FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Single-Program Hardware Specialization

Paulo Miguel de Jesus Coutinho dos Santos Fidalgo

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Prof. João Paulo de Castro Canas Ferreira

July 26, 2026

Single-Program Hardware Specialization

Paulo Miguel de Jesus Coutinho dos Santos Fidalgo

Mestrado em Engenharia Eletrotécnica e de Computadores

Approved in oral examination by the committee:

President: Prof. José Carlos dos Santos Alves
Examiner: Prof. Manuel Gradim de Oliveira Gericota
Member: Prof. João Paulo de Castro Canas Ferreira

July 26, 2026

Resumo

O abrandamento da lei de Moore e o fim da escala de Dennard limitaram os ganhos de desempenho que o avanço tecnológico antes proporcionava, deslocando o foco do desenvolvimento de processadores para a especialização da arquitetura. Um processador de uso geral privilegia a abrangência à custa da eficiência, pelo que uma aplicação concreta tende a utilizar apenas parte do hardware disponível, enquanto a fração não utilizada ocupa área e consome energia. Ao mesmo tempo, a generalidade da arquitetura limita a inclusão de operações especializadas capazes de acelerar a execução. Estas ineficiências são particularmente relevantes na computação na periferia, onde um dispositivo executa inúmeras vezes a mesma carga de trabalho, num contexto sensível à latência e limitado em energia. Especializar hardware para uma única aplicação exigiu, tradicionalmente, conhecimento profundo de hardware associado a longos ciclos de desenvolvimento, o que confinou esta abordagem aos produtos de grande volume capazes de diluir o custo de engenharia.

Esta dissertação apresenta uma metodologia para especializar automaticamente um processador a partir do código-fonte da aplicação e a sua implementação no Automated RISC-V Workload-Specific Specialisation (ARVIS), uma ferramenta de código aberto que a aplica ao núcleo Reduced Instruction Set Computer V (RISC-V) CV32E40P sem qualquer edição manual de Register-Transfer Level (RTL). A metodologia combina três transformações, escolhidas em função da aplicação alvo. A primeira remove a lógica que a aplicação não utiliza. A segunda funde pares de instruções recorrentes em instruções personalizadas e integra-as automaticamente no compilador GNU Compiler Collection (GCC). A terceira acrescenta ciclos em hardware parametrizáveis, com uma profundidade de imbricamento escolhida entre 1 e 8 níveis.

A avaliação inclui 21 aplicações, 19 do *embench-iot* em conjunto com o CRYSTALS-Kyber e o CRYSTALS-Dilithium, com todos os valores obtidos após *place-and-route* num Application-Specific Integrated Circuit (ASIC) CMOS SkyWater de 130 nm e numa Field-Programmable Gate Array (FPGA) Spartan-7 XC7S15. Por si só, a remoção de lógica reduz o produto área-atraso em 57 % no fluxo ASIC e em 49 % no fluxo FPGA, alcançando, em média geométrica, acelerações de 1.41× e 1.27×, respetivamente. A fusão de instruções eleva estas acelerações para 1.54× e 1.34×. Escolhendo a variante mais rápida para cada aplicação, o fluxo atinge 1.55× e 1.36× no conjunto completo, e 1.59× e 1.38× nos casos que beneficiam de mais do que a remoção de lógica, chegando a 2.08× em ASIC e 1.88× em FPGA na carga de trabalho mais favorável.

Nenhuma transformação se impõe em todos os casos. A remoção de lógica é decisiva quando o objetivo é minimizar o produto área-atraso ou o consumo energético, pois apenas retira hardware, enquanto a fusão de instruções e os ciclos em hardware produzem as maiores acelerações nas aplicações computacionalmente intensivas. No conjunto, os resultados mostram que a especialização automática, ao nível do RTL, é um caminho prático para acelerar o desenvolvimento de processadores dedicados.

Palavras-chave: especialização de hardware • RISC-V • instruções personalizadas • fusão de instruções • ciclos em hardware • remoção de hardware • ASIC • FPGA • processadores embebidos

Abstract

The slowdown of Moore's law and the end of Dennard scaling have curtailed the performance gains once delivered by technological progress, shifting processor design towards architectural specialisation. A General-Purpose Processor (GPP) sacrifices efficiency in favour of breadth, so a concrete application tends to use only part of the available hardware, while the unused fraction occupies area and consumes energy. At the same time, the generality of the architecture limits the inclusion of specialised operations capable of accelerating execution. These inefficiencies are particularly relevant in edge computing, where a device runs the same workload many times in a latency-sensitive and energy-constrained context. Specialising hardware to a single application has traditionally required deep hardware expertise associated with long development cycles, confining this approach to high-volume products able to amortise the engineering cost.

This dissertation presents a methodology for automatically specialising a processor from the application's source code and its implementation in Automated RISC-V Workload-Specific Specialisation (ARVIS), an open-source tool that applies it to the CV32E40P Reduced Instruction Set Computer V (RISC-V) core without any manual Register-Transfer Level (RTL) editing. The methodology combines three transformations, chosen according to the target application. The first removes logic that the application does not use. The second fuses recurring pairs of instructions into custom instructions and integrates them automatically into the GNU Compiler Collection (GCC) compiler. The third adds parametrisable hardware loops, with a nesting depth chosen between 1 and 8 levels.

The evaluation includes 21 applications, 19 from *embench-iot* together with CRYSTALS-Kyber and CRYSTALS-Dilithium, with every value obtained after *place-and-route* on a SkyWater 130 nm CMOS Application-Specific Integrated Circuit (ASIC) and a Spartan-7 XC7S15 Field-Programmable Gate Array (FPGA). By itself, logic removal reduces the area-delay product by 57 % on the ASIC flow and by 49 % on the FPGA flow, achieving geometric-mean speedups of 1.41× and 1.27×, respectively. Instruction fusion raises these speedups to 1.54× and 1.34×. Choosing the fastest variant for each application, the flow reaches 1.55× and 1.36× over the full set, and 1.59× and 1.38× in the cases that benefit from more than logic removal, reaching 2.08× on ASIC and 1.88× on FPGA on the most favourable workload.

No transformation dominates in all cases. Logic removal is decisive when the objective is to minimise the area-delay product or energy consumption, since it only removes hardware, whereas instruction fusion and hardware loops produce the largest speedups in computationally intensive applications. Overall, the results show that automatic specialisation at the RTL level is a practical path to accelerating the development of dedicated processors.

Keywords: Hardware specialisation • RISC-V • custom instructions • instruction fusion • hardware loops • hardware pruning • ASIC • FPGA • embedded processors

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. They comprise 17 goals that address the world's most pressing challenges, among them poverty, inequality, climate change, environmental degradation, peace, and justice.

This work contributes to that agenda through the energy and resource efficiency of the computing that runs at the edge. The resulting specialised cores draw less power, occupy less silicon, and finish each task sooner, so the same useful computation is delivered for a smaller energy and material cost. Embedded processors are deployed in very large numbers, so a per-core saving in power and area scales across a whole fleet of devices and over their full operating life.

The specific Sustainable Development Goals supported here have the following names.

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

Target 7.3 By 2030, double the global rate of improvement in energy efficiency

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

Target 9.4 By 2030, upgrade infrastructure and retrofit industries to make them sustainable, with increased resource-use efficiency and greater adoption of clean and environmentally sound technologies and industrial processes, with all countries taking action in accordance with their respective capabilities

SDG 12 Ensure sustainable consumption and production patterns

Target 12.2 By 2030, achieve the sustainable management and efficient use of natural resources

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.3	Workload-specific tailoring of the processor cuts the dynamic power and the energy spent per task of an embedded core, raising the energy efficiency of the compute it carries.	Reduction in total core power relative to the unmodified core
9	9.4	Automated specialisation retrofits an existing processor design for greater resource-use efficiency, delivering the same function from a smaller and leaner core.	Reduction in core area relative to the unmodified core
12	12.2	A smaller core consumes less silicon area and less power for each delivered function, supporting the efficient use of material and energy resources across large device fleets.	Area and power saved per delivered workload, aggregated over the deployed device population

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor João Canas Ferreira for his guidance and support throughout this dissertation. His feedback was essential in shaping the direction of this work and in helping me organise my ideas more clearly. I am especially thankful for his patience and generosity with his time, which were invaluable.

My gratitude to Professor João Canas Ferreira goes beyond this dissertation. I was fortunate to have him as a professor from the first year of my academic path, in courses that introduced me to the topics on which this work is built. I remember looking forward to them, not only because of the lectures themselves, but also because of the conversations that often continued afterwards. Those moments fed my curiosity and made me want to understand the field more deeply.

That early interest eventually led to an internship at INESC TEC, together with Wagner, which opened the door to a research grant and, more importantly, helped shape the academic direction that followed. What began as curiosity in those first-year classes became one of the reasons I moved from Informatics to a Master's in Electrical Engineering, and ultimately led to the work presented here.

My friendship with Wagner dates back to the seventh grade, in school. Years later, after each of us had followed a different academic route and completed our own Bachelor's degree, we challenged ourselves to take on another one, this time together, and continued into the Master's. From school to university, from the internship at INESC TEC to the research that followed, he was always present as a friend, colleague, and companion. Having him beside me made the work far more enjoyable. We shared classes, projects, doubts, deadlines, and achievements, and this year we also share the experience of defending our dissertations. I am sincerely grateful for his friendship throughout all these years.

I am also deeply grateful to my family and to my girlfriend, Mafalda, for their unconditional support. Their belief in me never wavered, even in moments when the path was uncertain or when I doubted my own abilities. Their patience and understanding gave me the stability I needed to continue, especially during the most demanding periods of this work. To them, I owe much more than I can express in these lines, and every achievement along the way was made easier by their presence, their sacrifices, and their confidence in me.

Finally, I would like to thank all my friends who, in different ways, stood by me throughout this period.

During this Master's, I was honoured to receive the *Prémio de Mestrado* of the Ordem dos Engenheiros – Região Norte (OERN), a merit award whose evaluation weighed the project behind this dissertation heavily, so I take it as a recognition by the engineering community of the potential of the work presented here. It is a distinction I share with everyone named above, whose support made it possible.

Paulo Fidalgo

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship. A disclosure of the use of Artificial Intelligence tools is provided in Appendix D.

Paulo Miguel de Jesus Coutinho dos Santos Fidalgo

"If I have seen further, it is by standing upon the shoulders of giants"

Isaac Newton

Contents

List of Figures

List of Tables

List of Acronyms

ABI	Application Binary Interface
ADP	Area-Delay Product
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ARVIS	Automated RISC-V Workload-Specific Specialisation
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
AST	Abstract Syntax Tree
BRAM	Block Random-Access Memory
CGRA	Coarse-Grained Reconfigurable Array
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CSR	Control and Status Register
DAG	Directed Acyclic Graph
DCT	Discrete Cosine Transform
DFG	Data-Flow Graph
DSA	Domain-Specific Architecture
DSE	Design Space Exploration
DSL	Domain-Specific Language
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FF	Flip-Flop
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GCC	GNU Compiler Collection
GPP	General-Purpose Processor
HDL	Hardware Description Language
HLS	High-Level Synthesis

HTML HyperText Markup Language
IoT Internet of Things
IP Intellectual Property
ISA Instruction Set Architecture
LLVM Low Level Virtual Machine
LOB Low Overhead Branch
LSU Load-Store Unit
LUT Look-Up Table
MAC Multiply-Accumulate
MIPS Microprocessor without Interlocked Pipeline Stages
MISO Multiple-Input Single-Output
ML Machine learning
NIST National Institute of Standards and Technology
NTT Number-Theoretic Transform
OBI Open Bus Interface
PC Program Counter
PDK Process Design Kit
PQC Post-Quantum Cryptography
RISC Reduced Instruction Set Computer
RISC-V Reduced Instruction Set Computer V
RTL Register-Transfer Level
SDG Sustainable Development Goal
SHA Secure Hash Algorithm
SIMD Single-Instruction Multiple-Data
SVA SystemVerilog Assertions
TPU Tensor Processing Unit
UVM Universal Verification Methodology
VCD Value Change Dump
VLIW Very Long Instruction Word

Chapter 1

Introduction

1.1 Context

For decades, computing performance improved as a by-product of technology scaling. Moore's law described an exponential growth in transistor density [1], while Dennard scaling kept power density constant as transistors shrank [2]. Both trends have slowed. The end of Dennard scaling limits further gains from raising the clock frequency, since higher rates now meet the power and thermal limits known as the power wall [3], and a growing fraction of a chip cannot be kept switching at full speed within its power budget [4]. This slowdown has shifted computer architecture towards parallelism, heterogeneity, and hardware specialisation as key sources of continued improvement [5].

A General-Purpose Processor (GPP) is built for generality. It may include architectural and microarchitectural support, such as debug logic, interrupt handling, optional arithmetic extensions, and general datapath features, that is valuable for broad deployment but may bring little benefit to a specific target workload, while still occupying silicon area and drawing power. Inefficiency also arises within the datapaths that are exercised, because a general-purpose pipeline is designed to support a wide range of applications rather than to match the operation patterns that dominate any single workload. A processor specialised to a known workload addresses both sources of waste. It removes or simplifies the components the program does not use, and it aligns the remaining datapaths with the operations the workload actually performs, improving energy efficiency, performance, and resource utilisation.

The case for workload-oriented specialisation is strongest at the edge, where many embedded devices execute a fixed or narrowly defined workload over long deployment periods. Such deployments favour low latency, energy efficiency, and responsiveness. Performance in this setting is rarely captured by raw speed alone. Many such devices run on batteries or harvested energy, so the energy spent per useful computation matters as much as, or even more than, the time taken, and the device must keep operating for months or years between service intervals. Workload-oriented specialisation addresses both dimensions at once, reducing the silicon area that draws power and the number of clock cycles a computation needs. The open Reduced Instruction Set Computer

V (RISC-V) Instruction Set Architecture (ISA) [6] makes this direction especially attractive, since its modular definition and the availability of open-source implementations give a practical basis for modifying both the instruction-set interface and the underlying Register-Transfer Level (RTL) [7].

1.2 Problem Statement

Despite the potential of specialised hardware, its development remains costly and time-consuming, and the established approaches each demand a different form of expertise and impose a different trade-off. Manual Application-Specific Integrated Circuit (ASIC) design can deliver highly efficient implementations for a fixed function, but it demands deep knowledge of both the hardware implementation and the application domain, together with long design cycles and high non-recurring engineering costs that confine it to high-volume products. An Application-Specific Instruction-set Processor (ASIP) reduces the effort by starting from a configurable processor template whose instruction set, functional units, memory interfaces, register file, and pipeline can be tailored to the application. The designer must still analyse the workload to decide which customisations are worthwhile, write the corresponding RTL changes, and adapt the compiler toolchain to expose the resulting extensions [8], [9]. High-Level Synthesis (HLS) automates the generation of RTL from a high-level description, but efficient hardware still requires the designer to understand coding idioms specific to HLS, pragma-driven optimisation, and the mapping between source constructs and the generated microarchitecture. In common use, the result is a hardware kernel or accelerator rather than an automatically specialised version of an existing processor and its toolchain.

Prior work also tends to pursue one specialisation axis at a time. Some approaches focus on custom-instruction generation, others on instruction-set subsetting, parameter exploration, or hardware removal. These techniques demonstrate the value of individual optimisations, but they do not provide a unified flow that analyses a workload, chooses among several possible specialisation mechanisms, modifies the processor implementation, adapts the compiler support, and validates the resulting design. The problem this dissertation addresses is therefore the absence of such an automated flow, one that asks for neither hardware design expertise nor deep application-domain knowledge. This leads to the following research question.

> *Can an automated methodology analyse a target program and, by combining several specialisation axes, generate a specialised RISC-V processor, together with the compiler support it requires, that improves area and energy efficiency without degrading the operating frequency?*

1.3 Motivation

The payoff from specialisation is clearest in the embedded and edge systems described above, where a saving in energy or area translates directly into cost and autonomy. The Strategic Research

and Innovation Agenda for European electronic components and systems reflects this, identifying energy-efficient and low-latency computation for embedded artificial-intelligence workloads as a priority [10], which is precisely what tailoring a processor to a known workload provides.

The choice of an open ISA and open-source processor implementations also matters methodologically, since it makes the proposed flow inspectable, reproducible, and extensible, not tied to a proprietary core or a closed compiler. This openness also connects the work to a wider European effort to democratise chip design. The open letter on open-source chips for Europe argues that open-source Electronic Design Automation (EDA), open-source Intellectual Property (IP), and accessible chip manufacturing are critical to education, prototyping, and the growth of chip design expertise in Europe [11]. In this work, that motivation is directly reflected in the methodology, because using open cores and open toolchains allows the generated processors, compiler changes, and evaluation flow to be inspected, reproduced, and extended by others.

1.4 Objectives

The main objective of this dissertation is to develop and evaluate an automated methodology that generates a workload-specific RISC-V processor from a target program. The methodology must identify the processor modifications worth making from the execution characteristics of that program. It must be general enough to accommodate several specialisation axes, and this work implements three of them, namely hardware pruning, custom-instruction fusion, and hardware-loop optimisation. The methodology must then be realised as an automated flow that emits the specialised core together with the software support it requires. Finally, the generated processors must be validated and evaluated against the unmodified baseline core across the simulation, Field-Programmable Gate Array (FPGA), and ASIC flows.

1.5 Contributions

This dissertation makes three main contributions.

The first is a workload-driven methodology for specialising a RISC-V processor to a target program, using information drawn from the program's execution to guide the architectural modifications.

The second is an automated end-to-end toolflow that realises the methodology. It emits a specialised processor implementation, including the automatic generation of the software support that lets the program run on the specialised core, together with the compiler changes that expose the generated architectural extensions.

The third is a validation and evaluation framework. The modifications are integrated into the RTL of an open-source core, producing complete processor variants that can be simulated and implemented, and the framework spans functional simulation, FPGA implementation, and ASIC synthesis and place-and-route, allowing each variant to be compared against the unmodified baseline in area, timing, performance, and energy-related metrics.

1.6 Dissertation Structure

The remainder of this dissertation is organised as follows. Chapter 2 reviews the required background on processor architecture and hardware specialisation, and surveys the state of the art in subtractive and additive approaches, ending with the research gap that motivates this work. Chapter 3 presents the proposed methodology for single-program hardware specialisation, defined independently of any particular RISC-V core or implementation flow. Chapter 4 applies the methodology to the target core and details how each specialisation is realised on a concrete microarchitecture. Chapter 5 describes the software architecture of the automated toolchain that implements the methodology end to end. Chapter 6 presents the experimental setup, analyses the results obtained on both the ASIC and FPGA flows, compares the work with the state of the art, and discusses its limitations. Finally, Chapter 7 concludes the dissertation and outlines directions for future work.

Chapter 2

Background and State of the Art

2.1 Processor Architecture

A processor executes a sequence of instructions stored in memory. The programmer, typically with the help of a compiler, expresses the desired computation as a program written in an ISA, and the resulting binary is placed in instruction memory. The ISA defines the abstract contract between software and hardware. It specifies the instruction set, their encodings, the architectural register state, the memory model, and the privilege structure. From this perspective, the processor has no knowledge of the program's intent. It simply fetches instructions from memory, decodes their bit patterns, executes the corresponding operations, and writes results back to the architectural state. This uniform mechanism is what makes a processor general-purpose, since any computation expressible in the ISA can be executed without changing the hardware.

The RISC-V ISA [6] is an open standard developed at the University of California, Berkeley, and now maintained by RISC-V International. It follows a modular design centred around a small base integer instruction set (RV32I or RV64I), extended by optional standard extensions such as integer multiplication and division (M), atomic operations (A), single-precision and double-precision floating point (F and D), and compressed instructions (C), among others. Because it is free of licensing constraints and designed for extensibility, RISC-V has become widely adopted in both academic research and embedded system design [7], [12].

The execution of a program is realised by a datapath that implements this contract. Figure 2.1 shows the architectural elements that a simplified processor includes. The Program Counter (PC) holds the address of the next instruction to fetch from the instruction memory, advancing either by a natural increment to the next sequential address or by a branch or jump that redirects control flow. The decoder interprets the bit pattern returned by the instruction memory to drive the rest of the datapath. The register file holds the architectural state and supplies operands to the Arithmetic Logic Unit (ALU), which performs arithmetic and logical operations and writes results back to the register file or sends addresses and data to data memory for loads and stores. The branch and control logic determines the next PC from the result of conditional operations and from any unconditional control-flow instructions in the stream [13, pp. 486–499].

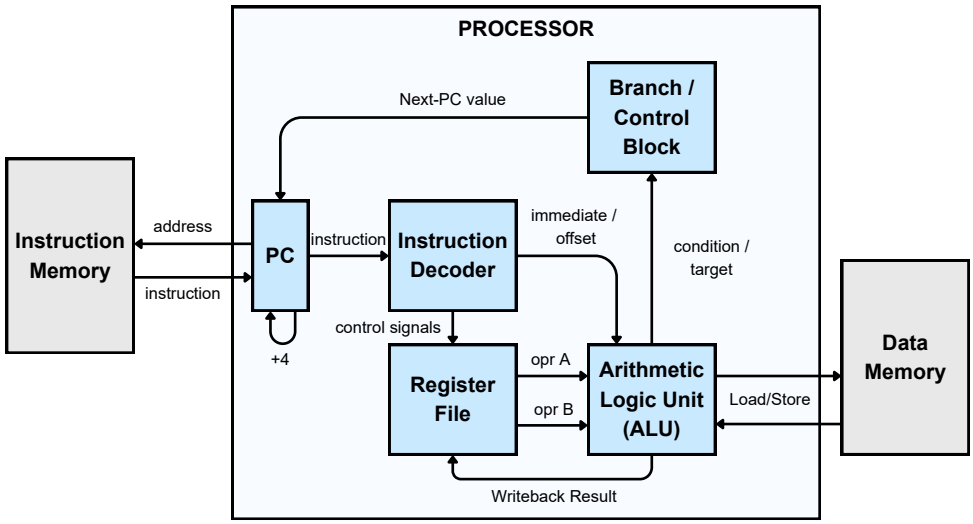


Figure 2.1: Architectural elements of a generic processor. The ALU reads operands from the register file, performs the operation selected by the decoder, and writes the result back. The PC, fed by the branch and control logic, drives the next instruction fetch.

Within the RISC-V base integer ISA, each instruction is encoded as a fixed 32-bit word with a structured layout. The lower seven bits contain the opcode, which defines the instruction class, while the remaining fields encode register specifiers, immediate values, and function bits that refine the operation. Instructions within the same opcode class are distinguished by additional function bits in the instruction, allowing a compact encoding of a large instruction set. The base ISA defines six instruction formats. Figure 2.2 shows the two most relevant to this work, the R-type and I-type, together with the R4-type format used by three-source-register instructions. The decoder exploits this structure by first interpreting the opcode to select an instruction class, and then refining the operation using the function fields, producing the appropriate ALU control signals, register file accesses, and immediate values.

In a simple implementation, each instruction passes through the fetch, decode, execute, memory access, and write-back stages sequentially, with each stage completing one instruction per cycle before the next begins. This limits throughput to one instruction per multi-stage latency. Pipelining improves performance by overlapping these stages across multiple instructions, so that different instructions occupy different pipeline stages simultaneously. While one instruction executes, another is decoded and a third is fetched, increasing throughput at the cost of additional pipeline registers and hazard management logic [13, pp. 518–522]. Although the classical RISC-V textbook example divides the pipeline into five stages (instruction fetch, instruction decode, execute, memory access, writeback) [13, p. 522], practical implementations vary in depth depending on performance and power trade-offs. In the ideal case where every stage handles one instruction per cycle, the steady-state throughput approaches one instruction per cycle.

Hardware specialisation modifies this contract by narrowing the range of intended workloads in exchange for efficiency improvements. The following section explores this spectrum, ranging from fully programmable architectures to fixed-function implementations.

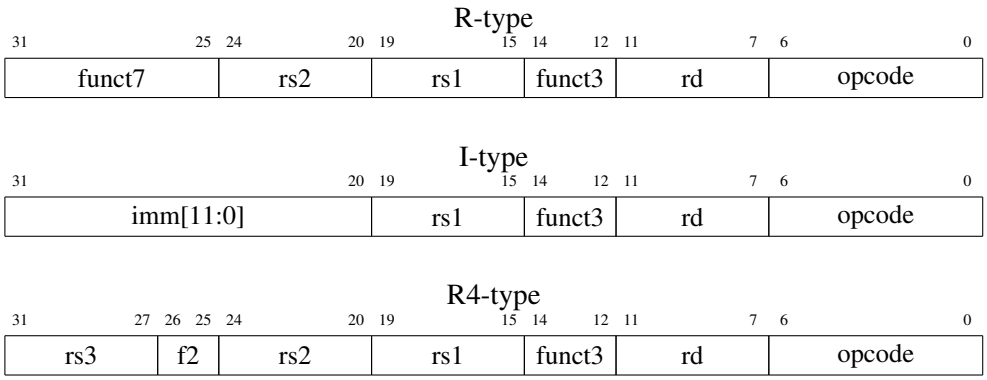


Figure 2.2: The RISC-V R-type and I-type base instruction formats together with the R4-type format used by three-source-register instructions. Every instruction is a 32-bit word. The opcode in the lowest seven bits selects the instruction class, and the remaining fields carry the register specifiers, the funct3 and funct7 refinement, and any immediate operand. R-type encodes register-register arithmetic, I-type covers immediate arithmetic and loads, and R4-type adds a third source register `rs3` in place of part of the function field.

2.2 The Hardware Specialisation Spectrum

Specialisation is the practice of building hardware for a specific workload under specific targets for execution time, energy, or area. A GPP must execute any program written in its instruction set, so its datapath, control logic, and memory hierarchy are sized for the full range of programs the ISA supports. A specialised circuit only has to run the chosen workload, and the design can drop, resize, or replace any element that does not contribute to meeting the targets for that workload.

The cost of generality dominates the energy budget of a programmable processor. On a simple in-order processor at 45 nm and 0.9 V, executing one instruction consumes around 70 pJ, of which only a few pJ is the actual operation. The rest is instruction-cache access (around 25 pJ), register-file read (around 6 pJ), and the pipeline registers, decoder, and control logic that move the instruction through the pipeline [14]. Dally et al. frame the same gap as overhead, reporting that even a simple in-order processor spends over 90 % of its energy on overhead such as instruction fetch, decode, data supply, and control, and that a modern out-of-order processor spends over 99.9 % once branch prediction, speculation, register renaming, and instruction scheduling are added. They put the per-operation cost at 10× to 4000× the energy of the underlying operation, with a 32-bit integer add taking 63 fJ in 28 nm Complementary Metal-Oxide-Semiconductor (CMOS) but the same add executed as an instruction on a 28 nm ARM Cortex-A15 taking over 250 pJ [15]. Specialised hardware avoids this cost by running the workload directly on a circuit shaped for it rather than through a programmable interpreter, and by replacing global memory access with locally-stored data sized to the workload's actual access pattern. The reported energy-efficiency advantage of dedicated hardware over a Central Processing Unit (CPU) reaches two to three orders of magnitude on the workloads where this combination has been measured [14].

