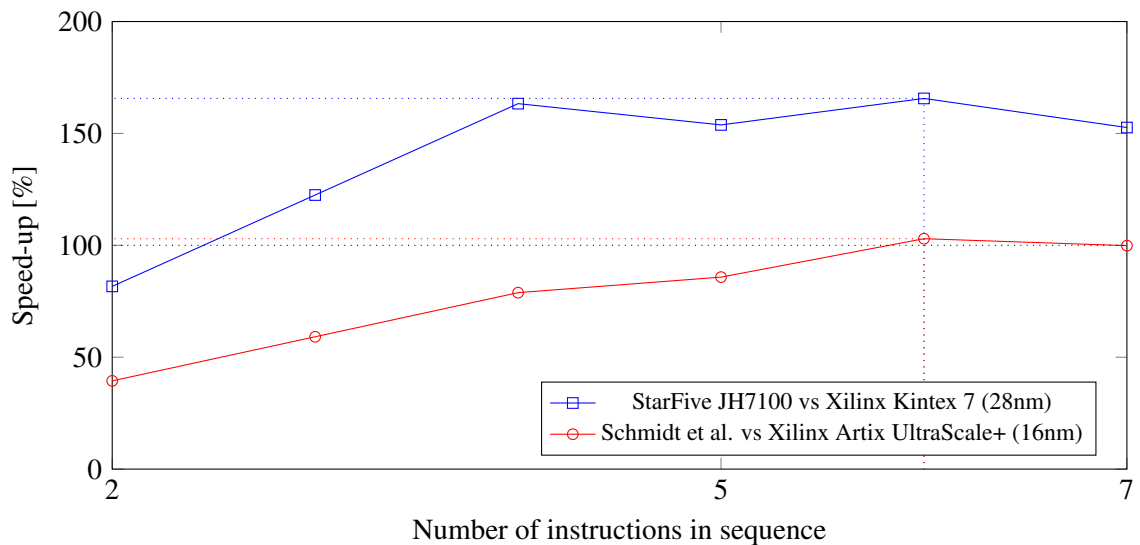


Figure 4.8: Speed-up per number of instructions in a frequent sequence of the conv2x2 benchmarks.

The conv2x2 benchmark stopped showing an improvement in the performance gain obtained with the increase of the number of instructions in the sequence after 4 instructions due to after this amount of instructions in a sequence multiplication operations will always be present in the sequence. In this benchmark the 28nm devices showed a greater benefit from being accelerated using this approach due to the fact that the U74 core performs multiplications in 3 cycles, while the 16nm devices showed no performance gain from using this approach.

The impact of the presence of multiplications can also be seen if two sequences of similar sequences that only differ in the presence of a multiplication are compared. In the gemm benchmark of the PolyBench/C benchmark set, 2 sequences of 3 instructions were detected. The detected sequences are shown in Listing 4.9.

addi a7, a7, 0x1	addi a5, a5, 0x4
addi t4, t4, 0x64	addi a3, a3, 0x1
addi a6, a6, 0x4	mul a4, a4, t0

Listing 4.9: Extracted sequences of 3 instructions from the gemm benchmark of the PolyBench/C benchmark set

Both the sequences contain 3 independent operations, but the second sequence contains a multiplication operation. In Table 4.10 the maximum operating frequency of the generated modules for each of the sequences, presented in Listing 4.9, is shown.

Table 4.10: Decrease in maximum operating frequency with the inclusion of a multiplication.

Device	Maximum frequency of generated module (MHz)		
	Additions	Additions and a multiplication	Difference
Xilinx Artix Ultrascale+ (16nm)	1869	284	1585 (84.8%)
Xilinx Kintex 7 (28nm)	437	136	301 (68.9%)

The sequence that contained a single multiplication operation showed a decrease of the module's operating frequency of almost 85% in the Xilinx Artix Ultrascale+ device and of 69% in the Xilinx Kintex 7 device.

The results obtained indicate that the presence of multiplication operations in an extracted sequence has a large impact on the performance gain that can be obtained by using the proposed approach.

4.3.4.4 Conclusion

From the obtained results the speed up that can be obtained depends on multiple factors such as:

- **Number of instructions:** Sequences with more instructions tend to benefit more than shorter sequences, until a certain number of instructions where the performance gain seems to stagnate or even decrease. This is due to the synthesis tool used, Xilinx Vivado, starting to use more complex primitives in larger modules. In the context of the proposed approach, due to the simplicity of the modules and the generated modules being fully combinational, this optimization negatively impacts the performance gains that can be obtained;
- **Inherent parallelism of instructions in sequence:** Sequences that are more inherently parallel tend to benefit more, since they can exploit the acceleration provided by FPGAs in these types of workloads. This also includes sequences that are composed of sub-sequences that are independent, since these sub-sequences do not use the same registers, so they can be computed separately;
- **Type of operations in the sequence:** Sequences that contain more complex operations such as multiplications tend to benefit less from this approach. This is due to current FPGA architectures, that due to being designed to be as flexible as possible, are not designed for this specific purpose. Due to this reason, when the extracted frequent sequences benefit from HW accelerators in the FPGA fabric, such as in the case of multiplications, these sequences incur a large performance gain decrease because of the distributed placement of the HW multipliers in the FPGA fabric leading to longer signal paths which leads to a larger critical path delay of the generated module. This severely limits the performance gain that can be obtained in these types of sequences.