Hardware platforms occupy distinct points on this trade-off, summarised in Figure 2.3. A GPP sits at the most flexible end and runs any program expressible in its ISA at the cost of the

per-instruction overhead described above. An ASIP customises the processor for an application class, with the customisation extending beyond instruction-set extensions to register-file size and width, memory hierarchy, pipeline depth, and the inclusion or removal of functional units. Surveys of ASIP customisation report concrete cases such as a fine-grained, in-line error-correction-code extension to a RISC-V core delivering between 10 % and 100 % speedup across the selected algorithms, at the cost of up to a 25 % area increase in one instance, and an off-core clustering coprocessor delivering 13.72× average speedup by sharing a distance-calculation primitive across four clustering algorithms [8]. The same survey frames a recurring trade-off between the size of a custom instruction and its reusability, where smaller shared primitives recur more often and amortise their area across many call sites while larger extensions yield greater per-use speedup but are tailored to a single application, which is why area-reduction methods based on resource sharing across recurring instructions are central to keeping the cost of customisation low [8]. A Digital Signal Processor (DSP) goes further by pre-specialising the entire pipeline at design time for one application domain, typically signal processing, with deep pipelined multipliers, post-increment addressing modes, hardware loops, and Single-Instruction Multiple-Data (SIMD) lanes tuned to the kernels expected at runtime. A Coarse-Grained Reconfigurable Array (CGRA) replaces the fixed pipeline with a fabric of word-wide ALUs and small local memories that can be reconfigured at runtime to map a small set of computation kernels, retaining some programmability while eliminating most of the per-instruction fetch and decode overhead. An FPGA replaces a fixed datapath with reconfigurable logic that can be programmed to implement any digital circuit fitting its capacity, with the trade-off that each circuit takes up area whether or not it is in use. At the most specialised end of the spectrum sits the ASIC and its more recent incarnation as a Domain-Specific Architecture (DSA). The Google Tensor Processing Unit (TPU) is the canonical example, dedicating its silicon to matrix multiplication for neural-network inference and reporting roughly 29× speedup and over 80× better energy efficiency than a contemporary GPP on the same workload [5].

Moving along the spectrum from GPP to ASIC trades the range of programs the platform can run for higher efficiency on the supported workload, at the cost of design effort. A GPP is the most expensive in operation but the cheapest to develop because the design is amortised across every workload it ever runs. An ASIC is the cheapest in operation per task but the most expensive to develop because the entire memory hierarchy, datapath, and control logic are bespoke.

Restricting the workload to a single program or a small subset of programs lets a GPP be further specialised, which is the class of specialisation surveyed in the next section.

2.3 State of the Art in Processor Specialisation

Existing tools for processor specialisation fall into two categories, subtractive specialisation and additive specialisation.

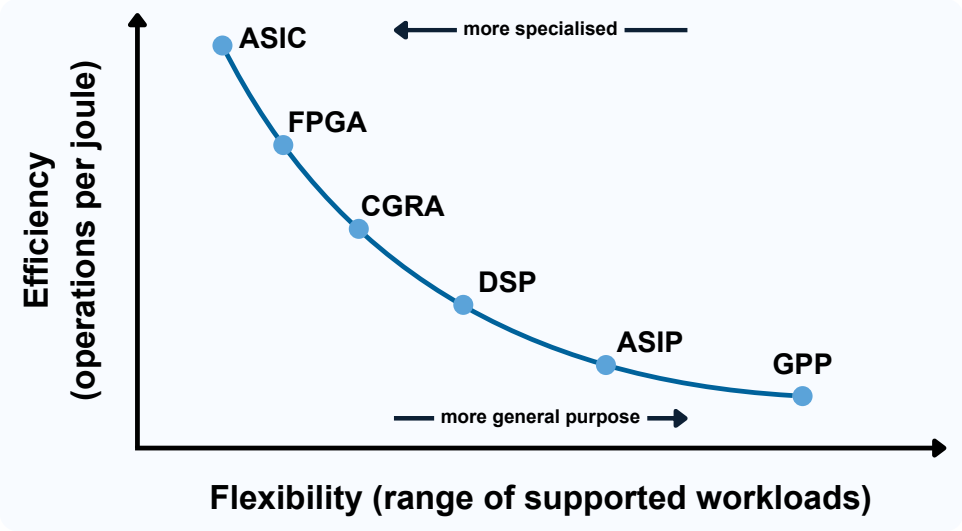


Figure 2.3: The hardware specialisation spectrum. Efficiency increases as the platform commits to a narrower class of workloads, at the cost of flexibility. The GPP runs any program but at the lowest efficiency, while the ASIC runs a single fixed task at the highest efficiency.

2.3.1 Subtractive Specialisation

Subtractive specialisation removes from a complete GPP the logic that is not required by a target application. The approaches differ in the granularity of their inputs, the level of abstraction at which the analysis is performed, and the direction of the applied transformations.

Cherupalli et al. [16] symbolically simulate a target binary on the gate-level netlist of an openMSP430 ultra-low-power microcontroller, propagating unknown values cycle-by-cycle to identify every gate that the simulation proves cannot toggle on any input the binary could receive. Each provably untoggled gate is then detached from the netlist with its fanouts tied to the gate's proven constant value (0 or 1), and the netlist is resynthesised to propagate the constants and eliminate the unreachable logic that the propagation exposes, resulting in a minimal core, which the authors refer to as *bespoke*. The flow is automated end-to-end. The evaluation runs fifteen benchmarks across three groups, specifically nine embedded-sensor kernels (*binary search*, *division*, *insertion sort*, *integer averaging*, *FIR filtering*, *multiplication*, *run-length encoding*, *threshold detection* and *TEA encryption*), four EEMBC DSP workloads (*FFT*, *Viterbi decoder*, *convolutional encoder*, *autocorrelation*) and two openMSP430 unit tests for the interrupt and debug paths. Synthesised in TSMC 65GP (65 nm) at 1 V and 100 MHz, the reductions average 62 % in area and 50 % in power. The debug-interface unit test sets the upper bound, saving 92 % in area and 74 % in power because non-debug applications allow the removal of all the debug logic, and *FFT* sets the lower bound at 47 % in area and 37 % in power.

The authors draw three conclusions that distinguish bespoke pruning from coarser alternatives. Compared to module-level pruning analogous to module removal in a configurable core, fine-grained bespoke pruning saves an additional 22 % to 75 % in power across the suite, with 35 % on average. Module-level power gating offers a weaker comparison even in its theoretical best

case, i.e. a perfect oracle deciding when each module is unused and zero hardware cost for the gating itself. In that idealised setting it still reaches at most 13 % power reduction on the same benchmarks, where bespoke pruning achieves a minimum of 37 %, and any realistic fine-grained power gating would itself incur a 40 % to 50 % area overhead for isolation and retention cells, ruling it out for ultra-low-power targets. Profiling-based pruning is also insufficient on its own because the set of gates an application exercises can vary by up to 13 % across different inputs to the same application, so gates that a profile-based analysis flags as unused on the profiling inputs might still be needed for other inputs and the resulting design would fail on them. The X-propagation symbolic simulation that the paper introduces provides the soundness guarantee that empirical profiling cannot.

To support multiple applications on the same bespoke processor, the authors propose intersecting the untoggled-gate sets across several binaries, so that the resulting core retains every gate that any of the supported binaries might exercise. The best ten-program combinations from the benchmark suite still admit a 41 % area and 20 % power reduction relative to the unmodified openMSP430. To support arbitrary in-field updates, the authors co-synthesise a Turing-complete escape instruction (subneg), which adds 8 % in area and 10 % in power on top of the single-application bespoke design while still leaving the resulting processor 56 % smaller and 43 % lower-power than the baseline. The authors acknowledge that performance-enhancing features such as caches, out-of-order execution and branch prediction introduce non-determinism into the instruction stream, but they argue the co-analysis can absorb it at the expense of analysis runtime. A cache tag check is treated as an unknown so that both the hit and miss paths are explored, branch prediction reuses the taken and not-taken path exploration the analysis already performs, and out-of-order execution might be reached by exploring the data-flow graph instead of the control-flow graph. None of these extensions are evaluated, so the reported gains characterise the technique only on its stated target class, the ultra-low-power in-order microcontrollers exemplified by the openMSP430, with scaling to more complex cores and applications left to heuristics the paper does not implement.

Bleier et al. [17] adopt a different approach, anchoring the pruning to a fixed ISA subset instead of a single binary. They replace the symbolic-simulation argument of Cherupalli et al. with formal property checking. The retained ISA subset is encoded as an environment assumption written in SystemVerilog Assertions (SVA), expressed as a disjunction of the bit patterns of the retained instructions and applied at the processor's instruction port. For ISAs with variable-length instructions or indirect branches, the instruction port cannot be constrained directly because the bit pattern at the port becomes ambiguous in the first case and unpredictable in the second. The assumption is moved instead onto a cutpoint placed on an internal pipeline register, typically the wire feeding the decoder. The cutpoint detaches that wire from the upstream fetch logic and treats it as a free variable, and the SVA assumption constrains the free variable to the bit patterns of retained instructions. The analysis thus sees only valid retained instructions reaching the decoder, regardless of how the fetch path would otherwise have produced them. The property checker then queries every gate of the synthesised netlist for whether its output is provably constant (always 0 or always 1) under the resulting assumption. For each gate that the check proves constant,

the net the gate drives is detached from the gate and tied to the proven value. The netlist is then resynthesised with the standard logic synthesis flow, which propagates the constants forward, simplifies the gates whose inputs have become constant, and eliminates the unreachable logic that the constant propagation exposes. No instruction is removed directly. Instructions disappear from the implementation as a consequence of the environment assumption combined with constant propagation in the second synthesis pass. The result is a smaller core that runs any program expressible in the reduced ISA. The flow is automated end-to-end, with the property check, the constant-tying step and the resynthesis all running without manual intervention. The framework is evaluated on three embedded-class cores synthesised in 45 nm NANGATE with Mentor Questa Formal as the property checker. The three designs are the in-order RISC-V Ibex (RV32imcz, $\approx$ 10 kgates), an obfuscated Arm Cortex-M0 of similar size implementing Armv6-M, and the out-of-order RISC-V RIDECORE ($\approx$ 100 kgates, RV32im). Constraining Ibex to the MiBench instruction subset (only 53 of 78 instructions exercised) yields a 14 % reduction in gate count and a 23 % reduction in area relative to the unoptimised Ibex core. The obfuscated Cortex-M0 reaches a 20 % area reduction under its full supported ISA, with the MiBench subset yielding the same result on this core because its obfuscation prevents the finer cutpoint-based constraints. RIDECORE achieves 14 % to 17 % gate-count reductions, comparable to Ibex's 14 % in relative terms despite RIDECORE being roughly ten times larger. With the environment assumption set to permit every instruction that the Ibex core officially supports (so the legal workload is unchanged), the property check already eliminates approximately 10 % of the area. The savings come from gates whose outputs the property checker proves constant under any valid Ibex execution but which the standard logic-synthesis flow nevertheless leaves in the netlist. Synthesis-time elimination of unused logic is therefore incomplete by approximately 10 % on this core, and that headroom is recoverable without changing the supported ISA at all.

Si et al. [18] operate at higher abstraction levels than the two works above. Cherupalli et al. and Bleier et al. both take the synthesised gate netlist of an RTL-defined processor as their input and remove gates from it. Si et al. start one step earlier, with a processor that is itself described in synthesisable C instead of in RTL. They prune unused C code, run HLS to obtain RTL, prune unused RTL, run logic synthesis to obtain a gate netlist, and prune unused gates. Pruning happens three times in succession, once per abstraction level, with each pass fully automated. At each level the identification of unused parts is profile-driven via `gcov` at the C source level, ModelSim simulation traces at the RTL level, and Value Change Dump (VCD)-based gate-toggle analysis at the gate netlist. The actual removal differs by level. At the C and RTL levels the lines of source that the profiling report flags as unexecuted are deleted from the description before it is fed to the next synthesis stage. At the gate level untoggled gates cannot simply be cut, since their fanouts would be left floating, so the same constant-tying mechanism that Cherupalli et al. apply is reused, though here it is the final step and is not followed by a further resynthesis pass. The motivation for the multi-level flow is that re-running synthesis after each pruning pass lets the synthesis tool re-optimise the result for the smaller workload. HLS can allocate fewer functional units (no divider when integer division is gone, fewer multipliers when most multiplies have been removed) or

substitute smaller and slower adder implementations when the timing headroom allows.

The evaluation runs nine benchmarks from several open-source suites (four DSP kernels, three image-processing kernels, and two of more general use) on a 32-bit scalar non-pipelined Microprocessor without Interlocked Pipeline Stages (MIPS) core synthesised in Nangate 45 nm at 100 MHz, with NEC CyberWorkBench 6.1.1 as the HLS engine and Synopsys Design Compiler 2016 as the back-end. Against the original processor, the multi-level flow saves 71.0 % in area and 62.1 % in power on average across the eight benchmarks of the exact flow. Compared to gate-level-only pruning of the same HLS-generated baseline, i.e. the same kind of pruning that Cherupalli et al. apply to openMSP430, the multi-level flow gains a further 9.3 % in area, 11.6 % in power and 8.9 % in delay. For applications that tolerate output imprecision, such as the DSP and image-processing benchmarks, the same flow can be applied with an output-error budget that drives further approximation of the datapath. Setting the maximum output error to 10 % or 20 % saves an additional 11.7 % or 19.5 % in area respectively over the exact bespoke version.

Three limitations affect how these numbers should be read. First, the methodology requires the processor itself to be described in synthesisable C rather than in RTL. Most published embedded cores are written in RTL, so applying Si's methodology to them would require porting the processor to C first. Second, the authors acknowledge that HLS cannot model the timing of forwarding and hazard-detection logic in a pipelined core, which restricts the methodology to simple non-pipelined processors even when a C model is available. Third, the identification of removable logic is profiling-driven, gathered from `gcov`, RTL simulation traces and gate-toggle activity over a finite set of inputs, so unlike the input-independent symbolic analysis of Cherupalli et al. the specialised core is sound only for the profiled behaviour, and the authors themselves note that the optimisations are entirely data dependent and need training data representative of the final workload.

Chaidos et al. [19] follow the per-binary perspective of Cherupalli et al. but operate at the RTL level rather than at the gate netlist, and on a different process technology. The Zero-Riscy core (PULP, two-stage RV32) and the TP-ISA minimal-Reduced Instruction Set Computer (RISC) core are synthesised in an EGFET printed-electronics standard cell library, where typical operating frequencies are a few hertz to a few kilohertz and feature sizes are several microns. A profiling pass over the compiled assembly identifies which architectural components and instructions the target binaries do not exercise, and the corresponding RTL changes are then applied. The debug unit, the interrupt controller and the compressed decoder of Zero-Riscy are removed, unused RISC-V instructions (`slt`, most Control and Status Registers (CSRs), system calls, `mulh`) are eliminated, the register file is trimmed to twelve registers, and the PC is shortened from 32 to 10 bits with the base address registers shortened from 32 to 8 bits, to match the small code size and limited address range required by the target workloads. The paper does not document the mechanism by which the resulting RTL changes are applied, so the removal appears to be performed manually, not through a parameterised RTL configuration or an automated rewrite. Across classification and regression variants of a multilayer perceptron and a support vector machine (MLP-C, MLP-R, SVM-C and SVM-R), giving three multilayer perceptrons and three support vector machines

over the Cardiography, RedWine and WhiteWine datasets from the UCI repository, pruning alone delivers a 10.6 % area reduction and an 11.4 % power reduction at zero accuracy loss. Chaidos et al. also add a SIMD Multiply-Accumulate (MAC) unit with four precision configurations (32, 16, 8 and 4 bits). The 16-bit configuration takes the combined area and power reductions to 22.2 % and 23.6 % with a 33.79 % speedup at zero accuracy loss, and dropping further to 8 and 4 bits reaches 29.3 % and 36.5 % area at the cost of a 0.5 % and a 15.66 % accuracy loss respectively. The work is therefore the only one of those surveyed here to combine subtractive specialisation with additive specialisation. The evaluation rests on six small Machine learning (ML) models drawn from a single benchmark family, so generalisation to non-ML or to larger workloads is not demonstrated. The headline gains are tied to a printed-electronics standard cell library, where the area and timing trade-offs differ from those of any silicon implementation.

While all the previous works start from a complete GPP and remove what the application does not use, Raisi-Ardali et al. [20] invert the direction of the transformation, assembling the core bottom-up from a library of pre-verified per-instruction RTL blocks, and are alone among the surveyed works in making formal verification an integral part of the methodology rather than an afterthought. Each block is independently verified with SymbiYosys, and the per-block testbench is validated with MCY (Mutation Cover with Yosys), an open-source tool from YosysHQ that injects small modifications into the design and counts how many the testbench catches. A generator then pulls automatically the blocks corresponding to the instructions that the target binary uses into a single combinational Modular Execution Unit, which is stitched with fixed fetch, register-file and memory-interface modules. The methodology targets flexible-electronics technology, so the evaluation runs 22 Embench benchmarks plus three extreme-edge applications (*armpit*, *xgboost*, *af_detect*) on the Pragmatic 0.6 μm IGZO process, comparing against an ISA-complete RISSP-RV32E baseline built with the same methodology and against the bit-serial Serv core. Profiling shows that the applications use only 24 % to 86 % of the RV32E ISA, and the resulting cores save 8 % to 43 % in area and 3 % to 30 % in power post-synthesis against RISSP-RV32E, and the *xgboost* core reaches 42 % in area and 21 % in power against the same baseline after place-and-route. Among the per-application RISSPs, the smallest area belongs to the one generated for *xgboost*. At synthesis the *xgboost* RISSP is 23 % larger than Serv in NAND2-equivalent gates, but after place-and-route at 300 kHz the same RISSP becomes 11 % smaller than Serv, because Serv's 60 % Flip-Flop (FF) proportion (against around 6 % in the RISSPs) drives a heavier clock-tree overhead that erases Serv's synthesis-time area advantage. Energy per instruction is roughly 40$\times$ better than Serv across all benchmarks, since Serv averages 32 clock cycles per instruction in its bit-serial datapath. Because each RISSP is single-cycle and combinational throughout, the methodology fits the kHz frequencies of flexible electronics but does not transfer to silicon targets that require a multi-stage pipeline to reach competitive $F_{\max}$.

2.3.2 Additive Specialisation

Where subtractive specialisation removes logic, additive specialisation introduces additional capabilities. The most common form is custom instructions, which add new opcodes to the ISA to-

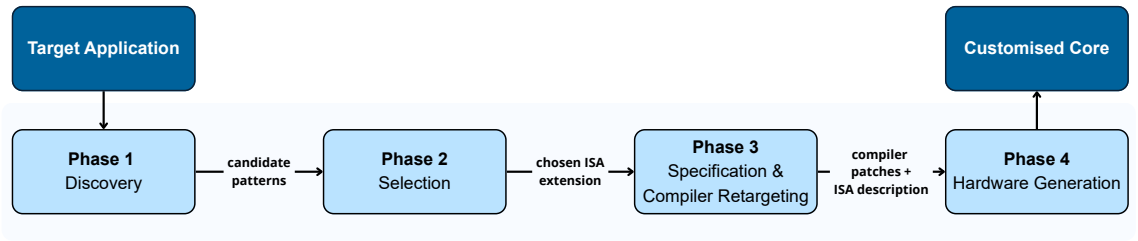


Figure 2.4: The four phases of custom-instruction extension and the artefacts that flow between them.

gether with the functional units that execute them. Some replace short recurring multi-instruction sequences such as a fused load-effective-address or a packed-SIMD dot-product step. Others add primitive operations that the base ISA does not provide directly, such as a single instruction that reverses the bit order of a register or one that performs a saturating addition. A second form is hardware loops, in which a dedicated hardware unit takes over the loop control, allowing the per-iteration branch of a software loop to be removed.

2.3.2.1 Custom-Instruction Extensions

Defining a custom-instruction extension requires, in principle, four phases, summarised in Figure 2.4. The first phase discovers the correct patterns to accelerate, by identifying recurring or hot instruction sequences in the application. The second is optional and selects, among the discovered candidates, a subset that fits an area or count budget under given microarchitectural constraints. The third retargets the compiler to emit the new opcodes in place of the original sequences, although some workflows skip this step by hand-writing assembly that uses the new opcodes directly. The fourth provides the hardware support that performs the new operations, either by reusing existing functional units, by adding new tightly-coupled units in the pipeline, or by attaching a coprocessor through a standardised interface. While these are the four phases of an idealised flow, the works in the literature differ along several points. Some address only a subset of the phases while others cover all four. Some are fully manual, others fully automatic, and several mix the two.

A custom instruction replaces a sequence of base instructions with a single instruction that performs an equivalent operation more efficiently, saving both clock cycles and energy. The savings come from a reduced instruction count, which removes the per-instruction fetch and decode overhead, combined with a tailored datapath that carries out the operation more directly than the general-purpose units. The size of the saving also depends on the integration position. Reusing an existing functional unit adds no new latency boundary, a new tightly-coupled unit adds only its own latency, and a coprocessor adds an offload-latency overhead on every call, which can dominate when the replaced sequence is short, as OpenASIP's [9] *nettle-aes* regression on Rocket shows. Figure 2.5 contrasts the three positions on a simplified pipelined core.

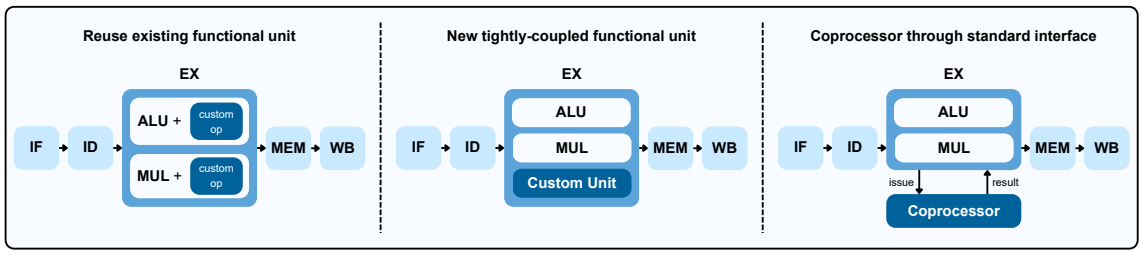


Figure 2.5: Three integration positions for a custom instruction in a typical RISC-V pipeline. From left to right, the custom operation reuses an existing functional unit, the custom operation runs in a new tightly-coupled functional unit added in the execute stage, or the custom operation is offloaded to a coprocessor through a standardised interface such as CV-X-IF or RoCC.

The works surveyed next automate different subsets of these four phases, ranging from single-phase tools that address one stage to integrated flows that span all four. ARISE [21] is a single-phase tool centred on discovery, generating and ranking candidate instructions but performing none of the hardware-aware selection, compiler retargeting or RTL generation the later works automate. It parses RISC-V assembly and applies a greedy algorithm, restricted to Multiple-Input Single-Output (MISO) instructions in the manner of MAXMISO, that extends each candidate by appending data-flow-connected instructions until the freely-encodeable instruction bits are exhausted, generating up to sixteen instructions with a nine-bit opcode per optimisation goal (static code size, dynamic code size, or dynamic instruction count) and ranked by the chosen metric. The resulting instruction-set extensions are written to CoreDSL files, ready for downstream compiler and hardware tools. The evaluation runs Embench-Internet of Things (IoT) compiled with the Seal5 retargeting compiler at `-Oz`, reporting average reductions of 1.48 % in static code size, 3.84 % in dynamic code size and 7.39 % in dynamic instruction count, with a best case of 24.2 % on *matmult-int* for the instruction-count metric and a code-size regression on *picojpeg* caused by compiler register-pressure changes that ARISE does not model. Consequently, the headline numbers are code-density proxies, not measured speedups. The assembly-level extraction cannot anticipate compiler decisions, so the *picojpeg* regression is structural, and the same concern applies to any approach that operates upstream of the back-end. ARISE also stops at the CoreDSL output and does not generate RTL.

Where ARISE addresses pattern discovery, Seal5 [22] addresses compiler retargeting, also as a single-phase tool. It consumes a CoreDSL2 description (a Domain-Specific Language (DSL) for specifying instruction-set behaviour) and produces a retargeted Low Level Virtual Machine (LLVM) toolchain, generating the compiler patches and applying them through LLVM's own build infrastructure instead of leaving them to the designer. Patch generation runs in two stages. The first emits TableGen records (LLVM's target-description format) and C++ patches for assembler and built-in support. The second, run on an intermediate LLVM build with the first stage's patches applied, extracts SelectionDAG patterns from the CoreDSL2 behavioural description through a custom Clang front-end so that the generated patterns match the Directed Acyclic Graphs (DAGs) LLVM produces during regular compilation. The flow also generates support for sub-word SIMD

types so they can use the existing general-purpose register file directly without extra moves between scalar and vector data, and lets the compiler generate the narrow SIMD instructions automatically from regular scalar loops. The evaluation runs eighty benchmark configurations in total, drawn from MLPerfTiny (40), Embench-IoT (22), CoreMark (1) and seventeen synthetic SIMD workloads, on a Verilator simulation of CV32E40P with the full Core-V extension. Seal5 emits 89 unique Core-V instructions (32 % of those available) where the OpenHW reference compiler emits 43 (16 %). With SIMD the runtime reduction averages 24 % over upstream LLVM and peaks at 86 % (7×) on selected MLPerfTiny benchmarks. Without SIMD the runtime reduction averages 14 %, comparable to the OpenHW reference compiler at 15 %. Area, $F_{\max}$ and power are not reported, because Seal5 generates only the compiler and the designer must supply the corresponding RTL separately. Complex Core-V instruction classes (shuffle, pack, hardware-loop) still require manual implementation effort, since the automatic extraction does not cover them.

OpenASIP [9] broadens the scope to a multi-phase toolchain that addresses specification, compiler retargeting, and hardware generation. It takes an architecture definition file describing the instruction set, together with operation semantics expressed either as Hardware Description Language (HDL) snippets or as DAGs of already-implemented primitive operations. From that description a tool, RISCVTDGen, transforms the DAG-based instruction descriptions into extensions to the LLVM RISC-V target descriptor, which are compiled into a dynamically-loaded library that the backend driver `llc` loads at compile time to select the custom instructions automatically without rebuilding the toolchain. A caching mechanism stores the built target modules and reloads them on a hit, invoking LLVM's TableGen to regenerate the back-end only on a miss. For instructions whose semantics cannot be expressed as DAGs, the flow falls back to automatically generated intrinsics that wrap inline assembly. The function-unit generator builds the operation hardware including pipeline registers from DAGs of primitives, and wraps the generated unit with one of the standardised CV-X-IF or RoCC coprocessor interfaces. The motivation is portability, since the coprocessor can attach to any core that already implements the interface without modifying the core itself. The costs are the protocol latency added at every offload and, for a core that does not yet implement the interface, the effort of adding it. The evaluation synthesises three host cores (CVA6 single-issue, CVA6 dual-issue and Rocket) in the ASAP7 predictive Process Design Kit (PDK) with Cadence Genus, running BEEBS *crc*, *aha-compress*, *nettle-aes* and a 4-bit 128×128 matrix multiplication kernel that exploits a packed SIMD integer dot product. Average execution-time reductions are 38 % on CVA6 single-issue, 40 % on CVA6 dual-issue and 27 % on Rocket. The matrix multiplication contributes the largest single reduction (88 % on CVA6 single-issue, 87 % on Rocket), while *nettle-aes* on Rocket regresses because the coprocessor offload latency exceeds the native execution time of the three suboperations forming the custom instruction. Area overheads are 7 % on CVA6 single-issue, 8 % on CVA6 dual-issue and 6 % on Rocket. Critical-path increases are 7 %, 1 % and 2 % respectively, the larger CVA6 single-issue figure reflecting the issue logic added at the offload point. Power is not reported. The designer remains responsible for selecting the custom instructions and writing their operation descriptions, since OpenASIP does not perform pattern matching automatically. Beyond this manual step, the *nettle-aes* regression

on Rocket reveals a structural limitation of interface-based acceleration. Short fused operations cannot benefit when the coprocessor offload latency approaches or exceeds their native execution time.

ADVICE [23] takes a different architectural route to multi-phase coverage. Where OpenASIP generates accelerator RTL from an architecture description and integrates it through a coprocessor interface, ADVICE describes the entire system, including the base RISC-V CPU, in synthesisable ANSI-C and runs a single HLS pass over the merged source. The flow has four steps. The designer marks the kernels to accelerate with `//ADVICE HW_START` and `//ADVICE HW_END` pragmas in the application's C source, and an exhaustive HLS Design Space Exploration (DSE) over loop-unroll, array-partition and function-inline pragma combinations characterises each kernel by area, latency and power. The remaining application code is compiled by the standard RISC-V compiler, and a post-processing pass parses the resulting assembly to replace the function call to the accelerated kernel with a custom instruction in the format `hwacc rs, offset(rd)`, generating one custom instruction per accelerated kernel. The chosen kernel is then copied into the C description of the base CPU, in the switch-case branch corresponding to the new opcode, and the merged description is run through HLS to produce the final RTL. The single-process synthesis is the architectural payoff. HLS can share an adder of the accelerator with the ALU adder, reuse registers across the CPU register file and the accelerator, and rebalance the critical path across the CPU/accelerator boundary, none of which is available when the accelerator is synthesised as a separate module. The evaluation uses Comet, a five-stage pipelined RISC-V described in synthesisable C, on seven individual benchmarks (*ave8*, *fir*, *interp*, *adpcm*, *cholesky*, *fft* and *jpeg*) plus a JPEG-encoder case study that explores kernel combinations (Discrete Cosine Transform (DCT), quantisation, run-length encoding, Huffman coding) at the application level. The flow is run with NEC CyberWorkbench 6.1.1 as the HLS engine, Synopsys Design Compiler for logic synthesis, and Nangate 45 nm OpenCell as the target technology library. Across the seven individual benchmarks the proposed flow averages a 78 % area overhead against the unmodified RISC-V baseline, a 98 % energy reduction and a 27× speedup, against 108 %, 85 % and 16× for an alternative integration that synthesises the accelerator as a separate module connected through a hand-shake protocol. The single-process HLS flow constrains the base core to be HLS-friendly, available as a synthesisable ANSI-C description rather than as RTL, a constraint the five-stage pipelined Comet core used here satisfies, so the approach is bounded to processors describable in that style.

CIDRE [24] goes one step further, automating discovery, selection, and hardware generation in a single end-to-end flow. The input is a compiled RISC-V binary and a profile of its execution. CIDRE lifts each hot basic block into a Data-Flow Graph (DFG) annotated with operand positions and commutativity, then enumerates candidate patterns with the algorithm of Xiao and Casseau [25]. Given the DFG and limits on how many input and output operands the custom functional unit may have, the algorithm enumerates every subgraph, connected or disconnected, that satisfies convexity (no path between two nodes inside the pattern may pass through a node outside it) and the input/output count limits, the disconnected patterns capturing instruction-level parallelism within a single custom instruction. Figure 2.6 illustrates the constraint with a convex

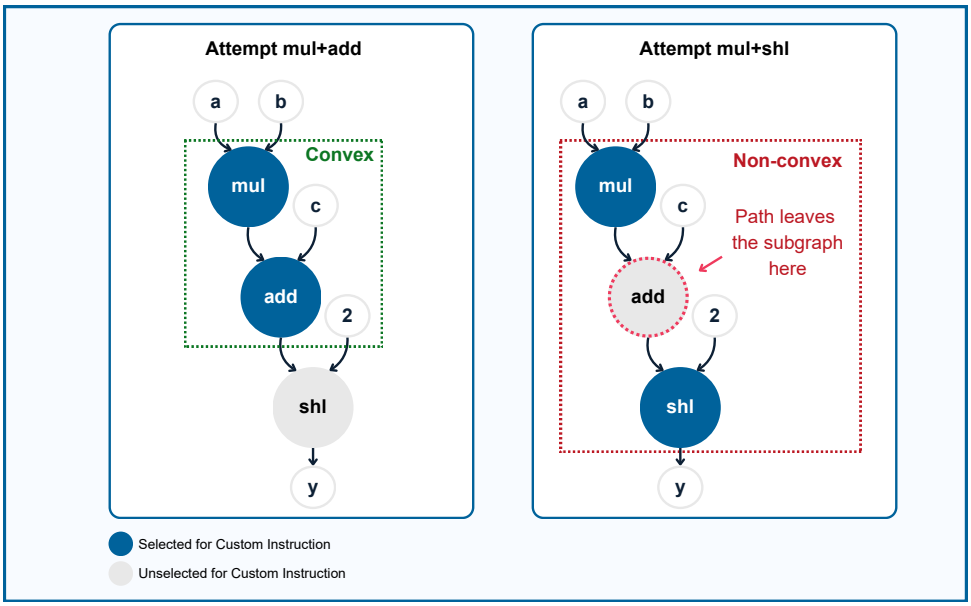


Figure 2.6: Convexity constraint on candidate subgraphs. (a) The selection {`mul`, `add`} is convex because every path between selected nodes stays inside the selection. (b) The selection {`mul`, `shl`} is non-convex because the path from `mul` to `shl` passes through `add`, which is not selected. Non-convex subgraphs are excluded from the candidate set because a single functional unit cannot leave its boundary, execute an external operation and re-enter.

and a non-convex selection on the same data-flow graph. The algorithm grows each pattern by adding nodes in reverse topological order, and a node-filtering step rules out any addition that would break convexity, so every enumerated pattern is convex by construction without an explicit convexity check at each step. The runtime grows polynomially with the basic-block size, with the polynomial's degree determined by the input and output bounds. On the GSM benchmark from MediaBench, the algorithm reports 1 454 539 valid patterns at 490 nodes in 13.69 s for up to six inputs and two outputs. CIDRE then removes isomorphic duplicates from the enumerated set by graph canonisation. The canonisation method, due to Ahn and Choi, assigns each graph a unique label such that two graphs share the same label if and only if they are isomorphic, so deduplication reduces to comparing labels in linear time, in place of the quadratic number of pairwise isomorphism comparisons that a naive deduplication would require.

The selector grows the chosen set one candidate at a time. Each candidate is scored by a merit function, specifically the number of base instructions the candidate replaces (a candidate of size n saves $n-1$ base instructions per instance, since the custom instruction itself counts as one) multiplied by the dynamic execution frequency of the basic block where the pattern occurs, summed over every instance of the candidate in the program. The greedy step adds the highest-merit candidate to the chosen set. The two-optimal look-ahead refines this step by trying every plausible immediate candidate together with the candidate the greedy step would select after it, and committing to the immediate choice only if no other such pair has a higher combined merit. The contribution of CIDRE beyond the greedy core is synthesis-in-the-loop microarchitecture

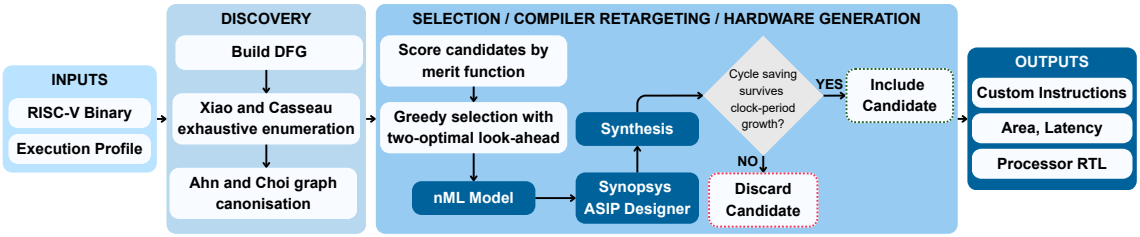


Figure 2.7: The CIDRE end-to-end flow. Discovery (build the DFG, exhaustive enumeration via Xiao and Casseau, isomorphism detection via Ahn and Choi canonisation) feeds selection, where each candidate is scored by the merit function, selected greedily with a two-optimal look-ahead, and synthesised through Synopsys ASIP Designer to check whether its clock cycle saving survives the clock-period growth. Candidates that pass the check are mapped to the RISC-V CUSTOM-0/1/3 opcodes and emitted as the customised processor RTL.

awareness. Every chosen instruction is synthesised inside a full processor model (Synopsys ASIP Designer with the trv64p3 baseline, 32/28 nm CMOS at 0.85 V), and instructions whose clock cycle saving is wiped out by clock-period growth are discarded. The benchmark *mb-susan* with up to two inputs and one output illustrates why the check matters. It achieves less than a 1.01× clock cycle reduction, which is consumed entirely by the resulting clock-period increase, leaving a 0.99× net slowdown that a clock-cycle-only merit function would have accepted. CIDRE writes the surviving instructions as nML descriptions and passes them to Synopsys ASIP Designer, which generates the RTL for the new instructions. The hardware generation is therefore handled by the commercial toolchain, while the reported speedup is the latency saving the merit model predicts and not the result of running compiled code, since the flow does not retarget a compiler to emit the custom instructions. Figure 2.7 summarises the full pipeline. The evaluation runs six Embench and two MiBench workloads compiled at `-O2`. Five instruction templates are mapped to the RISC-V CUSTOM-0/1/3 opcodes and integrated into the nML processor model, so that the synthesis check accounts for their effect on the decoder, register file, pipeline, and critical path. CIDRE reports estimated execution-time speedups ranging from 1.32× to a best of 2.47× on *eb-aes*, an average area overhead of 11.3 % (maximum below 24 %), and clock-period increases bounded between 0 % and 4.9 %. Power is not reported. The greedy selection with look-ahead does not guarantee a globally optimal instruction set, and the gap to the global optimum is not quantified. The flow depends on Synopsys ASIP Designer for the processor model used in the synthesis check and for the RTL generation, restricting reuse to an EDA environment with the same licences, and integrating compiler support is named as future work.

Where CIDRE automates discovery, selection, and hardware generation, RISQ-V [26] demonstrates what manual expert effort can achieve, addressing all four phases on CV32E40P. The authors hand-select twenty-nine custom instructions for Post-Quantum Cryptography (PQC), organised into seven classes covering the Number-Theoretic Transform (NTT), modular arithmetic, Keccak permutation, and binomial sampling. The instructions are added under a single unused R-type opcode (`0x77`). Compiler support is manual, the hand-designed instructions being invoked from the source as opposed to being emitted by an automatically retargeted compiler. The hard-

Table 2.1: Coverage of the four phases of custom-instruction extension by the surveyed works.

Work	Discovery	Selection	Spec. + Compiler	RTL gen.
ARISE [21]	✓			
Seal5 [22]			✓	
OpenASIP [9]			✓	✓
ADVICE [23]			✓	✓
CIDRE [24]	✓	✓		✓
RISQ-V (manual) [26]	✓	✓	✓	✓

ware support reuses existing processor resources. RISQ-V folds its polynomial MAC (`pq.mac`) into the existing CV32E40P multiplier, reusing the existing hardware path. The four accelerators, an NTT and modular-arithmetic unit, a Karatsuba/Toom-Cook multiplier, a Keccak round unit, and a binomial-sampling unit, are deeply integrated into the RISC-V pipeline. The Keccak state is mapped onto the floating-point register file plus part of the integer register file, a combination the authors call the Post-Quantum Register Set, to avoid a dedicated Block Random-Access Memory (BRAM). The evaluation uses UMC 65 nm with PULPino GCC 7.1.1 at `-O3`, on NewHope, Kyber, and Saber at their standardised parameter sets. RISQ-V achieves 7.7× to 11.4× clock-cycle-count speedup on Kyber and NewHope over the clean-C PQ-M4 reference, with Saber gaining less at 2.7×, and corresponding energy reductions per operation between 2.1× and 9.5×. The cell count rises 1.59×, and the maximum frequency drops 43 %, caused by the long critical path through the Modular Arithmetic Unit. Against assembler-optimised ARM Cortex-M4 implementations, RISQ-V reports clock-cycle speedups of 4.30× (NewHope-512), 4.46× (NewHope-1024), 2.89× (Kyber-512), 3.05× (Kyber-768) and 3.84× (Kyber-1024), and 1.16× to 0.97× on the Saber family. The twenty-nine instructions and four accelerators are the product of expert manual effort, so the engineering cost is high and the result does not generalise beyond the PQC workloads it targets. The reuse of the floating-point register file for Keccak state forces the programmer to accept that floating-point and PQC kernels cannot coexist easily on the same core. The headline speedups are also measured against a clean C reference rather than against an assembler-tuned RISC-V implementation, so part of the gain is attributable to compiler-level looseness and not to the hardware extension itself.

Table 2.1 summarises how the six works partition the four phases, and the coverage is uneven. The single-phase tools take one stage each, ARISE the discovery and Seal5 the compiler retargeting. OpenASIP and ADVICE both cover phases three and four from a manual input, differing mainly in how they integrate the result, OpenASIP through a coprocessor interface and ADVICE by merging the accelerator into the CPU with a single HLS pass. CIDRE automates the most, discovery, selection, and hardware generation, but leaves compiler retargeting to future work, while RISQ-V spans all four only by manual expert effort. No surveyed flow automates the four phases end to end.

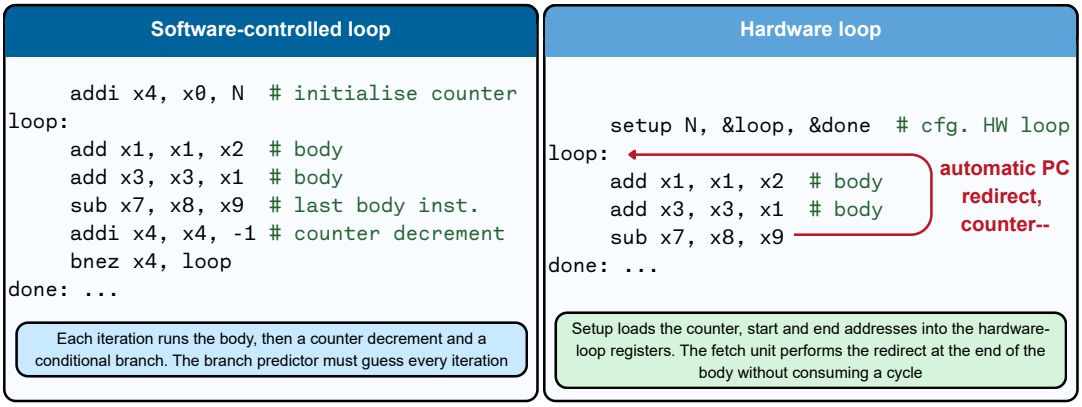


Figure 2.8: Software-controlled loop versus hardware loop. In the software case, every iteration ends with an explicit counter decrement and a conditional branch back to the start. In the hardware case, dedicated registers hold the counter and the start and end addresses; when the PC reaches the end address, the fetch unit redirects it to the start address and decrements the counter automatically, without consuming a clock cycle in the loop body.

2.3.2.2 Hardware Loops

While custom instructions add new computational capability to the core, hardware loops take a different route by adding a control-path mechanism instead. A hardware loop replaces the per-iteration overhead of a software-controlled loop with a small set of dedicated registers that the instruction-fetch unit consults at every clock cycle. The registers hold the loop count, the start address and the end address. When the PC reaches the end address with a non-zero counter, the fetch unit redirects the PC to the start address and decrements the counter. The body executes uninterrupted, and the branch predictor, the comparator and the decrement port of the integer unit are spared. Figure 2.8 contrasts the software and hardware-loop styles on a small example.

Hardware-loop implementations differ in their supported loop topology, in their maximum depth of nesting, and in their compiler support, all of which affect both the area cost and the workloads the unit can accelerate. The supported topology ranges from counted single-entry loops only to arbitrary irreducible multiple-entry, multiple-exit structures. The compiler support determines whether the unit is exposed only through hand-written assembly and intrinsics or is generated automatically by the compiler from regular C source.

Kavvadias and Nikolaidis [27] target the irreducible-loop axis. The Zero-Overhead Loop Controller sits at the instruction-fetch stage and supplies the next instruction address for any task in a Task Control-Flow Graph, a directed cyclic graph that uniformly represents arbitrary loop nests including irreducible regions with multiple entry or exit nodes. The same paper notes that earlier hardware-loop mechanisms, such as those in the Motorola DSP56300, ST120 and TMS320C54x DSPs and the Xtensa LX configurable processor, were limited to perfectly nested counted loops with a single entry, so ZOLC covers a strictly larger set of loop shapes.

Internally the unit comprises a task-selection lookup table, loop-parameter tables (initial, step, final values), and an index-calculation finite-state machine, with three new opcodes for initialisa-

tion. The authors evaluate three configurations that span nearly a 15× range in gate count. The minimal μZOLC supports a single loop and costs 298 NAND-equivalent gates plus 171 storage bits in 0.13 μm STMicroelectronics standard cells. ZOLClite supports the same eight-loop structure but without multiple-exit loops, so a loop can be left only from its tail instruction, at 4056 gates and 1552 storage bits. The full ZOLC variant adds multiple-exit support, with up to four entries and exits per loop, and reaches 4428 gates and 3088 storage bits. The maximum clock frequency, around 170 MHz, is unaffected by the addition.

The evaluation runs on XiRisc, a MIPS-I-compatible 32-bit RISC core. Programs are compiled with GNU Compiler Collection (GCC) at `-O3 -mips1` and the ZOLC initialisation sequence is inserted by post-processing the resulting assembly, since adding a new transformation pass to GCC was deemed too invasive. The same paper presents a Machine-SUIF and SALTO compiler-pass implementation as a proof of concept of full compiler integration, although the XiRisc benchmark figures rely on the post-processing approach. For two kernels (`rcdct` and `fsme_dr`), loop distribution is applied by hand to expose larger loop structures. On ten data-intensive kernels and ten MiBench applications, ZOLClite reduces execution clock cycles by 25.5 % on average for the kernels and 10 % for the applications, exceeding 40 % on `rcdct` and *matmult*. The minimal μZOLC recovers most of the kernel benefit at 17.5 % on average. The same kernels run on a Xtensa LX configuration with native zero-overhead loops (single-loop only) achieve only 3 % to 7.5 % improvement, because the Xtensa native loop unit cannot accelerate any loop body containing conditional control flow. ZOLC fitted to a custom DSP ASIP (hDSP) yields 44.15 % average speedup with extreme cases approaching 75 %. Whichever of the three configurations the architect selects, that choice is fixed at design time, with nothing in the methodology adapting it to the application.

Gautschi et al. [28] target the depth-of-nesting axis on an open-source RISC-V core. The RI5CY core (the name used in the PULP repositories, later renamed CV32E40P by the OpenHW Group) implements two hardware-loop register sets mapped into the CSR address space, each holding a loop counter, a start address and an end address. Dedicated setup instructions populate the registers from either an immediate value or a register operand. When the PC reaches the end address and the counter is non-zero, the fetch engine is redirected to the start address without executing a branch instruction, removing the compare-and-branch overhead of a software loop. Each register set costs around 1500 NAND-equivalent gates, and the authors evaluate the depth choice empirically before committing to two sets, on the grounds that longer nesting offers only marginal benefit for the targeted DSP kernels. The two-set choice is then fixed in the released silicon. Once fixed, the choice has the same area cost on every instantiated core, so workloads whose deepest convertible loop nests fewer than two levels carry the second register set as unused area, while workloads with deeper convertible nesting cannot exploit the additional levels at all. Unlike Kavvadias and Nikolaidis, Gautschi et al. deliver a modified GCC back-end that detects eligible loops and emits the hardware-loop setup instructions automatically, so the programmer does not have to mark loops by hand or post-process assembly.

The CV32E40P hardware-loop unit imposes specific constraints on which loops it can host [29].

The setup instructions and the body's start and end addresses must all be 32-bit aligned. The end address must be strictly greater than the start address and must point to the instruction just after the last one of the body. The body must contain at least three instructions. When two loops are nested, the outer loop's end address must lie at least eight bytes (two instructions) beyond the inner loop's end address. Control flow into the body is restricted to its start address, so no branch or jump from outside the loop may target an instruction inside the body, and the loop's own control CSRs cannot be modified from inside the body. Several instruction classes are forbidden inside the body, including compressed (RVC) instructions, branches and jumps, the memory-ordering instructions `fence` and `fence.i`, and the privileged instructions `mret`, `dret`, and `wfi`. The instructions `ebreak` and `ecall`, although privileged, are admitted as exceptions. Together these constraints define the set of convertible loops on which any methodology layered on the unit must operate.

On a benchmark suite that combines signal-processing kernels (FIR filters, matrix operations, convolutions and an FDCT) with cryptographic and control-intensive workloads, the addition of hardware loops together with post-increment addressing yields a 37 % average speedup over the baseline RV32IMC ISA. The paper does not isolate the hardware-loop contribution from post-increment, so the 37 % figure is the joint result for the two extensions.

Commercial products also adopt the two-level depth, as in Qualcomm's Hexagon V5 Very Long Instruction Word (VLIW) DSP [30], while other implementations opt for a single low-overhead loop instead. The Armv8.1-M Low Overhead Branch (LOB) extension on Cortex-M cores [31] accelerates only the innermost loop. Even with this single-loop limitation, Arm reports LOB-only speedups on Cortex-M55, obtained by recompiling the same code with and without LOB support. The per-kernel gains reach 5 % to 70 % on EEMBC TeleBench, which raises the suite total score by 10 %, and 4 % to 39 % on EEMBC FPMark-SP. The largest gains come from autocorrelation and inner-product kernels, where the loop body is small enough that the per-iteration branch overhead dominates [32].

What unites every hardware-loop implementation surveyed is that the topology and depth of the loop unit are fixed at design time, with nothing in any of the methodologies adapting the choice to the application. The architect selects one configuration and that choice propagates into every instance of the resulting core. Fixing one configuration is reasonable for a general-purpose part, but a workload-specific choice would be more efficient, matching the unit to the loop nesting each application actually contains.

With both the additive and subtractive axes now surveyed, the next section collects the dimensions of the chapter into a single comparison and identifies the research opportunity that the survey leaves open.

2.4 Synthesis and Research Opportunity

Two forms of processor specialisation were surveyed in the previous section, subtractive and additive. The synthesis below collects the works on each axis into a comparison table, draws out

Table 2.2: Subtractive specialisation works.

Work	Target core	Input artefact	Abstraction level	Direction	Headline result
Cherupalli et al. [16]	openMSP430	Single binary	Gate netlist	Top-down	62% area, 50% power (avg)
Bleier et al. [17]	Ibex, Cortex-M0, RIDECORE	ISA subset	Gate netlist	Top-down	14% to 18% gates, up to 23% area
Si et al. [18]	MIPS (non-pipelined)	Single binary	C, RTL, gate	Top-down	71.0% area, 62.1% power
Chaidos et al. [19]	Zero-Riscy, TP-ISA (printed)	Single binary	RTL	Top-down	10.6% area, 11.4% power
Raisi-Ardali et al. [20]	RV32E (flexible)	Binary-derived block list	RTL blocks	Bottom-up	8% to 43% area, 3% to 30% power

the dimensions on which the works converge or differ, and identifies the research gaps the survey leaves open and the opportunity the chapters that follow take up.

Each of the five subtractive works in Table 2.2 carries a different scoping decision that limits where the same approach can be carried over. Cherupalli et al. apply X-propagation at the gate netlist of an in-order openMSP430, acknowledging that caches, branch prediction, and out-of-order execution introduce non-determinism into the instruction stream that the analysis could absorb at the expense of analysis runtime, an extension they leave as future work. Bleier et al. reuse a once-authored SVA property library across all cores implementing the same ISA, but the retained ISA subset is selected by hand, since the framework offers no extraction of it from a workload binary. Si et al. require the host processor itself to be written in synthesisable C, which their HLS-driven flow needs but which restricts it to simple non-pipelined cores. Chaidos et al. work at the RTL level but through a designer-driven pass rather than a packaged automated tool, so each new workload repeats the manual effort. Raisi-Ardali et al. build single-cycle combinational RTL suited to the sub-megahertz frequencies of flexible electronics, a microarchitecture not demonstrated on the multi-stage pipelined silicon where competitive $F_{\max}$ depends on pipelining.

The six custom-instruction works in Table 2.3 each carry a different gating constraint on how broadly the method generalises. ARISE discovers candidates with a greedy heuristic that offers no optimality guarantee and stops at CoreDSL output that a downstream toolchain must consume. Seal5 retargets the compiler automatically but discovers nothing itself, so the designer hand-authors each instruction and supplies its RTL separately. OpenASIP covers specification, compiler retargeting and RTL generation but leaves selection to the designer, and its coprocessor interface loses on short fused operations. ADVICE covers the same phases at HLS, exposing cross-boundary co-optimisations but constraining the base core to be HLS-friendly. CIDRE is the only flow to automate discovery, selection, and RTL generation together, but it depends on

Table 2.3: Custom-instruction extension works.

Work	Kind	Phases addressed	Target core	Headline result
ARISE [21]	Discovery tool	1	RV32 assembly	7.39% avg dyn. instr. count (proxy)
Seal5 [22]	Compiler retargeting tool	3	CV32E40P + Core-V	24% avg, 86% peak with SIMD
OpenASIP [9]	Multi-phase toolchain	3, 4	CVA6, Rocket	27% to 40% avg execution time
ADVICE [23]	HLS-based multi-phase toolchain	3, 4	Comet (5-stage RV)	27× avg speedup, 78% area overhead
CIDRE [24]	End-to-end automated flow	1, 2, 4	trv64p3 (RV64IM, 3-stage)	2.47× best on *eb-aes*
RISQ-V [26]	Manual case study	1, 2, 3, 4	CV32E40P	7.7× to 11.4× clock-cycle-count speedup

Synopsys ASIP Designer, leaves compiler retargeting to future work, and evaluates on a custom three-stage RV64IM core, not a commodity RISC-V core. RISQ-V proves what an expert team can achieve on CV32E40P, at an engineering cost that does not transfer to other workloads and a 43 % frequency drop in the resulting silicon.