Chapter 5

Conclusion and Future Development

In this thesis, the feasibility of using programmable devices such as FPGAs to accelerate frequent sequences present in compiled binaries was explored, in particular binaries compiled for the RISC-V ISA. This tool-chain can also be used to accelerate MicroBlaze and ARM binaries since the tool-chain is integrated with the SPeCS Binary Translation Framework, that abstracts the ISA of the source binaries used, and the tool-chain was developed to be as generic as possible, so it can support any desired ISA. The tool-chain was also developed in a manner that if required other HDLs could also be targeted, such as VHDL or other higher-level alternatives such as Chisel, with only requiring the development of a generator for the language to be targeted.

This thesis implemented transformation and translation steps that operate in this IL layer that is ISA-independent. Namely, the removal of redundant registers was implemented, balancing of registers through alternative execution paths in a CDFG, and generation of these CDFGs by conversion of a list of the target instruction sequence's parse tree. In this thesis the automatic generation of modules and test-benches, and the automatic validation and characterization of the generated modules were also developed, which enable the use of custom HW accelerators without requiring any sort of digital HW design expertise and the lengthy validation process required.

The developed tool-chain is able to generate synthesizable modules from compiled RISC-V binaries, that according to the results obtained in Chapter 4 seem to indicate that this approach is able to provide measurable performance gains throughout a wide range of domains, and in some cases it provided large performance gains of up to 700%. The performance gains obtained seem to be limited by several factors such as, as mentioned in Section 4.3, the number of instructions in an extracted frequent sequence, the inherent parallelism of the sequence's instructions and the type of operations in the sequences. The current scope of the developed tool-chain is limited since sequences that contain divisions, floating-point operations, memory accesses and branch/jump instructions are not accelerated, and not all of the RISC-V ISA standard extensions are supported. With further optimizations and the inclusion of all instructions in the RISC-V ISA standard extensions, the tool-chain could be applicable in more domains. This would lead the tool-chain to be an alternative to standard HLS flows that would not require any sort of hardware expertise or changing any of the target software's code base.

5.1 Future Development

In order to try and solve the issues mentioned in this chapter, and to try and explore more avenues for obtaining an higher performance gain, the following options should be studied:

- Generation of pipelined modules, that could improve the performance gained in frequent sequences that occur in rapid succession, such as in the case of loops;
- An automatic selector of sequences that picks which sequences to accelerate according to the expected performance gains. This may include repeated iterations of detection and synthesis to determine the expected operating frequencies;
- Broadening the scope to not only include frequent sequences but also sequences that have a large computational cost and could benefit from this approach;
- More optimization passes should be performed on the CDFG, such as the deserialization of sequences in order to better exploit the parallelism acceleration that FPGAs offer. The performance gain that could be obtained by performing such a pass can be seen in Section 4.3.4.2, since the add8par benchmark represents the deserialization of the add8seq benchmark;
- Instead of generating FPGA independent modules, that are not aware of the primitives present in the FPGA fabric, if the tool-chain generated FPGA device specific modules this could lead to overall better performance gains and better FPGA resource utilization. As an example, multiplication operations should be transformed if possible to match the FPGA's multipliers operands sizes (25×18 in Xilinx devices);
- The inclusion in the generated module of pre-design modules (component library) for complex operations such as divisions, square roots and floating-point operations. This can be done by exploring the isomorphism detection capabilities of the JGrapt library used;
- Exploring how to perform memory accesses in order to allow the inclusion of frequent sequences with memory access instructions. This might be particularly beneficial, since generally all functions within the binary need to perform stack operations when they are called or they return;
- The overall flow of this approach is similar to that of most compilers. If this tool-chain was integrated into a compiler framework such as LLVM, this tool-chain could also target not only already compiled binaries but also programs that have not yet been compiled. This would lead to the possibility of accelerating high level languages without requiring the use of HLS workflows;
- Generation of RISC-V custom instructions, and modifying the target binaries to utilize them.