The four hardware-loop implementations in Table 2.4 differ in topology, nesting depth and compiler integration, but each carries a constraint that limits how the same choice could be made differently. Kavvadias and Nikolaidis size the controller for a chosen number of irreducible loops at design time, with the evaluated full configuration handling an eight-loop structure, and although they demonstrate a Machine-SUIF and SALTO compiler flow as a proof of concept, the reported XiRisc figures instead rely on manual assembly post-processing, so the automated path was never exercised end-to-end for the published results. Once the design is synthesised the loop capacity and topology are fixed in the resulting silicon. Gautschi et al. commit to two levels of counted single-entry loops in the PULP RI5CY silicon and report a 37 % average speedup that bundles the hardware-loop benefit with post-increment loads and stores, so the isolated hardware-loop contribution is not separable from the bundled measurement. Qualcomm's Hexagon V5 carries the same two-level counted choice, but the published results do not isolate the hardware-loop contribution from the rest of the VLIW DSP, so no separate hardware-loop figure can be extracted. Armv8.1-M LOB on Cortex-M55 supports a single loop only, so any kernel containing nested loops falls back to software-managed loops.

Although the works target different problems, each axis still shares one metric on which they can be placed side by side. Among the subtractive works that metric is the relative area reduction against each work's own baseline, and it orders them by how aggressively they specialise. The single-binary gate-level flows remove the most (Si et al. 71 %, Cherupalli et al. 62 %), the bottom-up and RTL-level flows remove less (Raisi-Ardali et al. 8 % to 43 %, Chaidos et al. 10.6 %), and the ISA-subset flow removes least (Bleier et al. up to 23 %), with the four works that also report

Table 2.4: Hardware-loop implementations.

Implementation	Topology	Nesting depth	Compiler integration	Headline result
Kavvadias and Nikolaidis [27] (ZOLC)	Irreducible, multi-entry, multi-exit	Up to eight loops	Machine-SUIF + SALTO pass (proof of concept)	25.5% kernels avg, 10% MiBench apps
Gautschi et al. [28] (PULP HWLoop)	Counted, single-entry	2 levels	Modified GCC, automatic loop detection	37% avg speedup (with post-increment)
Qualcomm Hexagon V5 [30]	Counted, single-entry	2 levels	Commercial Qualcomm toolchain	Bundled with whole VLIW DSP
Armv8.1-M LOB [31] on Cortex-M55 [32]	Counted, single-entry	1 level (single loop)	Arm Compiler AC6.16, automatic loop detection	5% to 70% EEMBC TeleBench, 4% to 39% FPMark-SP

power following the same order from 11.4 % to 62.1 %. Among the custom-instruction works the nearest shared metric is the speedup, though it is not reported on a common basis. Even so, it separates the lightweight fused-instruction tools (CIDRE 1.32× to 2.47×, OpenASIP roughly 1.37× to 1.67×) from the accelerator-style designs (RISQ-V 7.7× to 11.4×, ADVICE 27×) that buy their larger gains with much larger area. The hardware-loop works compare on per-kernel clock cycle reduction, from 25.5 % for ZOLC and 37 % for the PULP core, bundled with post-increment, to as much as 70 % on a single kernel for Armv8.1-M LOB.

What these comparisons cannot be is absolute. The technologies range from a 7 nm predictive PDK to printed and 0.6 μm flexible substrates, so a gate count, a μm^2 area or a clock frequency is not commensurable across rows, and the figures above are read as relative gains over each work's own baseline rather than as a single ranking. Two metrics also fall outside any common comparison for want of data, not relevance. Power or energy is reported by only seven of the works (Cherupalli et al., Si et al., Chaidos et al., Raisi-Ardali et al., Gautschi et al., ADVICE and RISQ-V), with OpenASIP and CIDRE stating outright that they do not measure it, and frequency by fewer still, while ARISE and Seal5 stop at code-density or clock-cycle-count proxies. Beyond these individual omissions, no surveyed work reports an extensive evaluation across both an ASIC and an FPGA flow, measuring the effect of each specialisation on area, power and timing together with the resulting change in clock-cycle count.

Read together, the limitations of both forms expose a structural pattern. Chaidos et al. are the only published work to combine subtractive with additive specialisation, but both phases are designer-driven and the target is a printed-electronics process rather than silicon. Where end-to-end automation does exist it covers only one form, with CIDRE automating custom-instruction extension but performing no pruning and no compiler retargeting, and the subtractive flows of Cherupalli et al. and Bleier et al. automating pruning without adding anything to the core. No published flow automates the combination of these forms, nor lets the application binary itself drive all of the design parameters at once.

The opportunity that follows is an end-to-end automated flow that takes application source code as input and produces a customised core as output, combining pruning of unused logic with custom-instruction addition and per-application selection of the hardware-loop nesting depth, all driven by the workload itself rather than by an ISA subset or a designer-authored specification. It runs entirely on a standard open-source toolchain, so the engineering effort is not gated by commercial licences, and it targets a commodity pipelined silicon-class RISC-V core over a flexible-electronics, printed-electronics, or non-pipelined target.

Chapter 3

A Top-Down Methodology for Single-Program Hardware Specialisation

3.1 Methodology Overview

This chapter defines a methodology for specialising a baseline RISC-V core to a single workload. The workload is profiled to establish what the processor must actually support, and which optimisations could be applied to the core's RTL.

The methodology operates directly on the core's RTL sources, enabling each architectural modification to be traced back to the workload characteristics that motivated it. The source code can be transformed both by removing existing logic and by introducing new functionality where beneficial. Because the transformations are applied before synthesis, the downstream tools still operate on ordinary RTL, which keeps the flow compatible with conventional FPGA and ASIC implementation steps and leaves synthesis free to optimise the specialised design further.

Figure 3.1 sets out the end-to-end flow, which proceeds in three stages. In the first, the workload's source code is compiled to a baseline binary, and that binary is profiled to extract statistics about the program, such as the record of the hardware actually exercised, that will be the base of later optimisation decisions. In the second stage, the profile and the baseline RTL description of the core are provided to a set of optimisation strategies. Each strategy analyses the profile and determines what modification, if any, is beneficial, and these decisions are then realised as edits to the RTL sources. The strategies operate independently, allowing them to be applied individually or composed to produce a single specialised processor variant. In the third stage, the resulting design is generated as synthesis-ready RTL, suitable for a conventional FPGA or ASIC implementation flow.

The methodology investigates three optimisation strategies that transform the core's RTL. Each strategy addresses a distinct source of inefficiency identified by the workload profile, resulting in a modified RTL implementation of the baseline processor.

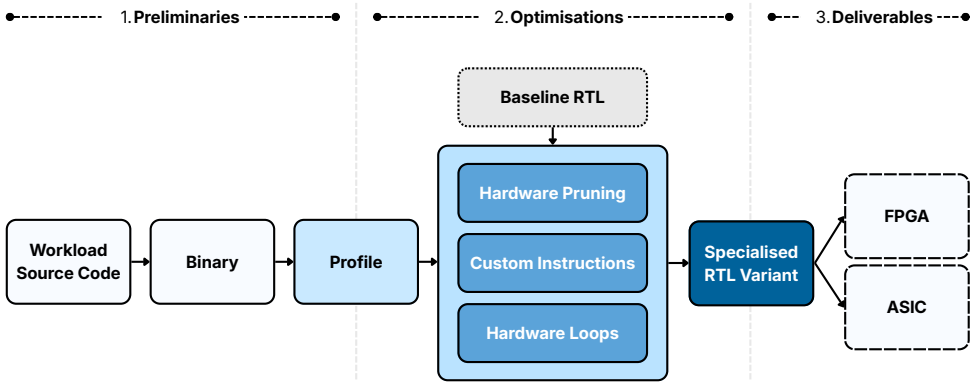


Figure 3.1: End-to-end methodology flow. The workload's source code is compiled to a baseline binary and profiled to extract its hardware requirements. The profile drives three independent transformations of the RISC-V core (pruning, custom instructions, and hardware loops), which can be applied individually or in any combination. The result is a specialised RTL core ready for downstream FPGA or ASIC synthesis.

- *Hardware pruning* removes logic that the binary does not require and resizes parameterised components whose configured dimensions exceed the workload's needs.
- *Custom instructions* identify recurring short instruction sequences and replace selected occurrences with newly defined instructions that compute the same value in one operation.
- *Hardware loops* replace software-managed loop control with dedicated loop hardware, reducing the instruction overhead associated with loop control.

Alongside these, a *hyperparameter-tuning* mechanism was also studied, a search that turns a core's configurable size knobs, such as buffer depths, towards a chosen target function on an already-built RTL core. It is fully implemented in the toolchain, but on the target core its only such knob is the prefetch-buffer depth, whose measured benefit proved too marginal to justify a place in the main evaluation. Its methodology, implementation, and results are therefore given in Appendix A, and the main text fixes that knob at the shipped default.

Each of the three transformations produces an RTL variant that can be evaluated in isolation. Additional variants are obtained by composing them, ranging from a processor specialised through hardware pruning alone to one incorporating all three transformations. The preferred variant is not fixed by the methodology itself, since it depends on the metric chosen for evaluation, such as area, power, speedup, or a combined score. The remainder of this chapter defines the three transformations in turn, beginning with hardware pruning.

3.2 Hardware Pruning

Pruning removes logic from the baseline core that is not required by the target application and would otherwise stay idle during execution, reducing the circuit area to the minimum the workload

needs. This reduction in area could potentially lead to improvements in timing and energy consumption. The opportunity exists because a general-purpose core implements a broad instruction set and a range of optional or configurable features so that it can serve many different programs, and any single program draws on only part of that generality. Some logic cannot be removed, but can be resized to a minimum value, which could also save resources.

The pruning pipeline proceeds in three stages. The compiled binary is analysed to obtain the workload's hardware requirements. These requirements determine a pruning configuration that records, for each feature of the core, whether it is retained or removed, and for each configurable parameter, the smallest value that still satisfies the workload. The configuration is then applied to the core's RTL sources through transformation mechanisms chosen according to how the affected logic is embedded in the source. Finally, a cleanup pass ensures there are no leftovers from the removal steps.

3.2.1 Identifying Removable Logic

For pruning, the core is described as a set of *features* and *parameters*. A feature is a unit of hardware that can be conditionally retained or removed. It is most often a complete functional block, such as a multiplier, a divider, a floating-point unit, or a debug or interrupt subsystem, but it may also be a smaller element contained within a larger block, such as a single operation in the ALU, an individual instruction handled by the decoder or an entry in the register file. A parameter is a configurable dimension of a component, such as the width of the PC or the depth of an internal buffer. A parameter is not removed but resized to the smallest value that still satisfies the workload.

Each feature and each parameter is associated with a *trigger* that maps the workload's requirements to a decision. A feature trigger is a Boolean predicate over the workload, that states whether the feature must be retained or removed. A parameter trigger evaluates instead to a numeric value derived from the workload, such as the number of bits needed to address the executable range the program occupies.

These triggers and the configuration they produce admit a precise representation. Let $\mathscr{F}$ be the set of features the core exposes and $\mathscr{V}$ its set of parameters, and let W denote the workload. Each feature $f \in \mathscr{F}$ carries a trigger τ_f and each parameter $v \in \mathscr{V}$ a trigger ϕ_v,

$$\tau_f : W \mapsto \{0,1\}, \qquad \phi_v : W \mapsto \mathbb{N}, \tag{3.1}$$

where a feature trigger maps the workload to the Boolean set $\{0,1\}$, with $\tau_f(W) = 1$ retaining the feature and $\tau_f(W) = 0$ removing it, and a parameter trigger maps the workload to a natural number $\mathbb{N}$, the value the parameter is to take. Evaluating every trigger yields the pruning configuration

$$C = (\Psi, \Phi), \qquad \Psi(f) = \tau_f(W), \quad \Phi(v) = \phi_v(W), \tag{3.2}$$

in which $\Psi : \mathscr{F} \to \{0,1\}$ records the retain-or-remove decision for every feature and $\Phi : \mathscr{V} \to \mathbb{N}$ the chosen value for every parameter. The pruned core is the result of a transformation T that

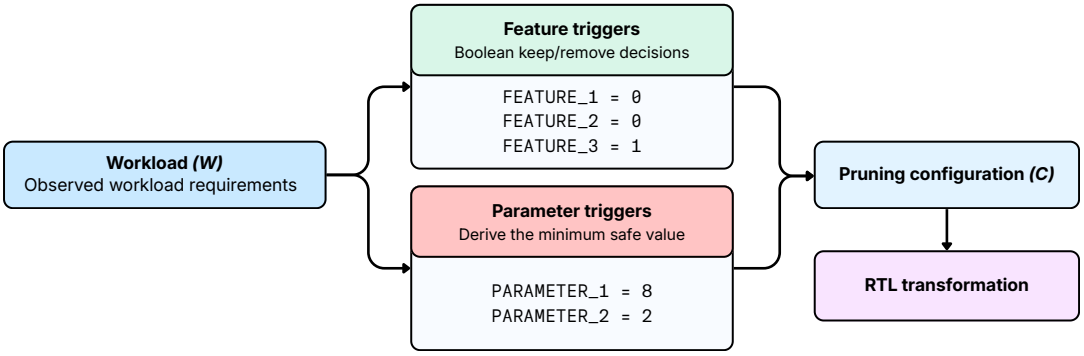


Figure 3.2: Hardware-pruning trigger model. The workload W feeds two classes of pruning triggers. Feature triggers produce Boolean keep/remove decisions, while parameter triggers derive minimum safe parameter values. Together, these decisions form the pruning configuration C, which is applied to the baseline RTL.

realises the configuration on the baseline sources,

$$\mathrm{RTL}' = T(\mathrm{RTL}, C), \tag{3.3}$$

where RTL denotes the baseline core's sources, C the configuration, and RTL' the specialised sources that T produces. Figure 3.2 summarises this pruning decision model, and Figure 3.3 works a small illustrative example through it.

The rest of the pruning methodology therefore has two tasks, constructing the workload profile used by the triggers, and defining the RTL transformation mechanisms, together with their cleanup passes, that realise the resulting configuration.

For each feature and parameter, the trigger condition and the location of the affected logic within the core must be defined in advance, which is a one-time effort per target core. The two parts differ in how far they carry across cores. A trigger that derives from the base RISC-V ISA or a standard extension is portable, because it identifies the same logical capability on any compliant implementation. The trigger for integer division, for instance, denotes the same feature regardless of the core. However, the location of that capability in the source remains implementation-specific and must be established for each core.

3.2.2 Binary Usage Analysis

The triggers are evaluated against the compiled binary, so the profile must carry whatever evidence those triggers consume. A number of observables are common to any RISC-V program. The set of instructions it executes, the registers it reads or writes, and the extent of its executable address range, which bounds the program counter, are all recoverable directly from the instruction stream and the section layout, because the opcode and operand fields are explicit in the RISC-V encoding. Other observables are present only when the core includes extra features or parameters. Consequently, the exact contents of the profile follow from the trigger set defined for a particular core.

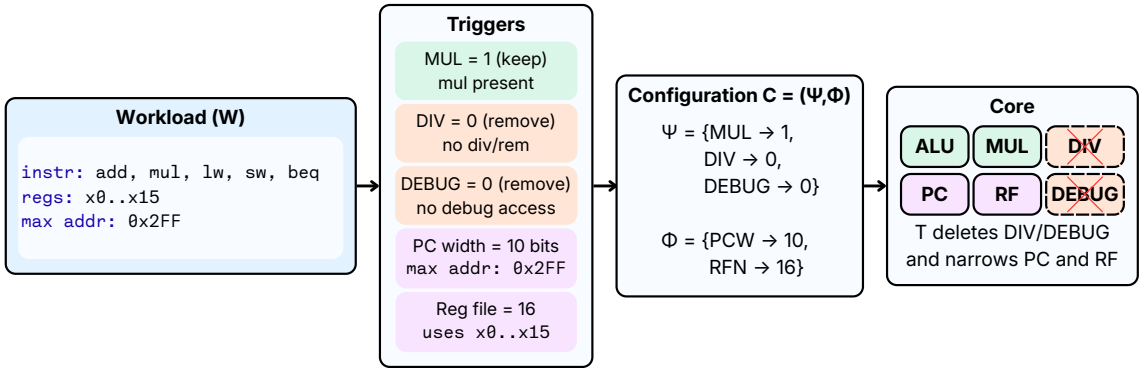


Figure 3.3: An illustrative pass of a small workload through the pruning model. The feature triggers keep the multiplier and remove the unused divider and debug unit, while the parameter triggers size the program counter to ten bits and the register file to sixteen entries. The resulting configuration $C = (\Psi, \Phi)$ drives the transformation T, which deletes the removed blocks and narrows the resized ones. The values are illustrative and not tied to any particular core.

This collection of observables constitutes the workload profile, the representation of the workload W that every trigger reads.

3.2.3 RTL Removal Mechanisms

Depending on the features or parameters a core exposes, realising the transformation T of Equation 3.3 on the RTL may require more than one kind of edit. A well-separated block can be excluded as a whole, logic buried inside a construct that must otherwise remain intact has to be edited in place, and logic spread across multiple interdependent control paths cannot be expressed by either of the first two. The transformation T is therefore the composition of complementary mechanisms, each suited to one of these forms and applied to the entries of C whose logic takes the corresponding shape.

3.2.4 Post-Removal Cleanup Passes

After unused logic is removed, the modified source may still contain artefacts of the original design. Removing a block can leave declarations and assignments with no remaining consumer, and reducing the set of values a signal can take can leave its encoding wider than the survivors require. Downstream synthesis may remove some of this redundancy on its own, yet resolving it at the source keeps the delivered RTL faithful to the specialised design and lets the saved resources show through in later analysis. Consequently, the pruning pipeline closes with a cleanup step that clears these leftover artefacts. Once the cleanup has run, the modified RTL is the deliverable of the pruning flow, ready for whichever downstream synthesis or simulation flow is chosen.

Pruning removes or minimally sizes resources whose need can be inferred from functional workload evidence. The next strategy instead adds to the core, extending its instruction set so that recurring operation sequences can be carried out by new custom operations.

3.3 Custom Instructions

A RISC architecture is built from a small set of simple, highly optimised instructions, each performing one elementary register-transfer operation, in contrast to a Complex Instruction Set Computer (CISC) architecture whose instructions may carry out multiple such operations at once. Compiled code on a RISC core thus expresses a larger computation as a sequence of these elementary operations from the base ISA. A general-purpose ISA cannot afford a dedicated instruction for every useful composition, but a single workload exercises only a few of them. Custom-instruction fusion exploits this. It moves selectively in the CISC direction, introducing compound instructions, but only for the short operation sequences the target program actually uses. When a specific program executes the same short sequence whose instructions are linked by data dependencies, the result of one feeding the operands of a later one in the sequence, the sequence can be replaced by a single custom instruction, constrained to the core's single-cycle arithmetic path, that computes the same composed value.

This work uses a bounded form of fusion. Instead of looking for the most profitable groups among all the ways instructions in a program depend on one another, it considers only short chains in which each instruction feeds the next, and only those that fit the chosen core and instruction format. Two conditions decide which chains are admissible. A fused instruction must compute its result within the time the core already allows for a single-cycle arithmetic operation, so the pipeline timing is preserved and no extra stall logic is needed. The chain must also be short, for two reasons. The instruction format offers a fixed number of operand positions, so a chain that names more operands than the format can carry cannot be encoded, and each extra operation lengthens the path the result must travel within one clock period, so an over-long chain lowers the frequency the core can still meet.

To formalise this idea, let a fusion candidate be an ordered sequence of base instructions

$$s = (i_1, i_2, \ldots, i_n), \tag{3.4}$$

of length n, in which the result of each instruction may feed the operands of those that follow. A *pattern* is the operation that a sequence computes once its specific register names and constants are abstracted away, obtained through a pattern function

$$p = f(s), \tag{3.5}$$

so that two sequences differing only in their register allocation yield the same pattern p. *Fusion* is the operator that replaces an admissible sequence with a single instruction computing its composed result,

$$F(s) = i^{\star}, \tag{3.6}$$

where the fused instruction $i^{\star}$ is constrained to the core's single-cycle arithmetic path. Replacing one occurrence of a sequence of length n removes $n-1$ of its n issued instructions, so the per-

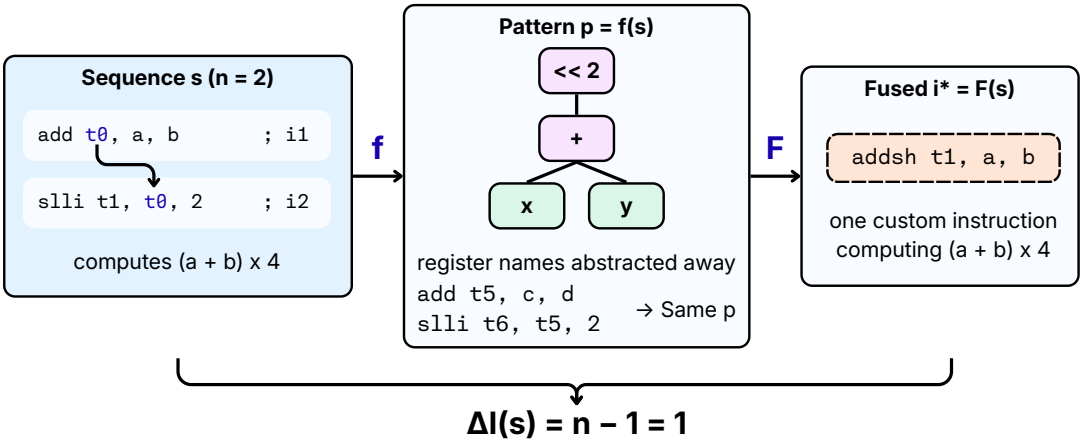


Figure 3.4: An illustrative fusion of a two-instruction sequence. The dependent pair `add` then `slli`, computing $(a+b)\times 4$, is abstracted by the pattern function f into the operation $(x+y)\ll 2$, so any occurrence with different registers yields the same pattern p. The fusion operator F then replaces it with a single custom instruction $i^{\star}$. The sequence has length $n=2$, so the saving is $\Delta I(s)=n-1=1$ instruction per occurrence. The mnemonic shown is illustrative.

occurrence instruction-count saving is

$$\Delta I(s) = n - 1. \tag{3.7}$$

This is the static reduction in instruction count, the number of instructions removed from the program image for that occurrence. The same value serves as a static clock-cycle-saving estimate under a one-instruction-per-cycle model of the replaced sequence, since each removed instruction is one fewer both stored and, when the sequence executes, issued. Figure 3.4 works a small illustrative example through these definitions.

The flow is then defined in four steps. The custom-instruction encoding space available to the core is fixed first. The candidate patterns are then discovered and their occurrences counted, the subset that fits the encoding budget and yields the greatest estimated saving is selected, and the selected operations $F(s)$ are connected to the core's logic.

3.3.1 Encoding Space and Compiler Integration

Before any pattern can be discovered, two design decisions must be settled. The first fixes how many fused instructions a single binary can hold, since the encoding space available for custom instructions on a fixed-width ISA is finite. The second fixes how the compiler is told about the chosen patterns, so that the binary the discovery analysed can be recompiled to actually emit the new instructions.

The RISC-V specification reserves `CUSTOM` opcode space for implementation-defined extensions [6], and a fused instruction occupies one encoding within it. Because the reserved space is finite, only a bounded number of fused instructions can coexist in one binary, a capacity denoted

K in Equation 3.8. Both the encoding format adopted for fused instructions and the resulting value of K depend on the target core.

Emitting the new instructions, and for some flows discovering them, is given to the compiler. Reusing the compiler's existing program analysis avoids rebuilding the same machinery and keeps the flow automatic, with no hand-written assembly for each workload. The methodology asks only that the chosen compiler can be extended to recognise a fused pattern and emit its custom instruction in place of the original sequence, so that adapting to a new workload is a matter of supplying the discovered patterns and recompiling.

3.3.2 Pattern Discovery

How a candidate pattern is found depends on the kind of operands it involves. Sequences built only from register-register operations, where each instruction reads register operands and writes one, are independent of the constants a workload uses and can be enumerated in advance from the core's operation set, so the compiler itself performs both their discovery and their emission. Sequences that include an immediate operand depend on the specific constant values the workload uses, which are unknown until the program is examined, so they are discovered by analysing the compiled binary, and later exposed to the compiler. In either case the compiler makes the final decision, emitting a custom instruction in place of a sequence during code generation only where doing so is possible. The two cases are treated separately below.

3.3.2.1 Register-Register Patterns

Register-register patterns are enumerated from the set of operations the core can chain within one clock cycle, which excludes loads, stores, branches, and any arithmetic operation the core does not complete in a single clock cycle, such as division. Within the surviving operations, two further restrictions apply. Patterns that involve a multiplication are admitted only when the multiplier in the target core already exposes a datapath that computes the composed expression of the pattern, so that no new multiplier datapath is introduced. The remaining single-cycle ALU operations are admitted in full.

Each ordered pair of admitted operations forms a forward pattern, where the result of the first feeds the first operand of the second. When the second operation is non-commutative, the result of the first may instead feed its second operand, producing a distinct composed value. The pair of an add followed by a subtract illustrates the distinction, since the forward variant computes $(x_1 + x_2) - x_3$ whereas the reverse variant computes $x_3 - (x_1 + x_2)$, as Figure 3.5 shows.

Forward patterns therefore contribute one entry per ordered pair of admitted operations, and reverse patterns contribute one additional entry per non-commutative second operation. The total count for a target core depends on the number and commutativity of its admitted operations and on how many multiplier compositions its existing datapath supports.

Because this catalogue is workload-independent, the candidate patterns are enumerated once and then matched against the workload during compilation, and counting how many times the

```
# forward:  x8 = (x1 + x2) - x3
add  x7, x1, x2
sub  x8, x7, x3

# reverse:  x8 = x3 - (x1 + x2)
add  x7, x1, x2
sub  x8, x3, x7
```

Figure 3.5: Forward and reverse variants of an add followed by a subtract. The intermediate value in `x7` feeds the minuend in the forward variant and the subtrahend in the reverse variant.

compiler emits each pattern gives the static occurrence count that the selection step of Section 3.3.3 uses.

3.3.2.2 Immediate Patterns

Unlike the register-register patterns, immediate patterns, those that involve a constant operand, are workload-dependent. The sequence of operations may be known in advance, but the constants appearing in those operations are properties of the compiled program. A pattern such as `addi` followed by `slli`, for example, may occur with many different constant values across different workloads, and those values influence both the instruction encoding and the hardware required to realise the fused operation. Immediate patterns cannot, as a consequence, be enumerated completely from the core's operation set and must instead be discovered from the workload.

Discovering these patterns from the binary is possible because the explicit operand encoding of RISC-V, in which every instruction names its register operands and data flows through registers under a load-store model [13, ch. 2], exposes the dependencies between operations by tracking register definitions and uses alone. The discovery does not enumerate arbitrary subgraphs of these dependencies, as the general custom-instruction problem would. It follows them only along short linear chains, where each operation feeds a later one in the chain. An immediate-pattern candidate is one such data-dependent chain of bounded length, at least one of whose operations consumes a constant.

An immediate-pattern candidate inherits the general fusion constraints stated at the start of this section, since it too becomes one fused instruction. The register-register patterns are discovered by the compiler, which guarantees that they lie in a single straight-line region with no branch targeting an instruction other than the first, and that their intermediate value is not read afterwards. An immediate-pattern candidate, by contrast, is found in the binary and must be verified against these conditions explicitly. The straight-line requirement keeps a branch from entering the fused instruction partway through, at a program point that no longer exists separately. The liveness requirement keeps a later read from seeing a stale value, since fusion drops the values the chain produces only for its own use rather than writing them to the register file. Because such a value may be read on a later iteration of an enclosing loop, it is decided from program-wide data-flow information.

An admissible candidate is then characterised by how its immediate value is distributed across its occurrences, since this determines how the constant must reach the fused instruction. When a pattern always uses the same constant, that constant is a property of the pattern itself and need not be transported by the encoding, so the fused instruction can carry one fewer operand and free an encoding slot. When the constant varies between occurrences, the encoding cannot carry a full machine word, so the constant must be delivered indirectly, which bounds how many distinct values such a pattern may take. The concrete scheme by which a fixed constant is embedded and a varying one is delivered is one realisation among several possible.

3.3.3 Pattern Selection

Discovery may yield more patterns than the encoding capacity K can hold. Let $\mathscr{P}$ be the set of discovered patterns and $\mathrm{occ}(p)$ the occurrence count of pattern p. This count may be static, the number of times the pattern appears in the program image, when the objective is code-size reduction, or dynamically weighted by execution frequency, when the objective is runtime reduction or another type. The two agree on which patterns matter only when occurrences are evenly executed. When $|\mathscr{P}|$ exceeds K, a selection retains the subset $\mathscr{S}^\star \subseteq \mathscr{P}$ of greatest total benefit,

$$\mathscr{S}^\star = \arg \max_{\mathscr{S} \subseteq \mathscr{P},\, |\mathscr{S}| \le K} \sum_{p \in \mathscr{S}} \mathrm{occ}(p)\,\big(n_p - 1\big), \tag{3.8}$$

where n_p is the length of the sequence that pattern p abstracts, so that $\mathrm{occ}(p)\,(n_p - 1)$ is the total saving the pattern contributes under a one-instruction-per-cycle assumption, by Equation 3.7. This work uses the static occurrence count, so the objective ranks patterns by the static instruction reduction they yield, a direct measure of code-size saving and a static proxy for clock cycle saving, with the realised clock cycle effect measured empirically afterwards. The objective is separable, since each pattern contributes its benefit independently of which others are retained, and the only constraint is the cardinality bound $|\mathscr{S}| \le K$. Selecting the K patterns of greatest individual benefit thereby solves this static objective exactly, so a greedy ranking by $\mathrm{occ}(p)\,(n_p - 1)$ is an exact maximiser of it. Figure 3.6 shows this rank-and-cut on a small illustrative set.

The result is nonetheless only an approximation of the clock-cycle-optimal selection, since the static count is a proxy for the dynamic saving and not the saving itself. Other formulations could instead weigh each pattern dynamically, or by its effect on the core's area, power, or maximum operating frequency, retaining the patterns whose hardware cost is best justified by their benefit. A cost term that couples patterns, such as a shared area budget, would turn the selection into a knapsack-style problem for which the same greedy ranking is no longer exact even for the static objective, but the static cardinality-bounded objective used admits the exact solution above.

3.3.4 Custom-Instruction Hardware Integration

Once the patterns to fuse have been discovered, the core's RTL sources must be extended so that the new instructions decode and execute correctly. The extension is generated automatically for

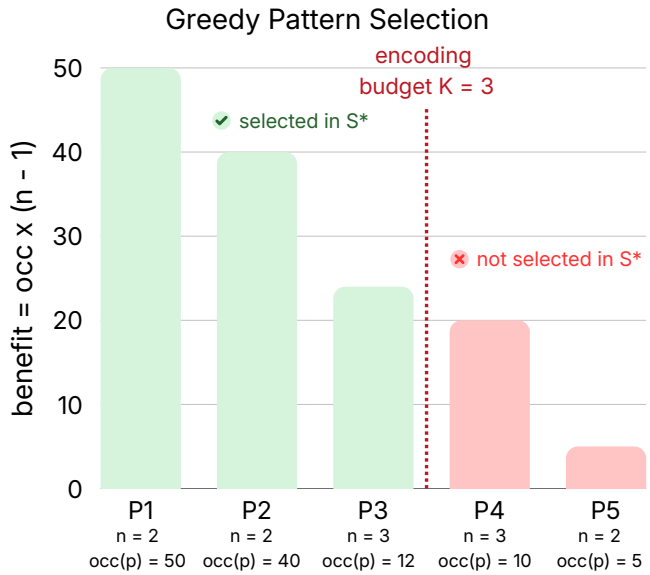


Figure 3.6: An illustrative ranking of five discovered patterns by their benefit $\mathrm{occ}(p)\,(n_p - 1)$, the static objective of Equation 3.8. With an encoding budget of $K = 3$, the three highest-benefit patterns are selected into $\mathscr{S}^{\star}$ and the remaining two are dropped. The values are illustrative.

each pattern and inserted at specific points in the core's sources, so the manual effort is confined to identifying those points once instead of coding each instruction by hand.

The first point is the decoder. The core fetches an instruction and decodes it to raise the control signals that steer the rest of the pipeline, and a custom encoding means nothing to the core until the decoder is taught to recognise it. Generating the decode logic for a fused instruction thus requires knowing how the core's control signals are defined and how its decoder is organised, so that the generated entry recognises the new encoding and asserts the same control the original sequence would have produced. This knowledge is what lets the logic be generated and placed in the right part of the decoder automatically. The second point is computation. Once recognised, the fused instruction must produce its composed result, and thus the pattern's operation is mapped to an expression in RTL that is added to the core's arithmetic path.

Exactly how these two points are realised depends on the core's organisation, since the way the decoder is structured, the way control reaches the datapath, and the combinational structures already present all differ between cores. The integration sites, the form each edit takes, and the reuse opportunities available are therefore core-specific.

3.4 Hardware Loops

Hardware loops specialise repeated control flow instead of arithmetic computation. In a software-controlled counted loop, each iteration executes both the useful body instructions and the loop-control sequence that updates the loop state and redirects control to the next iteration. The control

sequence performs no useful work, yet it runs on every iteration, so its cost accumulates in proportion to the iteration count and weighs most on the tight, frequently executed loops.

A hardware-loop unit moves this repeated control into the processor. The unit is a dedicated block of logic in the core that holds a loop's boundaries and remaining iteration count in private registers and redirects the program counter from the end of the body back to its start while iterations remain. To use it, the program first executes a setup sequence that loads those boundaries and the iteration count into the unit. The unit then redirects control around the body without a software back-edge branch, so the branch and the counter maintenance no longer need to appear in the body.

The optimisation problem is twofold. It is posed for a given target core together with the hardware-loop unit it carries, since both the loops the unit can track and the way they are set up are properties of that particular core. The software side selects which loops to convert and emits the setup sequence each one requires. The hardware side selects a configuration of the unit that supports the selected loops. That configuration includes the loop depth, which determines how many hardware loops may be active at once, and may also include the widths of counters, address fields, or other parameters the unit exposes. Hardware-loop specialisation is thus formulated as a joint loop-selection and hardware-configuration problem, developed in the subsections that follow.

3.4.1 Candidate Loops and Setup Information

Let W be the workload and $\mathscr{L}_W$ the set of loop candidates extracted from it. Each candidate ℓ carries a start address a_{start}, an end address a_{end}, an iteration count N, and a dynamic entry count E. The first three configure the unit and must be available, or cheaply derivable, before the loop is entered, so a candidate whose iteration count cannot be resolved at that point cannot be converted. The fourth value E, the number of times the loop is entered during execution, plays no part in configuring the unit and is not required in advance. It is a measure of how often the loop runs, used later to weigh how much a conversion is worth.

The candidate set is not fixed by the program text alone, since the compilation that produces the binary reshapes the control flow, alters iteration counts, and decides whether a loop appears as a closed body at all. A compiler pass aimed at the loop unit can therefore enlarge the candidate set, and the body of each function may be drawn from whichever compilation exposes the better candidates.

Not every candidate can be tracked, because the unit accepts only bodies that meet its restrictions. Consequently, each hardware loop unit supplies an admissibility predicate

$$A_{\text{tgt}}(\ell, \theta_{\text{HL}}) \in \{0, 1\},$$

which is true when loop candidate ℓ can be implemented as a hardware loop on the target core under hardware configuration θ_{HL}. The predicate is target-specific. It may depend on the loop-unit interface, the setup-instruction format, the way loop boundaries are represented, the supported instruction encodings, and any restriction the core imposes on the body.

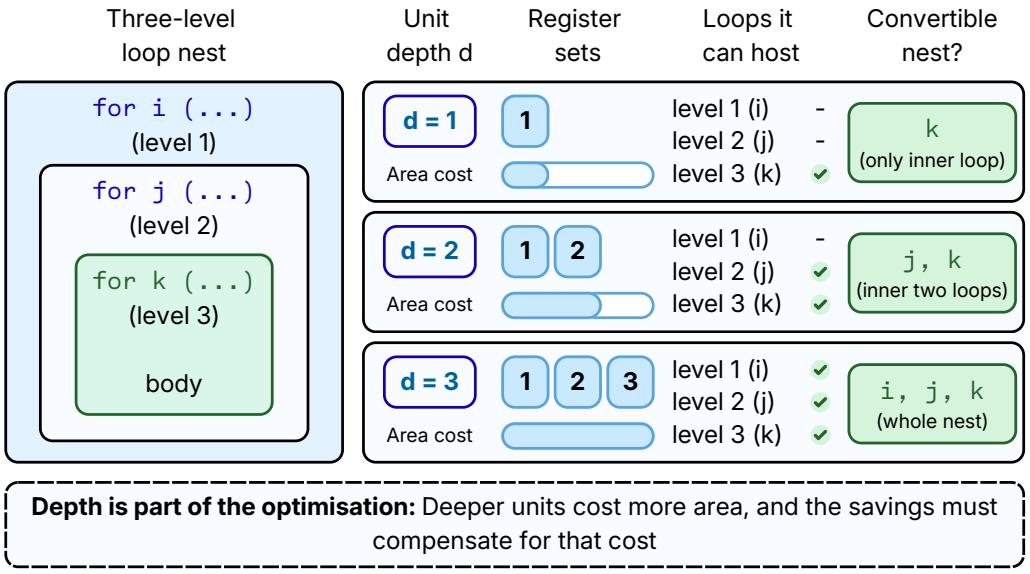


Figure 3.7: An illustrative three-level loop nest and the effect of the unit depth d. With $d = 1$ only the innermost loop `k` is converted, since it runs most often and a single hardware loop needs one register set; each further level of depth converts the next loop outward (`j`, then `i`) at the cost of one more register set, so the whole nest needs $d = 3$. The values are illustrative.

3.4.2 Hardware-Loop Configuration Space

The unit is itself configurable. Its configuration θ_{HL} gathers the loop depth d, the loop-counter width w_{cnt}, the loop-address width w_{addr}, and any further parameters a particular unit exposes. The admissible configurations form a space Θ_{HL} from which one is chosen for the synthesised core.

The depth d is the maximum number of hardware loops that can be active at the same instant, so it is governed by how deeply the converted loops nest. Loop A is the parent of loop B when A's body strictly contains B's and no other loop lies between them, and the nesting depth of a loop is one more than its parent's, with outermost loops at depth one. A loop runs while all of its ancestors are still in progress, so tracking a nest of depth k in hardware needs a unit of depth k. A workload built around deep loop nests, such as a multi-dimensional kernel, benefits from a higher depth that lets more of each nest be converted, while a workload whose hot loops are flat gains nothing beyond depth one, since no two of its converted loops are ever active one inside the other. A deeper unit admits more nested conversions but costs area on every core that carries it, since each level adds a register set and the logic that maintains it. Figure 3.7 illustrates this trade-off on a three-level nest.

Depth and the choice of loops cannot be fixed one before the other, because each depends on the other. A given set of loops needs a unit at least as deep as its deepest nest, while the depth in turn decides which loops can be converted at all, since a shallow unit rules out the deeper nests. A deeper unit converts more loops and removes more control overhead, at the cost of area, and the depth that gives the best area-and-performance result is found by weighing the two.

The two kinds of parameter in θ_{HL} are decided differently. The widths w_{cnt} and w_{addr} have a

Table 3.1: Illustrative application of the profitability estimate $\widehat{G}_\ell = E_\ell(N_\ell r_\ell - \sigma_\ell)$, with a fixed setup cost $\sigma_\ell = 3$ and one control instruction removed per iteration ($r_\ell = 1$). A loop must run more than σ_ℓ / r_ℓ iterations per entry to repay its setup, so the very short loop in the last row is rejected even though it is entered often. The saving shown is an estimate only, since code alignment and branch penalties give the converted loop further effects that the instruction count cannot capture; the decisive value comes from empirical execution.

Loop	N_ℓ	E_ℓ	$\widehat{G}_\ell$	Verdict
Long, entered once	100	1	+97	Convert
Short, hot re-entry	4	20	+20	Convert
Too short	2	20	−20	Keep software

single functionally-sufficient value, the smallest that holds the iteration counts and addresses the converted loops use, and below it the unit would be incorrect. Choosing them is the sizing problem of pruning in Section 3.2, applied to the loop unit. The depth d is different, since various values of it are all functionally valid once they cover the deepest nest, and they differ in the performance they yield and the area they cost. The pipeline thus selects the depth per workload, weighing the clock cycles a deeper unit saves against the area each added level costs. Depth zero is part of that choice, removing the unit entirely for a workload whose loops do not save enough to justify even one level, while a deeper value is taken only where the converted nests earn it. The hardware-loop strategy thus sizes the widths as pruning does, selects the depth on this performance-against-area basis, and adds to both the loop-selection decision and the RTL realisation that are specific to it.

Specialising a fixed unit through its parameters makes the strategy different in kind from custom instructions. A fused instruction is hardware built for one workload, its logic derived from the operation that workload repeats, so a different workload yields different hardware. The hardware-loop unit is the same logic on every workload, since redirecting a program counter around a loop body is a fixed function that does not depend on what the loop computes. What the strategy specialises is not the logic but its configuration, the depth and widths above, together with the software-side choice of which loops to convert.

3.4.3 Loop-Depth Profitability Model

Converting a loop is not always worthwhile, since the setup sequence is paid once per entry against a control saving paid once per iteration. For a loop ℓ, let σ_ℓ be the number of setup instructions inserted before it, r_ℓ the number of software loop-control instructions removed per iteration, N_ℓ the iteration count, and E_ℓ the number of dynamic entries. The instruction-count saving is

$$\widehat{G}_\ell = E_\ell \, (N_\ell \, r_\ell - \sigma_\ell). \tag{3.9}$$

This captures the main trade-off, since the setup is charged on each entry while the removed control instructions are saved on each iteration, so a loop iterated too few times between entries does not recover its setup. Table 3.1 illustrates the estimate on three small loops.

The estimate $\widehat{G}_\ell$ is purely an instruction count. Emitting the converted code can also change instruction layout, scheduling, and fetch behaviour in ways that depend on the target core, which the count cannot see. The methodology uses the static estimate to reject clearly unprofitable candidates, and obtains the value that actually decides a configuration from empirical execution of the transformed program.

3.4.4 Loop-Selection and Depth-Specialisation Problem

The strategy makes two choices jointly, which loops to convert and the hardware configuration that supports them, because each constrains the other. A selection is a subset $\mathscr{L} \subseteq \mathscr{L}_W$ of the loop candidates, a configuration is a point θ_{HL} in the space Θ_{HL}, and the merit function S scores each selection-and-configuration pair so that a lower score is better. The strategy seeks the pair that minimises this merit while keeping every selected loop implementable,

$$(\mathscr{L}^\star, \theta^\star_{\mathrm{HL}}) = \arg \min_{\substack{\mathscr{L} \subseteq \mathscr{L}_W \\ \theta_{\mathrm{HL}} \in \Theta_{\mathrm{HL}}}} S(\mathscr{L}, \theta_{\mathrm{HL}}), \qquad A_{\mathrm{tgt}}(\ell, \theta_{\mathrm{HL}}) = 1 \ \ \forall \ell \in \mathscr{L}. \tag{3.10}$$

The arg min searches over every selection $\mathscr{L}$ and every configuration θ_{HL}, and $(\mathscr{L}^\star, \theta^\star_{\mathrm{HL}})$ is the best pair it finds, while the side condition $A_{\mathrm{tgt}}(\ell, \theta_{\mathrm{HL}}) = 1$ for every $\ell \in \mathscr{L}$ admits only loops the chosen configuration can implement. This side condition couples the two choices, since a deeper or wider configuration enlarges the set of admissible selections at the cost of more hardware, which is what makes the configuration part of the problem rather than a value fixed afterwards.

The merit function S defines what the strategy optimises. A performance-only objective scores a configuration by its transformed clock cycle count against the baseline, while a hardware-aware objective combines that with an area term so the two are weighed against each other. Either objective, or another that weighs further metrics, can serve as S. The only requirement is that it scores all candidate configurations on a common scale, with lower being better, so the minimisation can select the best.

For each selected loop, the unit must be configured before the body executes, which requires placing a short setup sequence before the loop to write its boundaries, iteration count, and any other information required by the unit. This sequence may be produced either directly by the compiler or by a post-compilation transformation.

Emitting the setup is only the software half of the realisation. The core's RTL must also be able to act on it, which is two distinct additions. The first is decoding, so the setup instructions are recognised and their fields routed into the unit's registers. The second, and the one that does the work, is the loop-control logic itself, the registers that hold each active loop's bounds and remaining count, the comparison that detects the end of the body, and the redirect that sends the program counter back to the start while iterations remain.

The three RTL transformations have now been presented. Their instantiation on a concrete core, the catalogue and triggers driving each, the compiler integration, the RTL sites they touch, and the variants they produce are the subject of Chapter 4.

Chapter 4

Application of the Methodology to the CV32E40P Core

This chapter instantiates the methodology of Chapter 3 as Automated RISC-V Workload-Specific Specialisation (ARVIS), an end-to-end flow targeting CV32E40P. Given a workload, ARVIS compiles and profiles the baseline binary, applies the selected optimisation strategies to the core's RTL sources, and orchestrates the downstream compilation, simulation, and synthesis needed to verify and characterise each emitted variant. The chapter is organised by optimisation strategy, and for each one the abstract mechanism defined in the methodology chapter is mapped to the concrete steps and corresponding CV32E40P RTL code edits that realise it.

4.1 The CV32E40P Target Core

Applying and evaluating the methodology developed in Chapter 3 requires a concrete target core on which the proposed techniques can be implemented and assessed.

Within the RISC-V ecosystem, CV32E40P is used as a representative embedded-class core that exposes both standard and PULP-derived [28] extension surfaces. The core is highly parameterised, with synthesis-time configuration that turns off entire subsystems and resizes others, which aligns directly with the form of specialisation the previous chapter develops. It is an open core with documented RTL written in SystemVerilog, verified against the RISC-V specification and released with a detailed user manual [29], which makes it a reproducible target for an automated specialisation flow.

Other open-source RISC-V cores were considered. Ibex [33] is comparable in size, with a pipeline configurable as either two or three stages, but it does not provide the hardware-loop unit this work builds on. CVA6 [34] targets a larger application-class design point, with a six-stage pipeline, caches, deeper privilege support, and a branch predictor. CV32E40P sits between the two, small enough for source-level transformations to remain traceable, yet still exposing optional units, custom-extension support, and the hardware-loop infrastructure the strategies depend on.

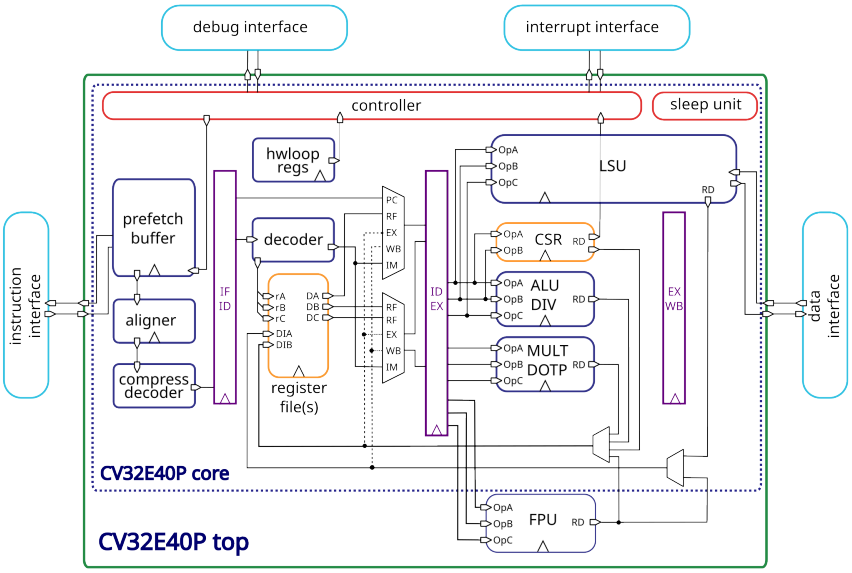


Figure 4.1: Top-level block diagram of CV32E40P, showing the core and the optional floating-point unit. Taken from CV32E40P user manual [29].

4.1.1 Pipeline and Extension Surface

CV32E40P is a four-stage in-order RISC-V core, with instruction fetch, instruction decode, execute, and writeback stages [29]. Its base configuration implements RV32IMC, combining the RV32I integer base with the M extension for integer multiplication and division and the C extension for compressed 16-bit instructions. It also supports Zicntr, Zicsr, and Zifencei, uses Open Bus Interface (OBI)-compliant instruction-fetch and load-store data buses, and operates in M-mode only. Figure 4.1 shows the top-level organisation of the core. The datapath contains the expected processing elements of an in-order embedded processor, including the instruction decoder, register file, ALU, multiplier, Load-Store Unit (LSU), CSR register file, and central controller. Instruction delivery is handled by a prefetch buffer, while the aligner and compressed decoder support the variable-length instruction encodings introduced by the C extension. Externally, the core exposes dedicated instruction-memory, data-memory, debug, and interrupt interfaces.

Several optional extensions are configurable at synthesis time. The standard F extension adds a separate 32-entry single-precision floating-point register file together with the floating-point arithmetic, conversion, and comparison instructions defined by the RISC-V specification. The Zfinx variant of the F extension reuses the integer register file for floating-point operands instead of providing a separate file, trading the performance benefit of the dedicated register file for lower area and a simpler register-allocation contract with the compiler. The non-standard Xcv extension brings the CORE-V PULP ISA extensions, which include the hardware loop unit and custom instructions for post-increment loads and stores, multiply-accumulate, bit manipulation, and SIMD.

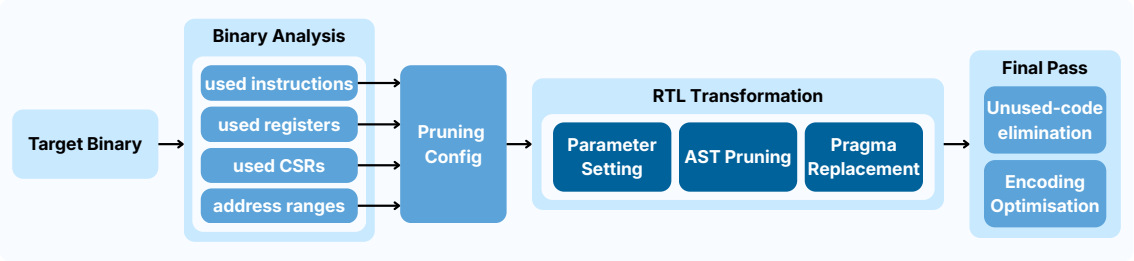


Figure 4.2: The hardware pruning pipeline on CV32E40P. The target binary is analysed to extract the workload's hardware requirements, which determine the pruning configuration. Three transformation mechanisms apply the configuration to the RTL sources, followed by unused-code elimination and encoding optimisation.

The architectural elements summarised are the surfaces on which the strategies of Chapter 3 attach. Pruning removes unused logic from across the core, fusion adds new instructions at the decoder and the arithmetic datapath, and hardware-loop modifies the existing loop unit and adds its own setup instructions.

4.2 Hardware Pruning

The methodology chapter defined pruning as a trigger-driven transformation from a workload profile to a specialised RTL configuration. On CV32E40P this becomes four concrete parts. The first is a catalogue of removable features and resizable parameters specific to the core. The second is a binary-analysis pass that extracts the quantities the triggers consume from the compiled workload and evaluates each trigger against them. The third is the set of three source-transformation mechanisms, parameter setting, Abstract Syntax Tree (AST) pruning, and pragma replacement, that apply the resulting configuration. The fourth is a final cleanup pass that resolves the artefacts these transformations leave behind. Figure 4.2 shows how the four parts compose into the pruning pipeline, and each is described in turn below.

4.2.1 Feature and Parameter Catalogue

For CV32E40P, the pruning catalogue associates each removable feature or resizable parameter with its enclosing module and with the trigger that determines the corresponding pruning decision. For features, whether the hardware is retained and for parameters, the minimum value required by the workload. Table 4.1 shows the catalogue prepared for CV32E40P.

Entries are organised by parent module to expose the hierarchical structure of the design. When a parent feature is disabled, the entire module is removed together with all contained subcomponents. For example, removing the multiplier eliminates the MULH state machine, the dot-product datapath, and the MSU subtract path. When a parent feature is retained, its internal components may still be removed. A multiplier preserved due to `mul` support can still drop the MULH and MULHSU paths if those instructions are not required, and a maintained ALU can similarly

eliminate unused operations such as division, population count, or find-first-one. The same applies to the decoder, register file, and CSR file, where individual entries or cases may be removed even when the enclosing module is preserved.

Some catalogue entries exist only because CV32E40P implements them as extra features, among them the compressed-instruction decompressor, the hardware-loop unit, the post-increment load and store unit, and the population-count and find-first-one units inherited from the PULP RI5CY heritage, whose triggers key off mnemonics from the CORE-V Xcv extension (the `cv.*` family) and the C extension (the `c.` prefix). The triggers themselves, recorded in the third column of the table, are the Boolean predicates the methodology of Section 3.2 defines for each feature. In the simplest case the predicate is a single membership test, retaining a unit when its instructions appear, and where a feature is reached through more than one capability the predicate is a Boolean combination of such tests. The interrupt controller is retained when any of `mie`, `mip`, `mtvec`, or `mscratch` is accessed or `mret` is used, and the debug module on `ebreak` or any debug CSR. The dependency case arises when one unit's logic is shared with another, so the population-count and find-first-one units are retained by a binary that uses division, since they share zero-detect and leading-zero counting with the divider, even when `p.cnt` and `p.ff1` are absent.