References

- [1] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, and L. Pozzi. Performance and energy benefits of instruction set extensions in an FPGA soft core. In *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, pages 6 pp.–, 2006. doi:[10.1109/VLSID.2006.131](https://doi.org/10.1109/VLSID.2006.131).
- [2] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A Compiler Infrastructure for Accelerator Generators, 2021. arXiv:[2102.09713](https://arxiv.org/abs/2102.09713).
- [3] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64, 2020. doi:[10.1109/ISCA45697.2020.00016](https://doi.org/10.1109/ISCA45697.2020.00016).
- [4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.
- [5] Weisong Shi and Schahram Dustdar. The Promise of Edge Computing. *Computer*, 49(5):78–81, 2016. doi:[10.1109/MC.2016.145](https://doi.org/10.1109/MC.2016.145).
- [6] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. doi:[10.1109/JIOT.2016.2579198](https://doi.org/10.1109/JIOT.2016.2579198).
- [7] B. Moyer. Low-power design for embedded processors. *Proceedings of the IEEE*, 89(11):1576–1587, 2001. doi:[10.1109/5.964439](https://doi.org/10.1109/5.964439).
- [8] Kenneth Flamm. Measuring Moore’s law: evidence from price, cost, and quality indexes. Technical report, National Bureau of Economic Research, 2018.
- [9] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Improving Performance and Energy Consumption in Embedded Systems via Binary Acceleration: A Survey. *ACM Comput. Surv.*, 53(1), February 2020. URL: <https://doi.org/10.1145/3369764>, doi:[10.1145/3369764](https://doi.org/10.1145/3369764).

- [10] Xian-He Sun and Yong Chen. Reevaluating Amdahl’s law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010. URL: <https://www.sciencedirect.com/science/article/pii/S0743731509000884>, doi: <https://doi.org/10.1016/j.jpdc.2009.05.002>.
- [11] Stijn Eyerman and Lieven Eeckhout. Modeling Critical Sections in Amdahl’s Law and Its Implications for Multicore Design. *ACM SIGARCH Computer Architecture News*, 38(3):362–370, June 2010. URL: <https://doi.org/10.1145/1816038.1816011>, doi: [10.1145/1816038.1816011](https://doi.org/10.1145/1816038.1816011).
- [12] Stephen M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, 2015. doi: [10.1109/JPROC.2015.2392104](https://doi.org/10.1109/JPROC.2015.2392104).
- [13] Krste Asanović and David A. Patterson. instruction sets should be free: The case for risc-v. Technical report.
- [14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html>.
- [15] Saman Biokhaghazadeh, Ming Zhao, and Fengbo Ren. Are fpgas suitable for edge computing?
- [16] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009. doi: [10.1109/MDT.2009.69](https://doi.org/10.1109/MDT.2009.69).
- [17] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets. *IEEE Access*, 8:152286–152304, 2020. doi: [10.1109/ACCESS.2020.3017552](https://doi.org/10.1109/ACCESS.2020.3017552).
- [18] Alessandro Barenghi, Michele Madaschi, Nicholas Mainardi, and Gerardo Pelosi. OpenCL HLS Based Design of FPGA Accelerators for Cryptographic Primitives. In *2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 634–641, 2018. doi: [10.1109/HPCS.2018.00105](https://doi.org/10.1109/HPCS.2018.00105).
- [19] Richard L Sites, Anton Chernoff, Matthew B Kirk, Maurice P Marks, and Scott G Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.
- [20] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3):40–45, 2000. doi: [10.1109/2.825694](https://doi.org/10.1109/2.825694).
- [21] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.
- [22] Huynh Phung Huynh, Yun Liang, and Tulika Mitra. Efficient custom instructions generation for system-level design. In *2010 International Conference on Field-Programmable Technology*, pages 445–448, 2010. doi: [10.1109/FPT.2010.5681456](https://doi.org/10.1109/FPT.2010.5681456).
- [23] Philippos Papaphilippou, Paul H. J. Kelly, and Wayne Luk. Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions, 2021. arXiv: [2106.07456](https://arxiv.org/abs/2106.07456).

- [24] D. Oapos Loughlin et al. Xilinx Vivado High Level Synthesis: Case studies. *IET Conference Proceedings*, pages 352–356(4), January 2014. URL: <https://digital-library.theiet.org/content/conferences/10.1049/cp.2014.0713>.
- [25] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi:10.1145/2228360.2228584.
- [26] Nuno Paulino, João Bispo, João C. Ferreira, and João M. P. Cardoso. A Binary Translation Framework for Automated Hardware Generation. *IEEE Micro*, 41(4):15–23, 2021. doi:10.1109/MM.2021.3088670.
- [27] Colin Schmidt, John Wright, Zhongkai Wang, Eric Chang, Albert Ou, Woorham Bae, Sean Huang, Vladimir Milovanović, Anita Flynn, Brian Richards, Krste Asanović, Elad Alon, and Borivoje Nikolić. An Eight-Core 1.44-GHz RISC-V Vector Processor in 16-nm FinFET. *IEEE Journal of Solid-State Circuits*, 57(1):140–152, 2022. doi:10.1109/JSSC.2021.3118046.
- [28] F H McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. 12 1986. URL: <https://www.osti.gov/biblio/6574702>.