4.2.2 Binary Usage Analysis

The triggers of Table 4.1 are supported on four quantities derived from the binary, the sets of used instructions, used registers, and used CSRs, and the largest text-section end address. Gathering all four takes two separate `objdump` [35] invocations, detailed in turn below. A disassembly invocation exposes the instruction stream, from which the used-instruction, used-register, and used-CSR sets are built, and a section-header invocation exposes the memory layout, from which the address bound is derived.

The disassembly invocation is shown in Figure 4.3. It is run with pseudo-instruction rewriting disabled, so that every line carries the mnemonic actually encoded in the binary, since the triggers match against real mnemonics instead of aliases. As such, a register move appears as `addi rd, rs, 0` and not `mv rd, rs`. Each disassembled line is parsed into its mnemonic and operand list. The mnemonic is added to the used-instruction set, its register operands to the used-register set, and, when the instruction is a CSR access, the addressed CSR to the used-CSR set. A compressed mnemonic contributes both its own entry, which keeps the compressed decoder, and its uncompressed equivalent, which keeps the underlying functional unit. In the excerpt, the `wfi` line keeps the sleep unit, while the compressed `c.addi` and `c.j` lines keep the compressed decoder and aligner alongside the base operations they expand to.

The section-header invocation is shown in Figure 4.4. The largest text-section end address is the maximum of $VMA + Size$ taken over the rows flagged as executable code. Although this is an exclusive end address and not the address of the final instruction, using it to size the PC is conservative, since no instruction lies above it. For the binary shown, this is the end of `.text` at `0xc132`, giving a 16-bit PC. The bound is sound under a flat memory map without runtime relocation beyond the linked address space, which holds for the bare-metal scenarios targeted.

Table 4.1: Feature and parameter catalogue for CV32E40P. Each entry pairs a feature or parameter with the module that contains it and the trigger that determines whether the feature must be retained or what value the parameter takes for a given application binary.

Feature or parameter	Parent module	Trigger
Multiplier	core	`mul`, `mulh`, `mulhsu`, or `mulhu` in the binary
MULH state machine	multiplier	`mulh` or `mulhsu` in the binary
Dot-product datapath	multiplier	PULP dot-product instruction in the binary
MSU subtract path	multiplier	`p.msu` in the binary, or the variant's decoder generates `MUL_MSU32` (fused mul+sub patterns)
Divider	ALU	`div`, `divu`, `rem`, or `remu` in the binary
Floating-point unit	top	any F or D instruction in the binary
Debug module	controller	`ebreak` or debug CSR access in the binary
Interrupt controller	core	interrupt CSR access or `mret` in the binary
Hardware loop unit	core	any hardware-loop instruction in the binary
Compressed decoder and aligner	fetch and ID stages	any `c.*` instruction in the binary
Sleep unit	core	`wfi` in the binary
Population-count unit	ALU	`p.cnt` in the binary, or any of `div`, `divu`, `rem`, `remu`
Find-first-one unit	ALU	`p.ff1` in the binary, or any of `div`, `divu`, `rem`, `remu`
Register file second write port	register file	any PULP post-increment load or store in the binary
Register file third read port	register file	any FPU or R4-type custom instruction in the binary
Individual ALU operation	ALU	the corresponding instruction present in the binary
Individual decoder case	decoder	the corresponding mnemonic present in the binary
Individual register entry	register file	the register index referenced in the binary
Individual CSR address	CSR file	a CSR access to that address in the binary
PC width	core	minimum number of bits required to represent the largest text-section end address in the binary
Performance-counter array size	CSR file	one if any `mhpm` CSR is read, zero otherwise

```
$ riscv32-unknown-elf-objdump -d -M no-aliases --no-show-raw-insn bin.elf

00000180 <_start>:
     180:       auipc   gp,0xc
     184:       addi    gp,gp,376 # c2f8 <sk>
     188:       auipc   sp,0x1d
     18c:       addi    sp,sp,-456 # 1cfc0 <_end>
     ...
     1a8:       c.addi  t0,4
     1aa:       c.j     1a0 <_start+0x20>
     1ac:       jal     ra,be86 <main>
     1ba:       wfi
     1be:       c.j     1ba <_start+0x3a>
```

Figure 4.3: Disassembly invocation for a compiled benchmark binary, with an abridged excerpt of its output covering the `_start` routine. Each line carries an address, a mnemonic, and a comma-separated operand list, with hash-prefixed comments showing resolved label addresses. The flag `-d` requests the disassembly, `-M no-aliases` disables pseudo-instruction rewriting, and `-no-show-raw-insn` omits the raw machine-code column.

```
$ riscv32-unknown-elf-objdump -h bin.elf

bin.elf:     file format elf32-littleriscv

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00000040  00000180  00000180  00000180  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text         0000bf72  000001c0  000001c0  000001c0  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata       000001c0  0000c138  0000c138  0000c138  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  ...
```

Figure 4.4: Section-header invocation for the same binary. Each row gives a section's size and its load and virtual addresses. The rows flagged `CODE` (`.init` and `.text`) are the executable sections; rows flagged `DATA` or carrying debug information are not.

The four sets and the largest end address are the concrete form of the workload profile of Section 3.2.2, the observables $(I, R, S, a_{\max})$ extracted from the workload W, with I, R, and S the used-instruction, used-register, and used-CSR sets and $a_{\max}$ the largest end address. Evaluating every trigger in Table 4.1 against W instantiates the feature decisions Ψ and parameter values Φ, yielding the pruning configuration $C = (\Psi, \Phi)$ that the RTL transformations consume.

4.2.3 RTL Pruning Transformations

The pruning configuration C produced by the binary analysis records, for each catalogue entry, whether its logic is kept or removed and what value each parameter takes. Applying it to the core's sources is the transformation $\mathrm{RTL}' = T(\mathrm{RTL}, C)$ of Section 3.2.3, realised by three mechanisms whose composition is T. The three exist because the logic a feature occupies is embedded differently from one feature to the next. Parameter setting handles module instantiations and parameterised dimensions, switched off or resized at their boundary. AST-based case pruning handles individual case arms that must be removed while the surrounding statement and its other arms stay intact. Pragma replacement handles logic woven through multiple procedural blocks and shared between features, where no clean boundary exists to switch off or to excise.

Two of the three mechanisms rely on a one-time manual preparation of the core sources, carried out once for CV32E40P and reused for every workload. The parameters switched by parameter setting are added by hand where the upstream sources do not already provide them, and the begin and end markers that pragma replacement acts on are likewise inserted around the relevant regions. The AST-based pruning needs no such preparation, since it locates the case arms to remove from the structure of the source itself. With the core so prepared, the per-workload flow is automatic, setting the parameters and evaluating the pragma regions from the configuration C. The following subsections show the RTL site each mechanism acts on.

4.2.3.1 Parameter Setting

Parameter setting is used both to switch off whole removable blocks and to resize parameterised dimensions to the workload. Such a block is wrapped in a synthesis-time parameter that the configuration sets, so that disabling the parameter removes the block and ties off its outputs, while the surrounding logic is untouched. The divider is a representative case. Figure 4.5 shows its instantiation wrapped in a `generate` block guarded by the parameter. The preparation stage introduces the `ENABLE_DIV` parameter with default value `1`, preserving the behaviour of the original core. When disabled, the divider module is removed and its outputs are tied off, leaving the surrounding logic unchanged.

Some parameters already exist in the upstream CV32E40P sources. A subset of these, including `FPU_ADDMUL_LAT`, `FPU_OTHERS_LAT`, `ZFINX`, and `COREV_CLUSTER`, are unused in this work because the corresponding functionality is disabled by default. Others are reused for pruning, including `FPU` for the Floating-Point Unit (FPU), `COREV_PULP` for the PULP custom extensions, and `NUM_MHPMCOUNTERS` for the number of hardware performance counters. The

```
generate
  if (ENABLE_DIV) begin : gen_div
    cv32e40p_alu_div alu_div_i (
      .Clk_CI    (clk),
      // ... other ports
      .Res_DO    (result_div),
      .OutVld_SO (div_ready)
    );
  end else begin : no_div
    assign result_div = '0;
    assign div_ready  = 1'b1;
  end
endgenerate
```

Figure 4.5: Divider instantiation after parameter setting.

remaining parameters are introduced during core preparation. The `PC_WIDTH` parameter is propagated through the fetch and CSR stages so that PC-related signals are declared using the reduced width rather than the full 32 bits, with boundary casts inserted wherever resized PC values interface with fixed-width signals. Table 4.2 summarises the parameters used for parameter setting and identifies whether each originates from the upstream core or from the preparation stage.

4.2.3.2 AST-Based Case Pruning

AST-based case pruning is used where the logic to remove sits inside a construct that must itself remain, so there is no clean boundary at which a parameter could switch it off. Removing such an entry means editing inside the construct while leaving the rest of it operating. The dispatch `case` statement is the representative example, illustrated in Figure 4.6, where individual arms must be dropped while the statement and its other arms stay intact. Editing the source as text would be fragile, since the arms span multiple lines and share comma-separated label lists, so the edit is made on the parsed syntax tree instead. The mechanism applies at three such sites in CV32E40P, the operation dispatch in the ALU, the CSR address dispatch in the CSR register file, and the mode dispatch in the multiplier. The library used is pyslang [36], the Python bindings for the slang SystemVerilog frontend.

Three independent passes apply the mechanism across the core. Pass A removes unused `ALU_*` operations from the ALU dispatch logic. Pass B removes unused `CSR_*` addresses from the read and write dispatches, including the M-mode performance-counter and physical-memory-protection sets. Pass C removes the `MUL_DOT8` and `MUL_DOT16` multiplier modes when the dot-product unit itself is removed.

These passes edit the dispatch logic in the consumer files, removing the case arms that select the pruned operations. The corresponding member definitions in the shared package, and the references to them across the decoder, the ALU, and the multiplier, are reconciled later by the encoding-optimisation cleanup pass of Section 4.2.4, which removes the now-unused members and renumbers the rest. Performing that reconciliation in one place keeps the three views of

Table 4.2: Parameters used for parameter setting. Parameters marked with a checkmark were already present in the upstream CV32E40P sources, while the rest were introduced during core preparation.

Parameter	Already in core	Effect
FPU	✓	Removes the floating-point unit
FPU_ADDMUL_LAT	✓	FPU adder and multiplier pipeline latency, not manipulated by pruning
FPU_OTHERS_LAT	✓	FPU comparison and conversion pipeline latency, not manipulated by pruning
ZFINX	✓	Float-in-integer-register-file mode, not manipulated by pruning
COREV_PULP	✓	Removes PULP extensions
COREV_CLUSTER	✓	PULP Cluster `cv.elw` support, not manipulated by pruning
NUM_MHPMCOUNTERS	✓	Sets the number of HPM counters
DEBUG_TRIGGER_EN		Enables debug trigger registers
ENABLE_DIV		Removes the divider
ENABLE_MUL		Wraps the multiplier instantiation, retained when any multiplier path is needed and otherwise removed in conjunction with the per-mode parameters below
ENABLE_MULH		Removes the MULH state machine
ENABLE_DOT_MUL		Removes the dot-product datapath
ENABLE_MSU		Removes the MSU subtract path
ENABLE_POPCNT		Removes the population-count unit
ENABLE_FF1		Removes the find-first-one unit
ENABLE_REGFILE_WR_B		Removes the register file second write port
ENABLE_REGFILE_RD_C		Removes the register file third read port
USED_REGS_MASK		Per-register write-enable mask
PC_WIDTH		Width of the main pipeline PC

```
always_comb begin
  case (operator_i)
    ALU_AND, ALU_OR, ALU_XOR:
      result_o = bit_op_result;
    ALU_ADD, ALU_SUB:
      result_o = adder_result;
    ALU_BEXT, ALU_BEXTU:
      result_o = bextins_result;
    ALU_SHUF, ALU_SHUF_H:
      result_o = shuffle_result;
    // ... further items
    default:
      result_o = '0;
  endcase
end
```

Figure 4.6: Excerpt of the ALU operator dispatch.

each enumeration consistent, so the decoder cannot be left dispatching to a multiplier mode the multiplier no longer accepts.

4.2.3.3 Pragma-Block Replacement

Pragma replacement handles features whose logic is cross-cutting, spread across multiple procedural blocks and interwoven with other features so that no parameter boundary encloses it. Such logic could in principle be reached by the same syntax-tree rewriting used for case pruning, since a tree edit can target any construct. What makes that impractical is that the required edit is not one uniform structural operation but a different, site-specific rewrite at each location. Removing a feature here deletes a block, there drops a single term from a Boolean guard, and elsewhere replaces a plain assignment with one conditioned on whether a related feature survives. Capturing each as a structural rule would mean a bespoke tree transformation per site, each kept valid as the surrounding code changes. Pragma replacement avoids this by separating where an edit applies from what it does. A marked region pins the location, stable across unrelated edits to the file, and a generator supplies the replacement for that region, so a new site is handled by marking it and writing its generator. The cross-cutting features of CV32E40P addressed this way are listed in Table 4.3, which maps each pragma prefix to its feature domain.

Table 4.3: Pragma prefixes and the cross-cutting features they cover.

Prefix	Feature
`DBG`	Debug module, `ebreak` handling, debug CSRs
`IRQ`	Interrupt controller, interrupt-related CSRs, `mret` handling
`HWLP`	Hardware loop unit, loop registers, fetch redirect logic
`CTRL`	Controller decode block, including the four-way IRQ-times-debug combinations
`PULP`	PULP-specific decoder, immediate multiplexers, and post-increment paths

```
// ARVIS_DBG_BEGIN: REMOVE
logic [31:0] dcsr_q;
logic [31:0] dpc_q;
// ARVIS_DBG_END: REMOVE

// ARVIS_CTRL_BEGIN: gen_ctrl_decode_debug_irq
if (debug_req_i && irq_pending_i) begin
    ctrl_fsm_ns = DBG_TAKEN_IRQ_PENDING;
end else if (debug_req_i) begin
    ctrl_fsm_ns = DBG_TAKEN;
end else if (irq_pending_i) begin
    ctrl_fsm_ns = IRQ_TAKEN;
end else begin
    ctrl_fsm_ns = DECODE;
end
// ARVIS_CTRL_END: gen_ctrl_decode_debug_irq
```

Figure 4.7: Pragma actions applied to representative RTL fragments.

Markers of the form `// ARVIS_<PREFIX>_BEGIN: <action>` and the matching `END` markers are placed during core preparation around the relevant logic. The `<action>` is one of two kinds. A `REMOVE` action deletes the bracketed region outright when the feature's trigger is inactive, which suits logic that is simply dropped when the feature is absent. A named-handler action instead delegates the region to a generator function identified by the name, which emits replacement RTL computed from the feature configuration, and is used where the region must be rewritten according to which related features survive. Figure 4.7 shows one fragment of each kind, a `DBG` region marked `REMOVE` and a `CTRL` region delegated to a named handler.

The handler `gen_ctrl_decode_debug_irq` generates the controller decode logic depending on whether the interrupt and debug triggers are active. Algorithm 1 summarises its behaviour. When neither trigger is active, it emits nothing. When both are active, it produces a four-way decode covering debug, interrupt, combined debug-and-interrupt, and default paths. When only one trigger is active, it produces the corresponding reduced decode.

Algorithm 1 The `gen_ctrl_decode_debug_irq` handler.

```
1:  function GEN_CTRL_DECODE_DEBUG_IRQ(irq, dbg)
2:      if not irq and not dbg then
3:          return empty string
4:      else if irq and dbg then
5:          return four-way decode logic
6:      else if dbg then
7:          return two-way decode (debug)
8:      else
9:          return two-way decode (IRQ)
10:     end if
11: end function
```

Algorithm 2 Iterative unused-code elimination on one source file.

function ELIMINATEUNUSEDCODE(*file*)
  repeat
    declared ← locally declared signals of *file*
    unused ← members of *declared* used nowhere else
    for all $s \in$ *unused* **do**
      remove the declaration of s
      remove the assignment that produces s
    end for
  until *unused* $= \varnothing$
end function

4.2.4 Post-Pruning Cleanup

After the primary transformations, ARVIS runs the cleanup passes of Section 3.2.4 on the modified CV32E40P sources, realised as two passes. The first, unused-code elimination, removes signals orphaned by pruning, and the second, encoding optimisation, repacks internal enumerations whose member sets have shrunk. Each is given below as the procedure it runs, together with a representative example from the CV32E40P sources.

The unused-code elimination pass converges in as many iterations as the longest chain of dependent signals left behind by the upstream transformations. Algorithm 2 states the procedure. Each iteration collects the locally declared signals of a file and tests each for use by scanning the rest of the file for a word-boundary occurrence of its name. A signal that occurs nowhere outside its own declaration is unused, and both its declaration and the assignment that produced it are removed. Removing an assignment can orphan the signals it read, so the pass repeats until an iteration removes nothing. Figure 4.8 illustrates a three-step cascade observed in the CV32E40P multiplier sources, where each signal feeds the next.

The encoding-optimisation pass applies to the enumeration types defined in the CV32E40P package, among them the ALU operator enumeration (seven bits wide, encoding the full ALU operation set), the multiplier mode enumeration, and smaller internal enumerations such as the

```
logic [31:0] mulh_res;
logic [31:0] mulh_ext;
logic [31:0] mulh_comb;

assign mulh_ext =
    mulh_res << 1;
assign mulh_comb =
    mulh_ext | 32'h1;
```

(a) Initial code.

```
logic [31:0] mulh_res;
logic [31:0] mulh_ext;
//logic [31:0] mulh_comb;

assign mulh_ext =
    mulh_res << 1;
//assign mulh_comb =
    mulh_ext | 32'h1;
```

(b) After iteration 1.

```
logic [31:0] mulh_res;
//logic [31:0] mulh_ext;
//logic [31:0] mulh_comb;

//assign mulh_ext =
    mulh_res << 1;
//assign mulh_comb =
    mulh_ext | 32'h1;
```

(c) After iteration 2.

Figure 4.8: Cascading unused-code elimination. Removing `mulh_comb` in the first iteration leaves `mulh_ext` unused, and removing `mulh_ext` in the second leaves `mulh_res` unused, so a third iteration removes the last signal.

Algorithm 3 Encoding optimisation over the package enumerations.

for all enumeration E in the package **do**
  if E carries externally fixed values **then**
    skip E
  else
    $kept \leftarrow$ members of E that survive pruning
    if E has bit-extraction constraints **then**
      assign encodings to $kept$ preserving the constrained low bits
    else
      assign $kept$ sequential encodings of width $\lceil \log_2 |kept| \rceil$
    end if
    remove the unused members and update the width parameter of E
  end if
end for

prefetch and multiplier-pipeline state enumerations. Some of these are subject to a bit-extraction constraint, where low-bit relationships must be preserved through re-packing, the canonical case being the ALU operator enumeration, whose low bits the divider reads individually.

Algorithm 3 illustrates the process. A member survives when its name occurs anywhere in the sources outside the package definition, found by the same word-boundary scan the unused-code pass uses, and the bit-extraction constraint is detected by searching for slice expressions of the form `signal[bits]` on a signal of the enumeration's type. For each enumeration the pass then determines which members survive pruning and re-encodes them into the smallest sufficient width, updating the type's width parameter accordingly. Two cases are exempt. An enumeration whose value carries an external meaning, such as a state machine the controller depends on or a field fixed by the RISC-V specification, is left untouched. An enumeration with this bit-extraction constraint, like the ALU operator above, is re-encoded with those bit relationships held fixed and the remaining members renumbered around them. Because members are referenced by name throughout the sources, rewriting the package is sufficient and the consumer files need no edits.

4.3 Custom Instructions

The custom-instruction methodology leaves various choices to the target platform, the compiler integration, the encoding format and its capacity, the scheme that delivers immediates, and the core-side decode and arithmetic the fused instructions reuse. The methodology asks only that the chosen compiler can be extended to recognise the discovered patterns and emit the corresponding custom instructions, and that the core offer a custom-encoding space and arithmetic hardware to reuse. This section fixes them for CV32E40P and GCC, extending the compiler to emit the fused instructions and the core to decode and execute them on its existing datapaths.

Figure 4.9 lays out the full implementation flow, in three stages that the rest of the section develops in turn. The first stage, discovery, turns the workload into a set of fused-instruction candidates from two sources. Register-register patterns are collected from the binaries that prebuilt GCC

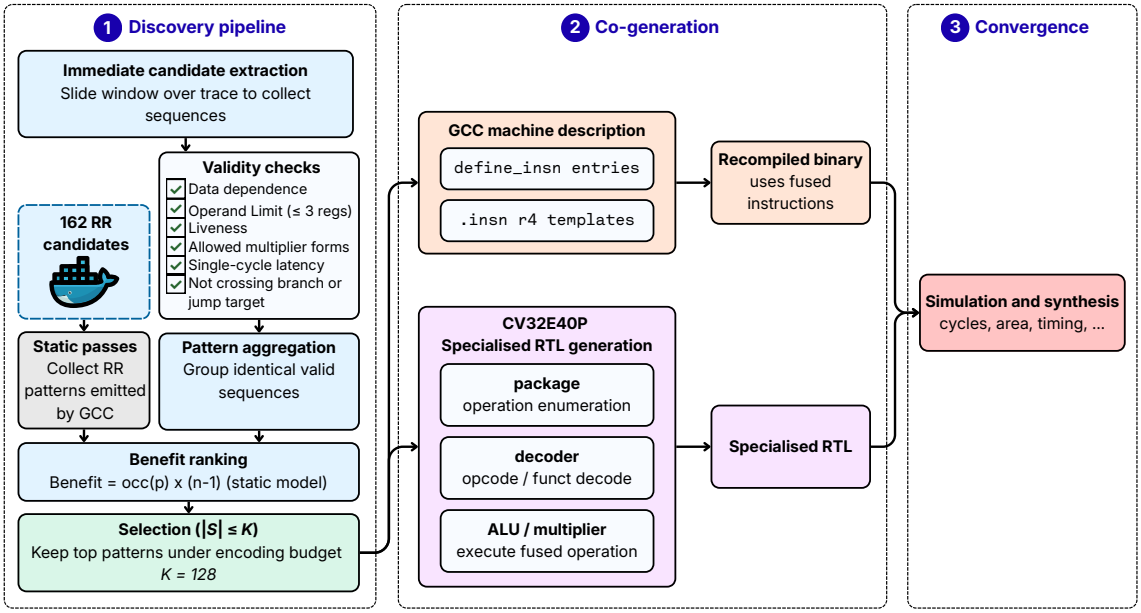


Figure 4.9: Custom-instruction fusion discovery, selection, and co-generation on CV32E40P. Candidate patterns are obtained from register-register discovery passes and from immediate-pattern extraction over the workload trace. Valid candidates are aggregated, ranked by estimated static benefit, and selected under the encoding budget. The selected set then drives both GCC machine-description generation and CV32E40P RTL generation, producing a fused binary and matching execution support for simulation and synthesis.

passes emit and are valid by construction. Immediate patterns are extracted by sliding a window over an execution trace of the workload and must pass the validity checks the methodology requires before they are admitted. The surviving immediate occurrences are then aggregated into patterns and, together with the register-register patterns, ranked by the static benefit $\mathrm{occ}(p)\,(n_p - 1)$, with the highest-ranked kept within the core's encoding budget. The second stage, co-generation, takes the selected set and produces two matching artefacts, the GCC machine-description entries that let the compiler emit each fused instruction and the CV32E40P RTL that decodes and executes it, so that recompiling the workload yields a fused binary that the specialised core runs. The third stage feeds both into simulation and synthesis to obtain the clock cycle count, area, and timing of the result.

The software side of this flow is realised first, since the generated compiler patterns determine which fused instructions can appear in the recompiled binary.

4.3.1 GCC Machine-Description Integration

This work encodes every fused instruction in the R4-type format of Figure 2.2, the only base RISC-V format with three register-source fields. The choice is made because a fused pair may name up to three distinct register operands, which no two-source format can carry. With the format fixed, the capacity K left abstract in Section 3.3.1 takes a concrete value. CV32E40P, like any RISC-V core, exposes the four `CUSTOM` opcodes the specification reserves, and within each

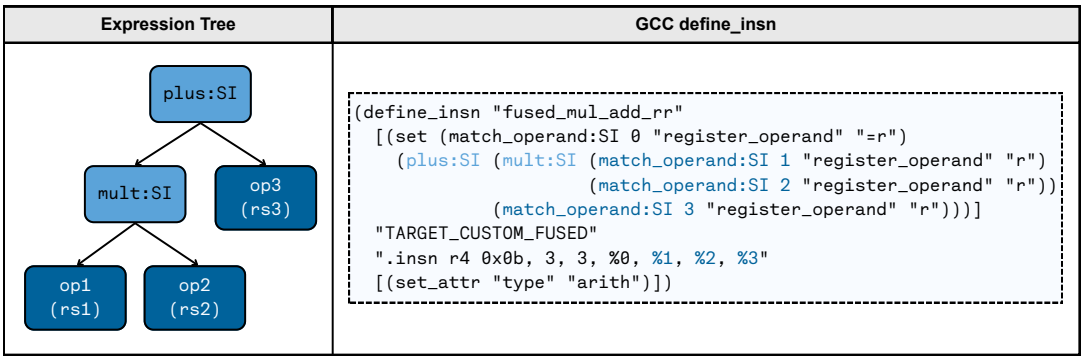


Figure 4.10: The multiply-accumulate fusion $rd = (rs_1 \times rs_2) + rs_3$ shown as a GCC expression tree (left) and as the `define_insn` entry that encodes it (right). The internal nodes of the tree are operators and the leaves are the `match_operand` operands, with the `plus` root and the `mult` child composing into the single fused operation. The assembly template emits the instruction through the RISC-V `.insn` directive in the R4-type form, naming the `CUSTOM` opcode `0x0b` together with the `funct3` and `funct2` values that select the operation.

the two-bit `funct2` field of the R4-type format distinguishes four operations per `funct3` class, so each opcode holds thirty-two R4-type encodings (eight `funct3` values multiplied by four `funct2` values) and the four opcodes together provide $K = 128$ distinct fused-operation encodings.

The GCC compiler realises each fused operation through a `define_insn` entry in the target's machine description, the file that lists the patterns the compiler is allowed to produce [37], [38]. Each entry pairs an expression over operands with the assembly to emit when that expression is matched, and the encoding chosen for the instruction is written into that assembly template. The 128 encodings bound the number of distinct fused operations the final binary can carry. The number of `define_insn` entries may be larger, since an immediate pattern can require one entry per selected constant while sharing a single operation encoding.

Internally, GCC represents each instruction as an expression tree over its operands. The leaves of the tree are operands matched by `match_operand`, and the internal nodes are operators such as `plus`, `minus`, `mult`, `ashift` for shift-left, `lshiftrt` for logical shift-right, `ashiftrt` for arithmetic shift-right, `and`, `ior` for bitwise or, and `xor`. The suffix `:SI` marks a single-integer mode, the thirty-two-bit word. A fused pattern is the composition of the two operator trees, where the output of the first becomes an operand of the second. A `define_insn` pairs this composed tree with the assembly the compiler emits when the tree matches. Figure 4.10 shows this correspondence for the multiply-accumulate fusion, placing the expression tree beside the `define_insn` that encodes it.

A `define_insn` numbers its operands from zero and refers to them in the assembly template as `%0`, `%1`, and so on. By convention operand 0 is the output, while the remaining operands are the source registers. The constraint `"=r"` on operand 0 marks it as a register that the instruction writes, and `"r"` on the others marks them as registers the instruction reads. The `TARGET_CUSTOM_FUSED` condition guards the pattern, so the compiler considers it only when the build sets the respective command-line flag, leaving normal compilation unaffected. The assembly template emits the in-

struction through the RISC-V `.insn` directive in the R4-type form, naming the `CUSTOM` opcode and the `funct3` and `funct2` values that select this operation, followed by the destination and the three source registers. The trailing `set_attr "type" "arith"` classifies the instruction as arithmetic, the same class the base ALU operations carry.

Two kinds of normalisation, both documented in the GCC machine descriptions reference [38], can prevent a literal composition from matching. The first is canonicalisation, which reorders an expression without changing the operation. The most relevant example for fusion is operand ordering, since for a commutative operator the constant operand is always placed second, so a pattern that intends to match a logical operation against an immediate must write the constant on the right of the operator. An `and` of a register with the constant 5, for instance, is canonicalised to `(and (reg) (const_int 5))` regardless of how it was written in the source, so a pattern that places the constant first as `(and (const_int 5) (reg))` never matches. The second is algebraic simplification, which replaces one operation with an equivalent operation. An exclusive-or against an all-ones constant simplifies to a bitwise negation, so a pattern that begins with such an operation must use the `not` node and not an `xor` with the constant -1. The generated machine description writes both kinds of cases in the shape the compiler expects.

4.3.2 Register-Register Patterns

Applying the methodology's single-cycle and datapath-reuse constraints to CV32E40P admits ten register-register ALU operations, `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt`, and `sltu`. The single-cycle multiplication `mul` is also admitted, although only in two specific compositions. The multiplier of CV32E40P exposes two pre-existing datapaths whose composed expression matches a fused multiplier pattern without adding a new multiplier datapath, with the added logic confined to decode and control. Those are the multiply-accumulate and the multiply-subtract paths.

The ten admitted ALU operations give $10 \times 10 = 100$ forward patterns. Six of the ten are non-commutative (`sub`, `sll`, `srl`, `sra`, `slt`, and `sltu`), so they contribute $10 \times 6 = 60$ reverse variants. The two admitted multiplier fusions add two further patterns. The complete set thus contains $100 + 60 + 2 = 162$ register-register patterns.

Since the 162 patterns exceed the 128 encodings available across the four `CUSTOM` opcodes, they are split across two independent compilation passes ($128 + 34 = 162$), each carrying a different subset in its machine description. Each pass is a GCC toolchain built once as a Docker image and reused across every benchmark ARVIS processes, since the pattern subset a pass carries is fixed and workload-independent. The workload is compiled once with each pass, and the resulting binary is disassembled with `objdump` to identify which fused instructions the compiler actually selected and how many times each occurs.

4.3.3 Immediate Patterns

Unlike the register-register catalogue, immediate patterns cannot be enumerated a priori, because the constants they involve are workload-specific and are not known until a compiled binary is

available. Their discovery operates instead on the disassembled binary itself. The decision to use a pattern ultimately rests with the compiler, but enumerating every candidate exhaustively would hand it far more patterns than are actually valid as single-instruction replacements. The discovery is therefore selective, filtering the candidates so that only the admissible and useful patterns are supplied to the compiler and the encoding space does not overflow.

The unit the discovery examines is a *window*, a run of two to four consecutive instructions within one basic block. The search weighs chains of up to four instructions, but only dependent pairs pass its admissibility conditions, each becoming one fused instruction. The discovery runs as a pipeline, first building the analyses the search depends on, then sliding the window across the program to enumerate candidates, validating each against the admissibility conditions, and aggregating the survivors into counted patterns ready for the compiler. The subsections below follow that order.

4.3.3.1 Precomputed Analyses

Four analyses are computed over the disassembled binary before any window is examined, since each is consulted repeatedly during the search and depends only on the binary, not on the window. Two of them, the basic-block structure and the set of branch-target addresses, support the program-wide validity checks the methodology requires. The instruction stream is partitioned into basic blocks linked to their successors, the standard control-flow structure over which the branch-safety and liveness checks are evaluated, and from it the branch-target addresses are collected, the targets of every branch and jump that the control-flow safety check consults. The third analysis is per-block execution counts, taken from an execution trace of the workload produced by the Spike RISC-V ISA simulator [39] running the baseline binary, whose per-instruction commit log is reduced to a count per block.

The fourth analysis is global liveness, used to determine which of a window's writes the fused instruction must produce. A fused instruction writes back only the registers that are read after the window and drops the rest, the values the chain produced only to feed a later step within itself, never writing those to the register file. A register the window writes is therefore an output the instruction must produce when it is read after the window, and an eliminable intermediate when it is not. The read that matters can be anywhere downstream, so the test is simply whether the value is used again. Answering it requires a whole-program analysis, not a within-block forward pass, which would miss a value read only after a back edge. The discovery runs a standard global liveness analysis over the basic-block structure, computing for each block a *live_in* set, the registers live on entry, and a *live_out* set, the registers live on exit. Algorithm 4 computes them using the backward data-flow fixpoint, in which a block's *live_in* is its used registers together with its *live_out* minus the registers it defines, and its *live_out* is the union of its successors' *live_in* sets, iterated until no set changes.

The block-level sets are not the final answer, however, because a window is only a sub-range of its block and the instructions between the window and the block's end can themselves read a register the window writes. The set of registers live at the window's exit is therefore obtained

Algorithm 4 Global liveness by backward data-flow fixpoint.

for all basic block b **do**
    $use[b] \leftarrow$ registers read before being written in b
    $def[b] \leftarrow$ registers written in b
    $live\_in[b] \leftarrow \varnothing, \quad live\_out[b] \leftarrow \varnothing$
end for
repeat
    for all basic block b **do**
        $live\_out[b] \leftarrow \bigcup_{s \in succ(b)} live\_in[s]$
        $live\_in[b] \leftarrow use[b] \cup (live\_out[b] \setminus def[b])$
    end for
until no *live_in* or *live_out* set changes

by starting from the block's *live_out* and walking backward over the instructions that follow the window inside the block, adding each register such an instruction reads and removing each one it writes. This per-window set, written $live\_after$, captures both a read later in the same block and a read in any successor block, so it is the set against which the window's defined registers are classified.

The $live\_after$ set is what lets the analysis decide which of a window's writes are needed. As an example, take a window whose chain writes `a2` as its final result and `a1` as an intermediate that feeds a later step inside the chain. The fused instruction will produce `a2` and drop `a1`. The window is admissible only if `a1` is absent from $live\_after$, meaning no instruction reads `a1` before it is next written, whether later in the same block or in any block reached afterwards, so dropping it changes nothing. If instead `a1` is in $live\_after$, some later instruction reads the value the chain discarded, whether the next instruction after the window, code further along, or a subsequent loop iteration, so the candidate is rejected and left unfused.

4.3.3.2 Candidate-Pattern Search

With those analyses in hand, the search slides the window over the basic blocks, but only those whose execution count from the trace exceeds a frequency threshold, skipping the rarely executed blocks where a fused pattern would recoup little, as Algorithm 5 states. The threshold is user-defined, and this work uses a value of ten. For each selected block, the search slides a window of two to four instructions. The window may be wider than the maximum fused sequence because it is only a search region, allowing the algorithm to look past an intervening non-participating instruction and still extract a data-dependent chain. The extracted chain is a dependent pair, and its up to three register operands fit the three operand fields of the chosen encoding. Any instruction in the window that does not belong to the chain remains in the program unchanged. Each extracted chain is then checked against the admissibility conditions of Section 3.3.2.2, using the branch-target and liveness information prepared above. Candidates that pass are recorded under their data-flow signature for aggregation in the next step.

Algorithm 5 Immediate-pattern candidate search.

```
1: B ← branch targets;   L ← liveness sets;   C ← block execution counts
2: for all basic block b with C[b] above the threshold do
3:     for n ← 2 to 4 do
4:         for all windows w of n consecutive instructions in b do
5:             if an instruction of w after the first is in B then skip w
6:             if w has no data-dependent chain then skip w
7:             if the fused latency of w is not below the original then skip w
8:             if w multiplies outside the core's multiplier forms then skip w
9:             if the operands of w exceed the ports then skip w
10:            if an intermediate of w is live afterwards by L then skip w
11:            sig ← data-flow signature of w
12:            record w under sig and accumulate its immediate values
13:        end for
14:    end for
15: end for
```

4.3.3.3 Pattern Aggregation

A validated window is one occurrence of a pattern, and many windows across the binary realise the same pattern with different registers and constants. Aggregation collapses each window into a signature, its sequence of operators with operands abstracted away and commutative operands placed in a canonical order, which is the pattern function $f(s)$ of Section 3.3 computed on the window, so that two windows computing the same operation yield the same signature. The signature is combined with the window's port requirements, the count of external input and output registers obtained by classifying the window's registers against the liveness sets, into an aggregation key, which keeps occurrences that share an operator sequence but need different read or write ports as separate patterns, since they require different hardware. Occurrences are grouped under this key, producing the discovered pattern set $\mathscr{P}$ of Section 3.3.3, with the number of occurrences in the disassembled binary recorded as the static count $\mathrm{occ}(p)$ and the immediate values accumulated per operand position. This static count ranks the patterns by the instruction reduction each would yield, and the accumulated immediates become the distinct constants that the encoding step below must represent.

4.3.3.4 Encoding the Immediate Value

The methodology distinguishes a fixed immediate, the same across every occurrence of a pattern, from one that varies, and this work realises the two cases differently. A fixed immediate is written directly into the operation the fused instruction computes, so the constant becomes a property of the instruction that the encoding need not carry. A varying immediate cannot be carried directly, since the encoding has no room for a full machine word, so it is represented by an identifier drawn from a finite domain D. On CV32E40P that domain is realised by reusing the R4-type register fields the pattern leaves free as identifier carriers. Each field is five bits wide, so $|D| = 32$ per field,

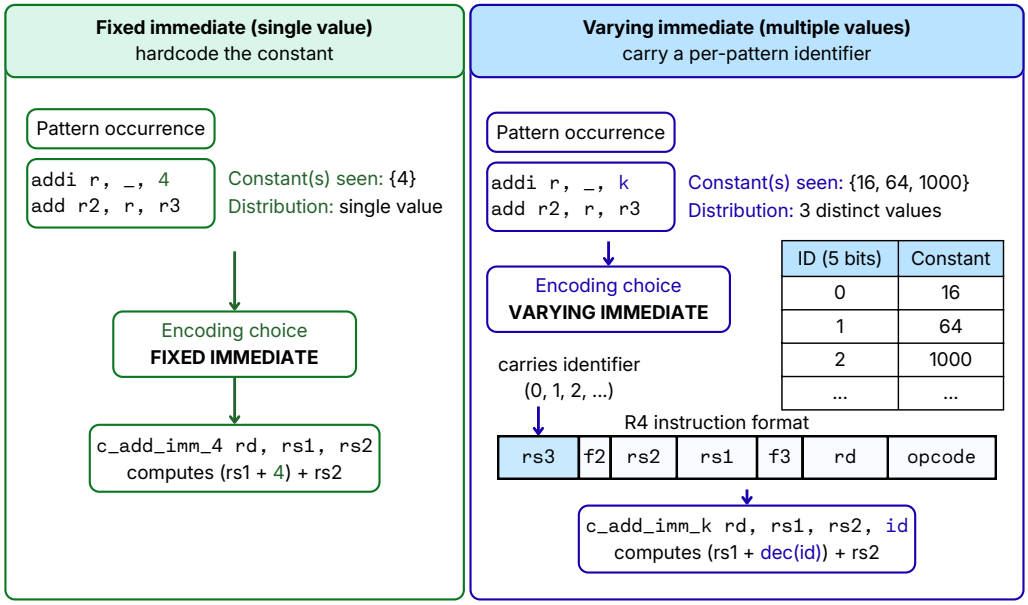


Figure 4.11: Encoding the immediate of a fused instruction. A constant that is the same across every occurrence of the pattern is written directly into the operation (left). A constant that varies is represented by a compact five-bit identifier carried in a free R4-type register field, which selects the workload-specific constant from a per-pattern table, so constants larger than the five-bit field can still be represented (right).

and the identifier transported there selects one of thirty-two distinct constants of any magnitude. A pattern with one parametric immediate uses one such field, and a pattern with two independent parametric immediates uses two, subject to the operand-port limit, so that the encoding carries the identifiers and not the values. Thus, a pattern whose workload immediate values are 16, 64, and 1000 assigns them the identifiers 0, 1, and 2, and one `define_insn` entry is generated per distinct constant. Figure 4.11 contrasts the two cases.

The assembly template emits the identifier into the register slot through the template syntax. The template writes a fixed register name (such as `x2`) in the slot reserved for the identifier. Writing a fixed register name in this way removes the register allocator's freedom over that field, and GCC also sees the instruction as reading a register at that position, which can interact with allocation decisions for nearby code. Figure 4.12 shows the resulting entry for the `addi` + `add` pattern with the immediate 1000 mapped to identifier 2.

At run time the fused operation's execution logic decodes each identifier back into its original constant, as the RTL integration of Section 4.3.5 shows.

4.3.4 Final Pattern Selection and Compiler Generation

The candidate set $\mathscr{P}$ is gathered from the two discovery sources. The static register-register passes are run first against their prebuilt images, and only the patterns each pass actually emitted, recovered from the disassembled binaries, are carried forward, which already discards the unused majority. The immediate-pattern discovery above contributes the rest. Together they can exceed

```
;; addi+add with imm = 1000 -> identifier 2
(define_insn "fused_addi_add_id2"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (plus:SI (match_operand:SI 1 "register_operand" "r")
                          (const_int 1000))
                 (match_operand:SI 2 "register_operand" "r")))]
  "TARGET_CUSTOM_FUSED"
  ".insn r4 0x0b, 4, 1, %0, %1, %2, x2"
  [(set_attr "type" "arith")])
```

Figure 4.12: A `define_insn` entry for the `addi` + `add` pattern with immediate 1000. The expression tree hardcodes the constant, and the assembly template emits the identifier 2 in the third register field via the `x2` register name. Distinct immediate values produce one entry each, with the identifier and not the raw constant reaching the encoded instruction, so the five-bit field carries the identifier even though it could not encode the constant value directly.

the 128 encodings the R4-type custom space provides, so a selection must reduce $\mathscr{P}$ to a set $\mathscr{S}$ with $|\mathscr{S}| \leq 128$ before the compiler is built, the selection rule of Equation 3.8.

The candidates are ranked by the static benefit $\mathrm{occ}(p)\,(n_p - 1)$ of Equation 3.8 and the R4-type slots are assigned in descending order of it, so when the candidates exceed the capacity K the allocation stops at the last free slot and the patterns dropped are those of least static benefit. The static occurrence count is the ranking figure, since it is the one measure available to both discovery sources, the register-register patterns and the immediate occurrences alike, so the two are ranked on a common scale.

The machine description holding $\mathscr{S}$ is the one from which the final GCC toolchain image is built. Unlike the static-pass images, this merged image is specific to the workload, since it carries exactly the patterns that the workload uses. Compiling the workload once more against it produces the final fused binary, in which the compiler applies the fusion operator F wherever a member of $\mathscr{S}$ is selected.

4.3.5 RTL Integration of Custom Instructions

The integration sites identified by the methodology materialise on CV32E40P in four RTL files, the package, the decoder and the ALU. A fifth concern, the immediate decoding bus, threads through the decoder and the ALU. Each site is bracketed by the same pragma markers that the pruning transformations use, the difference being that in this case, the bracketed region is rewritten to add the generated logic instead of removing unused logic.

4.3.5.1 Package-File Extension for Fused Operators

The core's package file declares the `ALU_OP` operator enumeration that selects the operation in the result multiplexer of the ALU and the `MUL_OP` enumeration that selects the multiplier mode. Each selected fused pattern adds one new member to the `ALU_OP` enumeration when the operation

does not match an existing one. Multiplier-routed fusions add no new logic, reusing the existing `MUL_MAC32` and `MUL_MSU32` members.

4.3.5.2 Decoder Integration of Fused Instructions

The decoder gains case arms under the `CUSTOM-0` through `CUSTOM-3` opcodes. Each fused instruction matches an opcode plus a `funct3` and `funct2` value chosen during the encoding allocation. The case arm sets the control signals the rest of the pipeline depends on. The register read enables determine which operands the register file delivers in the next clock cycle. The operator field selects the entry added to the `ALU_OP` or `MUL_OP` enumeration, and the write-back enable is set so that the result reaches the destination register identified by the `rd` field. For patterns whose multiplier path is reused, the decoder asserts the multiplier-enable signal and configures the multiplier operator instead of the ALU operator, reusing the existing multiplier modes with only their encodings changed.

4.3.5.3 ALU Integration and Datapath Reuse

A selected pair becomes a single new arm of the ALU's result multiplexer. The operator the decoder sets chooses that arm, and the arm computes the pair's combined value as one combinational expression, inside the cycle the ALU already takes. Building the arm has three steps, composing the pair's two operations into one expression, expressing each operation in SystemVerilog, and mapping the result onto the datapath.

The arm nests the pair's two operations. The first operation reads the pair's source registers, and the second reads the first operation's result together with its own remaining operand. Whether that running result enters the second operation as its first or its second operand is what separates the forward and reverse variants of Section 4.3.2, and it matters only for the non-commutative operations such as subtract. The source operands are wired from the ALU's inputs `operand_a_i` and `operand_b_i`, from `operand_c_i`, the third read port the R4-type encoding adds, or from an immediate where the pattern carries a constant.

Each base mnemonic maps through a fixed table to a SystemVerilog expression over its two operands. Most are direct, `add` to `(a + b)` and the bitwise operations to their operators. Three families are written with care so that synthesis keeps the width and sign correct. A shift masks its amount to the low five bits, the range that addresses a bit of a thirty-two-bit word [6]. An arithmetic right shift is expressed as a logical shift of a sign-extended value rather than with the signed-shift operator, so it does not depend on the signal being declared signed. A signed comparison uses explicit `$signed` casts, zero-extended to the word width.

Emitting the nested expression as fresh logic would rebuild an adder or shifter the core already owns, paying area for hardware that already exists. The generator instead steers the existing datapath. A pair whose only costly step is one add or subtract reuses the thirty-two-bit adder, with the cheap surrounding logic feeding the adder's two inputs, the result multiplexer reading its output, and a flag selecting subtraction, so no second adder is built. A pair whose shift amount is fixed

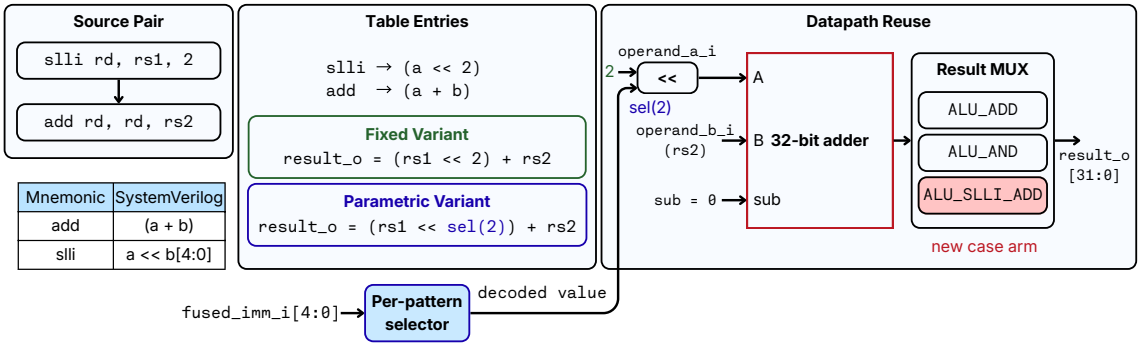


Figure 4.13: Mapping the `slli` + `add` pair to RTL. The source mnemonics select two table entries, which nest into one result-multiplexer arm with `rs1` on `operand_a_i` and `rs2` on `operand_b_i`. Because the pair has a single add preceded only by a constant shift, the existing adder is steered rather than duplicated. When the shift amount is parametric, it travels as an identifier on `fused_imm_i[4:0]` and is recovered by the per-pattern selector `sel` before indexing the shifter.

routes the barrel shifter the same way, since a constant shift is wiring and not logic. A pair built only from logic and comparisons, which the result multiplexer already evaluates, emits its expression directly. Only a pair that needs two adds or subtracts, which one adder cannot serve at once, emits the surplus step inline.

Figure 4.13 works this through for a `slli` + `add` pair, from the source mnemonics to the composed result-multiplexer arm and the adder it reuses, and shows how the pair changes when its shift amount is parametric instead of fixed.

A parametric immediate reaches the hardware through a small bus, `fused_imm_i`, that the decoder drives from the instruction word, routing the five bits of the third register field, `rs3`, onto `fused_imm_i[4:0]` when a parametric case arm is active, and the second register field, `rs2`, onto `fused_imm_i[9:5]` for a pattern with a second immediate. The composed expression in the result multiplexer reads these slices where the literal would otherwise sit, and a per-pattern selector, the inverse of the discovery-time assignment, maps each identifier back to its thirty-two-bit constant before the operation applies it. When a pattern uses only one constant across the workload, the bus and selector are dropped and that constant is written into the expression as a literal.

4.4 Hardware Loops

The hardware-loop methodology requires a hardware-loop unit on the target core. CV32E40P already provides such a unit, but its depth and body restrictions admit only a small fraction of the loops a compiler emits, which limits the saving the strategy can deliver. The unit is therefore modified to widen what it accepts, and the modified unit is described first, since its capabilities and costs determine which software loop shapes can be converted and what each iteration pays at run time. The software path that produces and patches those loops follows. Figure 4.14 lays out the

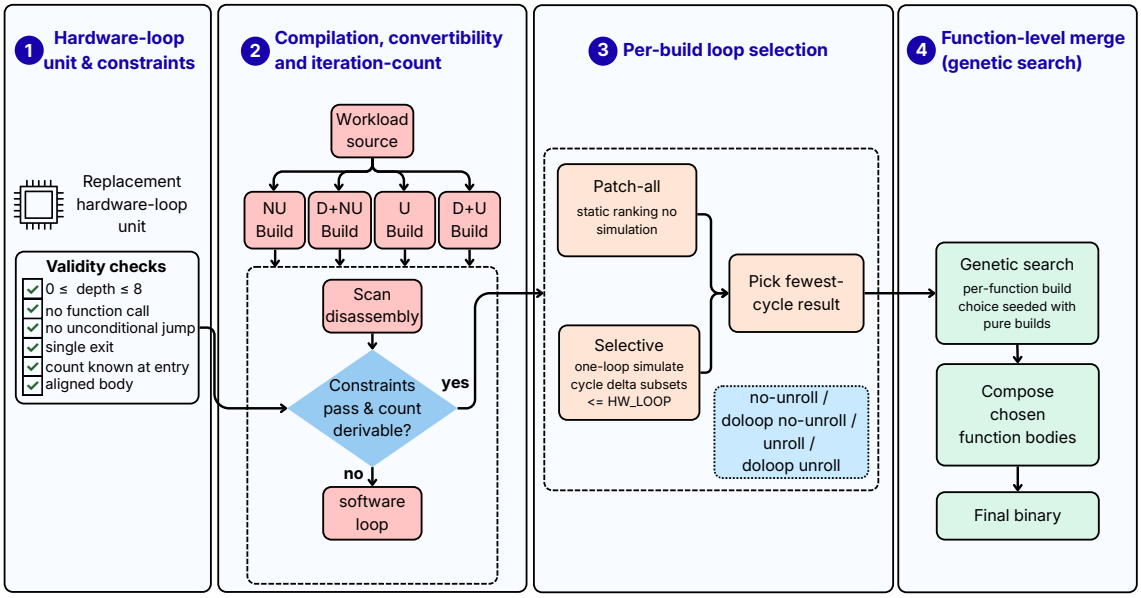


Figure 4.14: The hardware-loop specialisation flow on CV32E40P. The replacement unit sets the eligibility constraints. The workload is compiled four ways, the cross of loop unrolling on or off with the doloop hint on or off, giving the *NoUnroll*, *Unroll*, *DoLoop+NoUnroll*, and *DoLoop+Unroll* builds. For each build the patcher scans the disassembly, admits a loop only when every constraint passes and the iteration count is derivable, and otherwise leaves it as a software loop. Each build is then patched under two selection strategies, patch-all by static ranking and selective by per-loop simulation, and the build of fewest clock cycles is kept. A function-level genetic search finally composes each function's body from whichever build is best.

full flow, from the unit and the constraints it sets, through the four compilations and the per-loop convertibility and count derivation, to the per-build selection and the function-level merge that produces the final binary.

4.4.1 The Replacement Hardware-Loop Unit

The upstream CV32E40P inherits a hardware-loop unit from the PULP RI5CY core [28]. Two of its choices block its use as a specialisation knob. The depth is fixed at two levels, so every core pays for a second level even where its deepest convertible nest is one level, and no workload can exploit a third. In addition, the body constraints are tight, in particular forbidding any branch, jump, or compressed instruction inside the body.

The replacement unit is largely this work's own. Because the stock unit is hardcoded, with thirty-two-bit widths and a fixed two-level depth, the configurability is itself new, with the depth and both widths made synthesis-time parameters. This work also adds the per-level register file that holds each active loop's state, the shadow counter that reloads a nested inner loop automatically, and the acceptance of compressed instructions in the body, which the stock unit forbids. Its configuration is the tuple $\theta_{\mathrm{HL}} = (d, w_{\mathrm{cnt}}, w_{\mathrm{addr}})$ of Section 3.4.2, the depth d realised as the synthesis-time `HW_LOOP` parameter from 0 to 8, the counter width w_{cnt}, and the loop-address width w_{addr} sized to the binary. The values these can take form the admissible space Θ_{HL}. As

Table 4.4: The CV32E40P hardware-loop constraints and how the replacement unit treats each.

Upstream constraint	Treatment	How
No write to the unit's own CSRs in the body	Lifted	The replacement does not expose loop state as writable CSRs, so no in-body write can occur.
Depth fixed at two levels	Lifted	Depth becomes the synthesis-time `HW_LOOP` parameter, from 0 (unit removed) to 8, sized to the workload's deepest convertible nest.
No branch or jump in the body	Relaxed	Conditional branches and software inner loops are admitted. Unconditional jumps and branches onto the loop's back-edge stay excluded.
No compressed instruction in the body	Relaxed	Compressed instructions are admitted everywhere except the last body instruction, which is forced to four bytes.
Body at least three instructions	Kept	Retained unchanged from the upstream design and enforced in the patcher.
Body start and end addresses 32-bit aligned	Kept	The patcher emits the alignment directives.
Entered only from the start address	Kept	Enforced as a static eligibility predicate.
Single architectural exit	Kept	Inherent to a counted loop, whose only exit is the count expiring. Enforced as a static eligibility predicate.
Iteration count known before entry	Kept	Inherent to a counted loop, whose count register is written once before the body runs. Enforced as a static eligibility predicate.
No `fence` or `fence.i` in the body	Kept	Enforced as a static eligibility predicate.
No privileged instruction except `ebreak`/`ecall`	Kept	Enforced as a static eligibility predicate.

Section 3.4.2 sets out, the two widths are sized as the pruning strategy of Section 3.2 sizes any parameter, to the smallest value the converted loops need, while the depth is selected per workload by the merit defined below and reported in Chapter 6. Table 4.4 lists each upstream constraint [29] and how the replacement treats it. Lifted constraints are removed by added hardware, relaxed ones are lifted for the cases the hardware can handle and kept for the rest, replaced ones are exchanged for a different condition serving the same purpose, and kept ones are enforced unchanged.

The lifted and relaxed rows are where added or modified RTL does the work, in five SystemVerilog modules. The register file holds the per-level loop state, the controller detects the loop end and pulses the counter decrement, the prefetch controller and the aligner cooperate to wrap the fetch stream back to the start before the body's last instruction retires. Figure 4.15 shows the five modules and the signals that pass between them, and each module is described in turn below.

4.4.1.1 Per-Level Loop Register File

Each level holds five values. The start address a_{start} and the end address a_{end} bound the body's instruction range. The counter holds the remaining iteration count and decrements once per iteration. The shadow counter captures the count's setup-time value and reloads the counter automatically when an outer level decrements while the current level did not, so a nested inner loop does not need

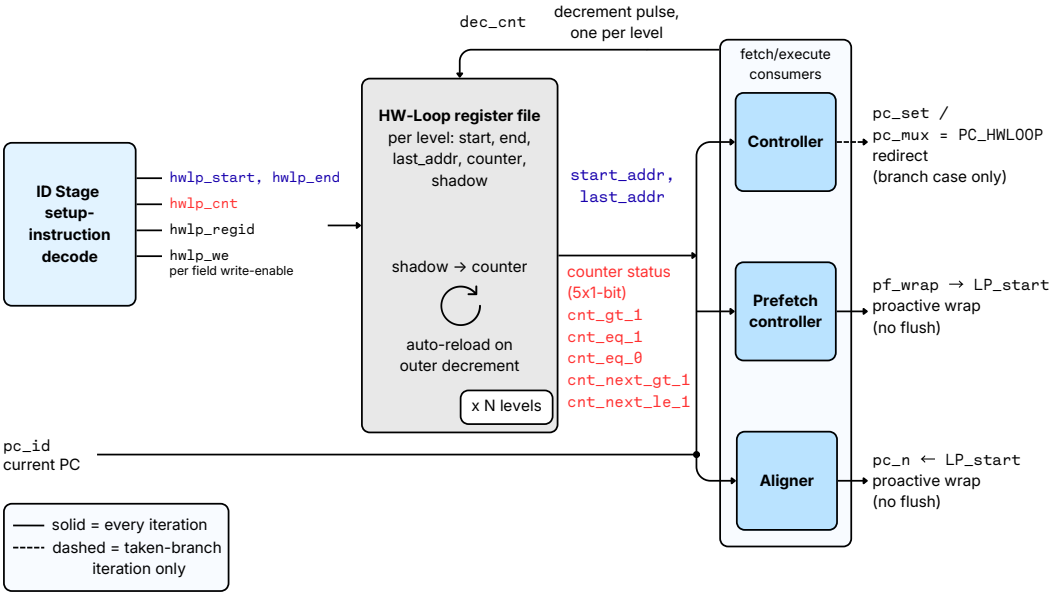


Figure 4.15: The five modules of the hardware-loop unit and the signals between them. The ID stage decodes the setup instructions and writes the per-level register file, which exports the start and last-instruction addresses and five one-bit counter-status flags to the controller, prefetch controller, and aligner. The controller returns a per-level decrement pulse. On a normal iteration the prefetch controller and the aligner wrap the fetch stream back to the loop start with no flush, and the controller issues an explicit redirect only when an in-body branch is taken (dashed). The register file is replicated across the `HW_LOOP` levels, and its shadow counter reloads an inner level automatically when an outer level decrements.

to be re-initialised on every outer iteration. The fifth value is the address of the last body instruction, pre-computed at setup time as the end address minus four bytes, since the patcher forces that instruction to a full four-byte width. Pre-computing it lets the controller detect the loop end with a bare comparison against the program counter. The counter's bit-width is the parameter w_{cnt}, tuned per benchmark, since the loops of different workloads have different iteration counts and the bits saved by a narrower counter propagate through every register that carries it. The shadow counter is only instantiated when the depth parameter is greater than one.

4.4.1.2 Loop End-Detection Controller

The controller compares the program counter against the pre-computed last-instruction address. On a match with the counter still above one, it pulses that level's decrement. A deferred-decrement mechanism handles the case where the body's last instruction is a conditional branch, as occurs when the body contains a software inner loop's back-edge. The decrement is held until the branch outcome resolves in the execute stage, then committed when the branch falls through. Without the deferred decrement, a body-internal branch would force the controller to decrement on speculation, so a branch-taken outcome would corrupt the iteration count.

4.4.1.3 Loop Wrap-Ahead in the Prefetch Controller

The prefetch controller redirects the fetch stream back to the loop start before the body's last instruction is fetched, so the start of the next iteration is already in the prefetch buffer when the current iteration's last instruction commits. A late-wrap flush discards stale entries in the corner case where a level's counter is written after fetches have already passed its end address, so the body is not executed with an offset window.

4.4.1.4 In-Fetch Loop Wrap in the Aligner

The aligner performs a parallel proactive wrap inside the fetch path. For each level it tests whether the program counter lies in the body range and, when the counter is greater than one and the next program counter would reach the end address, overrides the next program counter with the loop's start address in the same clock cycle. The aligner keeps this duty alongside its ordinary alignment of compressed and uncompressed instructions, and the patcher's forcing of the last body instruction to four bytes ensures the end address lands on a 32-bit boundary, so the wrap-back fetch never straddles a halfword.

4.4.1.5 Build-Time Assembly of the Hardware-Loop Unit

The hardware-loop unit is controlled by a single top-level parameter, `HW_LOOP`, that sets its depth, in the same way the pruning parameters control the core's optional units. When no loops are convertible, this depth is set to 0, removing the unit from the sources. The pragma-marked hardware-loop logic embedded in shared modules such as the aligner and the prefetch controller is then

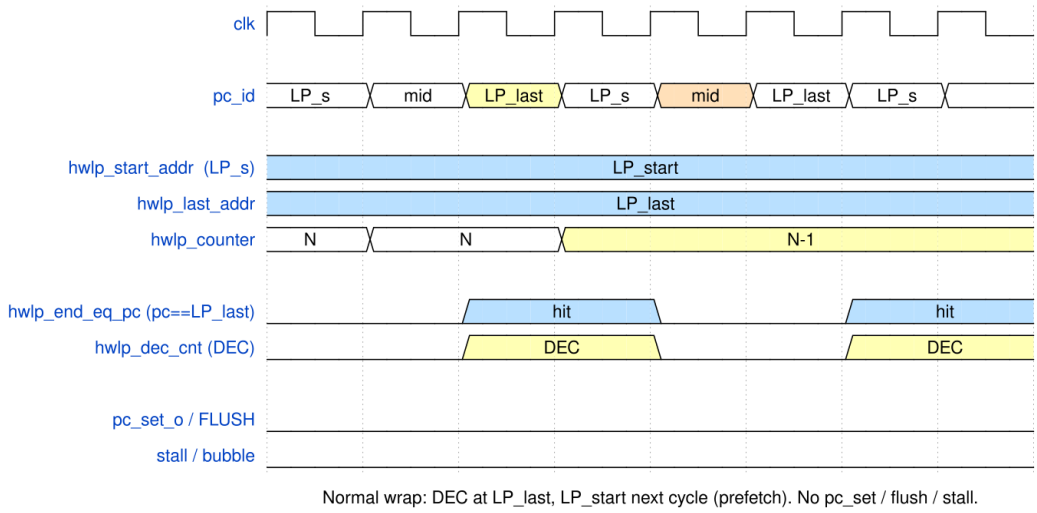


Figure 4.16: A normal iteration. The fetch-stage wrap steers the next-fetch program counter to the loop start before the program counter reaches the last body instruction, so at the last instruction the controller only decrements the counter, with no flush or stall.

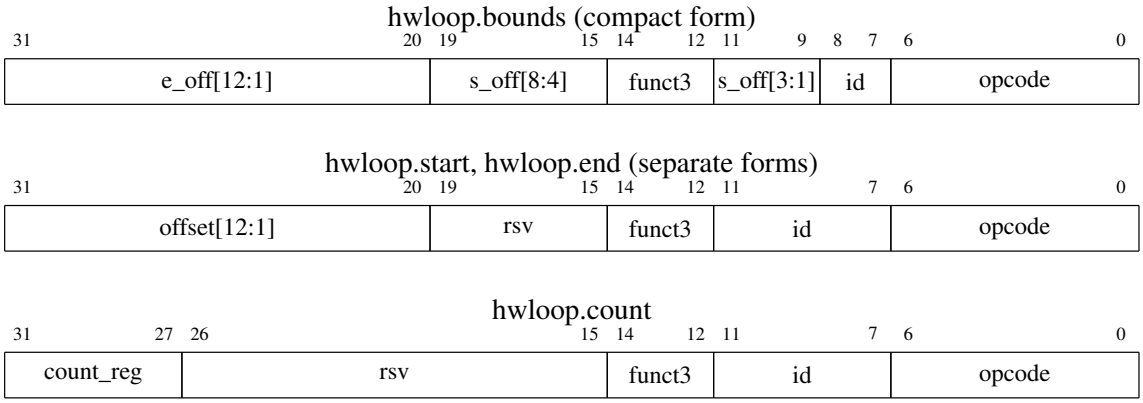
stripped out, and the decoder gains no hardware-loop case arms, so no area is wasted on a workload that does not benefit from it.

4.4.1.6 Run-Time Behaviour of the Hardware-Loop Unit

The five modules together produce two run-time behaviours, the common zero-overhead iteration and the exceptional branch iteration.

The common case, which carries the per-iteration saving, is a body that runs straight through, shown by the waveform of Figure 4.16. The prefetch controller and aligner wrap the fetch stream in the front-end, as described above, so the next iteration is already in the buffer and nothing has to be discarded. The controller's only action at the last instruction is to pulse the counter decrement, issuing none of the flushing redirect the stock core would use to reach the loop start by running past the end address and discarding the instructions in flight. This is the zero-overhead wrap the strategy is built on.

The zero-overhead wrap relies on the body falling through to its last instruction, the fall-through the prefetch controller and aligner predict. A taken branch inside the body invalidates that prediction. When a body branch is taken, the program counter jumps somewhere other than the next sequential address, so the instructions the prefetcher already wrapped in are no longer the ones that will execute, and the speculative wrap is wrong. The unit cannot use the prefetched stream, so it falls back to detecting the end address directly and issues an explicit program-counter redirect to the loop start. That redirect discards the wrongly-prefetched instructions, which is the flushed clock cycle. The counter decrement is held until the branch outcome resolves and then committed, so a branch that is taken cannot corrupt the count. One flushed clock cycle is thus charged to each iteration whose final executed instruction is a taken branch, and that cost is the



Each row is one 32-bit word, scaled to the text width. Bit positions run from 0 on the right to 31 on the left, `rsv` are reserved bits

Figure 4.17: Field layouts of the four directive types the patcher emits, with *start* and *end* sharing one layout. The compact *bounds* form packs an eight-bit start offset (`s_off`) and a twelve-bit end offset (`e_off`) into a single word. Both are byte offsets relative to the directive's own address with an implied zero low bit, so the stored fields span bits [8:1] and [12:1]. When either offset overflows its slot, the patcher falls back to separate *start* and *end* directives whose twelve-bit offsets cover the full range. The *count* directive places the iteration-count register identifier in the rs3 slot. The loop identifier (`id`) selects the hardware-loop level the directive configures and occupies the rd slot, the full five bits in the *start*, *end*, and *count* forms and two bits in the compact *bounds* form once the packed offsets are placed.

price of admitting the conditional branches and software inner loops the upstream unit forbids outright.

4.4.2 Setup-Instruction Encoding

Before a converted loop's body runs, the unit's registers must be loaded with the loop's boundaries and iteration count, and the setup instructions carry those values into them. The patcher emits four such directive types into the assembly, shown in Figure 4.17. The *bounds* directive configures both the start and end addresses of a loop in a single thirty-two-bit word, with an eight-bit start offset and a twelve-bit end offset, both relative to the directive's own address. Packing both offsets leaves only two bits for the loop identifier, so the compact form addresses just the first four hardware-loop levels. When either offset exceeds its allotted range, or the loop identifier needs more than two bits because the unit is configured with more levels, the patcher emits separate *start* and *end* directives instead, each carrying a twelve-bit offset and the full five-bit identifier, so wider bodies and deeper nests remain expressible. The *count* directive transfers the iteration count from a general-purpose register into the unit's loop-counter register.

4.4.3 Loop Eligibility and Iteration-Count Derivation

A loop detected in the assembly is converted only if it passes a sequence of static predicates on the disassembled body. The loops the disassembly scan detects form the candidate set $\mathscr{L}_W$ of Sec-

tion 3.4.1, and these predicates are the concrete admissibility predicate $A_{\text{tgt}}(\ell, \theta_{\text{HL}})$ of that section, true for a candidate ℓ exactly when every check below holds under the chosen configuration θ_{HL}. The Kept rows of Table 4.4, single-exit, an iteration count known before entry, and the absence of a fence or privileged instruction, are each enforced by one such predicate. Two further checks are specific to this design. The body must contain no unconditional `j` jump other than the back-edge, since the prefetch path wraps the program counter speculatively from the sequential next address and has no signal to suppress that wrap for an unconditional jump. Software inner loops with conditional back-edges, in contrast, are admitted, because the deferred-decrement path handles them.

The most involved of these predicates is the requirement that the iteration count be computable in the setup region. The implemented algorithm consumes three values recovered from the binary. The induction variable is identified from the back-edge branch. For a two-operand branch such as `bne rs1, rs2, .Lstart`, both operands are tested as candidates and the one that is written exactly once per iteration by a constant step is accepted. For a single-operand branch such as `beqz` or `bnez`, the operand is the induction variable and the bound is implicitly zero. The step magnitude is extracted from the induction variable's self-update instruction, taken from the immediate field of an `addi` or from the value of a loop-invariant register summed in by an `add` or subtracted by a `sub`. The bound is the other operand of the back-edge branch when one is present, or zero otherwise. The induction variable's initial value and the bound's value must each be *pre-computable*, in the sense that the value can be traced through the pre-loop region to a constant load, a move from another pre-computable register, pure arithmetic on pre-computable operands, or a function-argument register that is not overwritten before the loop. Loads from memory are accepted as pre-computable when their base address is itself pre-computable, since the loaded value is then stable at the loop's entry. When the bound register is modified inside the body by a descendant loop's induction variable, the modification is accepted if that descendant's bound is itself pre-computable, since the descendant's induction variable holds its bound at completion.

Given the inputs, the iteration count N_ℓ of loop ℓ, the value that Section 3.4.3 carries into the profitability model, follows the formula

$$N_\ell = \left\lceil \frac{|bound - init|}{|step|} \right\rceil . \tag{4.1}$$

The setup code that the patcher emits to compute N_ℓ is selected from one of three families that minimise the number of instructions and avoid `divu` where possible. Compile-time evaluation folds the count to a single `li` whenever the patcher can evaluate Equation (4.1) itself, which it can when both the distance from the initial value to the bound and the step are constants. This covers a loop whose initial value and bound are both literals, a loop whose initial value is a fixed offset from the bound, such as `i = N - 96`, where the distance is that constant offset whatever the runtime value of the bound, and the nested case where the bound is held in a register that resolves through a descendant loop to a literal. Strength reduction replaces division by a power-of-two step with a logical shift, drops the count instruction altogether when the step is one and the initial

Table 4.5: Iteration-count derivation by loop pattern. Three optimisation families avoid the multi-cycle `divu` instruction by exploiting the loop's algebraic structure or by reusing a register that already holds the count value. The fallback at the bottom of the table is taken only when no cheaper realisation exists.

Loop pattern (C-like)	When applies	Emitted setup	Cost
Compile-time evaluation. Known at the compiler's symbol table, so the count is folded to a literal load.			
`for (i = 5; i < 100; i += 7)`	Both endpoints are literals.	`li count, 14`	1
`for (i = N - 96; i < N; i += 4)`	init is bound minus a constant offset, bound unchanged before the loop.	`li count, 24`	1
Strength reduction. Division is replaced by cheaper arithmetic when the step has algebraic structure.			
`for (i = 0; i < N; i += 4)`	step is a power of two, init is zero.	`srli count, N, 2`	1
`for (i = 5; i < N; i++)`	step is one, init is a literal.	`addi count, N, -5`	1
`for (i = 5; i < N; i += 4)`	step is a power of two, init is a literal.	`addi count, N, -5` `srli count, count, 2`	2
`for (p = start; p < end; p += 4)`	pointer-stepping with a power-of-two element size.	`sub count, end, start` `srli count, count, 2`	2
`for (i = N; i > 0; i -= 4)`	countdown to zero, init in register, step is a power of two.	`srli count, N, 2`	1
Register reuse. An existing register already holds the count value, so no setup instruction is emitted.			
`for (i = 0; i < N; i++)`	N is read-only in the body, count equals N.	(reuse N register)	0
`for (i = N; i > 0; i-)`	init in register, induction variable read-only in the body apart from its self-update.	(reuse IV register)	0
Fallback. The step has no algebraic structure to exploit, so the iterative divider is invoked.			
`for (i = 0; i < N; i += 7)`	step is a non-power-of-two runtime value.	`li tmp, 7` `divu count, N, tmp`	2

`for (i = X; i < N; i += k)`	general case, both endpoints and step are runtime values, k a non-power-of-two.	`sub count, N, X`
		`li tmp, k`
		`divu count, count, tmp`
			3

value is zero (the bound register already holds the count), and combines a subtraction with a shift when the step is a power of two and the initial value is non-zero. Register reuse names an existing register directly as the count when the count happens to equal a register's current value, including the special case of countdown loops where the induction variable already holds the count at entry. When none of the families apply, the algorithm falls back to emitting `divu`, the only path that pays the cost of the iterative divider. Table 4.5 summarises the algorithm by listing the loop patterns each family handles, the setup code emitted, and the instruction cost.

4.4.4 Patching: Setup-Code Emission and Body Transformation

The patcher transforms the merged binary's assembly file in a single pass. For each selected loop, it inserts a setup block immediately before the loop's start label, or, where the loop's count can be computed earlier, in an enclosing loop, removes the back-edge branch at the loop's end, and removes the induction-variable self-update instruction when the variable is used only for loop control. The setup block contains the count-derivation instructions, then the *bounds* directive that encodes the start and end addresses, then the *count* directive that loads the iteration count.

The count derivation often needs a temporary register. Choosing that register is the same liveness problem the fusion path solves for its window writes, decided against the *live_in* and *live_out* sets of the backward data-flow analysis of Section 4.3.1, evaluated at the loop's setup point. A register is free to borrow exactly when it is not live across the setup, in the sense that it is not read after the setup before being written again. The patcher first looks among the integer temporary registers for one not used anywhere in the body, since a body-unused temporary is absent from the setup point's live set and can be written without disturbing later instructions. If no body-unused temporary is found, it considers callee-saved registers that the function's prologue has already saved, since their original values are preserved on the stack and will be restored by the epilogue regardless. The first instruction of the body is also tried as a free target when its destination register is overwritten before being read, which is exactly the condition that it is not in the setup point's live set either. Only when none of these candidates is available does the patcher fall back to spilling. A spill saves a temporary to the function's existing stack frame when free space is available, or grows the frame by sixteen-byte-aligned bytes for the duration of the setup otherwise, with a matching restore emitted after the setup. The fallback ensures correctness on the small fraction of loops where no register can be borrowed, at the cost of two memory accesses per setup execution.

The compressed-instruction handling is realised by forcing the surviving last body instruction to four bytes after the patcher removes the induction-variable self-update. The transformation makes the loop's end address land on a 32-bit boundary, which the prefetch path handles cleanly without the half-word straddle a compressed last instruction would introduce on the wrap-back fetch from the start address.

For nested loops, the patcher hoists each inner loop's setup block into the outermost setup block whose enclosing iteration leaves the inner setup's parameters unchanged. The invariance check reuses the pre-computability analysis above, with hoisting accepted when every register read by the inner setup is unchanged across outer iterations. When the check fails, the inner setup remains adjacent to its loop's start label and runs on every outer-loop iteration. The hoisting works because the register file's shadow counter automatically reloads the inner counter when an outer level decrements, so an inner loop set up once before the outer loop runs at full count on every outer iteration. A branch redirection ensures that any branch above the loop that would target the loop's start label is redirected to the setup block's label instead, so the unit's registers are always initialised before the body executes. This redirect requires the setup to sit immediately before the

Before: software nest

```
li t0, 8              // L0 iteration count
.L0_start:
  li t1, 4            // L1 iteration count
.L1_start:
  ...
  addi t1, t1, -1     // L1 step
  bnez t1, .L1_start  // L1 branch
.L1_end:
  ...
  addi t0, t0, -1     // L0 step
  bnez t0, .L0_start  // L0 branch
.L0_end:
```

After: patched hardware loops

```
li t0, 8
li t1, 4            // hoisted (invariant)
.L0_setup:
  hwloop.bounds     // [.L0_start,.L0_end]
  hwloop.bounds     // [.L1_start,.L1_end]
  hwloop.count t0   // L0 count
  hwloop.count t1   // L1 count
.L0_start:
.L1_start:
  ...
                    // L1 step removed
                    // L1 branch removed
.L1_end:
  ...
                    // L0 step removed
                    // L0 branch removed
.L0_end:
```

Figure 4.18: Assembly transformation for a two-level hardware loop with an invariant inner iteration count. Left, the original software nest with two back-edge branches and two induction-variable updates. Right, the patched code. The inner loop's count setup is hoisted above the outer loop because it is invariant across outer iterations, the two setup blocks are merged before the shared loop start, and both back-edge branches and induction-variable updates are removed. The inner end label is padded to at least six bytes from the outer end label so the controller does not wrap both loops on one fetch clock cycle.

loop, so a loop entered by such a branch cannot be hoisted, since a hoisted setup would send the redirected branch into the enclosing loop. Figure 4.18 illustrates the transformation on a two-level nest with an invariant inner count.

4.4.5 Selecting Loops for Conversion

A workload may expose more eligible loops than the hardware-loop depth allows, and not every eligible loop is worth converting even when the unit could host it. Conversion is not free, since the setup sequence is paid once on every entry while the back-edge branch and counter-control instructions it removes save only a few clock cycles per iteration. By the profitability model $\widehat{G}_\ell = E_\ell\,(N_\ell\, r_\ell - \sigma_\ell)$ of Equation (3.9), a loop that runs too few iterations between entries can raise the clock cycle count instead of lowering it, and a loop whose iteration count cannot be read from the static code, unlike one fixed by a load-immediate, cannot even be judged in advance. Additionally, within a nest the unit hosts only as many loops as its depth, so when a nested group holds more eligible loops than the available depth of one or two, only a subset can be converted.

Selection consequently poses two distinct problems. The first is individual, whether converting a given loop saves clock cycles at all. The second is grouping, which loops within a nest to convert together. Both resist exact modelling, because beyond the setup instructions, the removed branches, and the loop bodies, the patcher pads loop ends and shifts the surrounding code, and that change in alignment alone moves the clock cycle count. Two methods address the selection, the static *patch-all* and the measured *selective*, each seeking the selection $\mathscr{L}^\star$ of Equation (3.10)

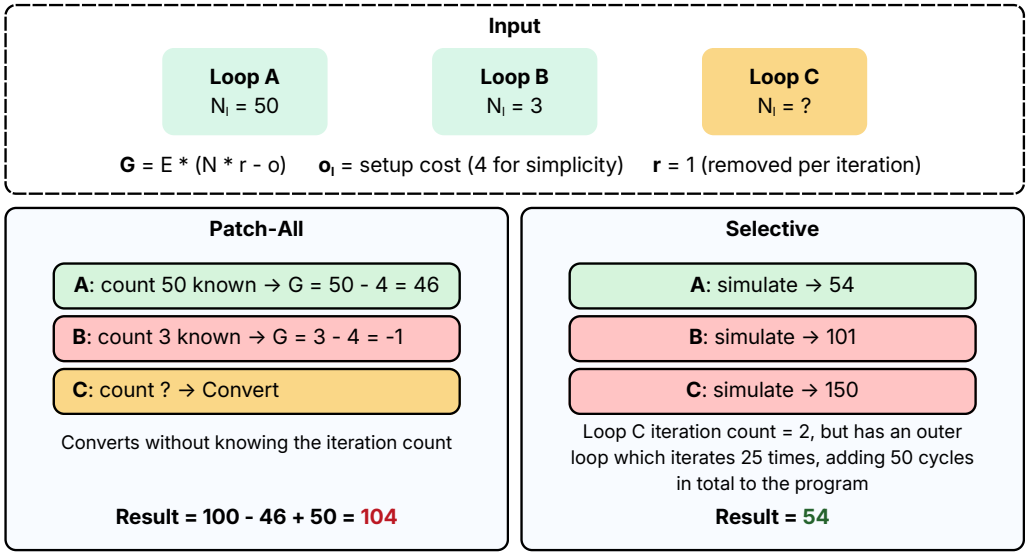


Figure 4.19: Patch-all versus selective on three loops, comparing the static estimate $\widehat{G}_\ell$ with the simulated clock-cycle count. Loop A has a known and profitable trip count, B a known but unprofitable one, and C a trip count unknown at compile time. Patch-all drops B by its estimate but converts C blindly, and since C's setup is never repaid its result exceeds the unconverted baseline, while selective simulates each candidate, keeps only A, and reaches the fewest cycles. The values are illustrative.

that minimises the clock cycle count S of the transformed binary. Figure 4.19 contrasts the two on a small example.

Patch-all with static per-nest selection. The *patch-all* method converts every eligible loop and drops only those it can show in advance cannot repay their setup, so it defaults towards conversion. A loop whose iteration count cannot be read from the static code is converted regardless, which means patch-all may convert loops that in the end save nothing, or even raise the clock cycle count. For the grouping it ranks each nest's loops by the static estimate $\widehat{G}_\ell = E_\ell\,(N_\ell\, r_\ell - \sigma_\ell)$ of Equation (3.9). The per-entry term $N_\ell\, r_\ell - \sigma_\ell$ weighs the instructions removed per iteration, r_ℓ, the back-edge branch together with the counter-control instructions removed with it when the loop carries them, over the trip count N_ℓ, against the setup cost σ_ℓ, the count-derivation instructions of Table 4.5 and the count directive paid once per entry. This term is then multiplied by the entry count E_ℓ, taken as the product of the trip counts of the enclosing loops, so each loop's saving is scaled by how often the nest re-enters it and the multiplicative effect of nesting is captured. With every loop in the nest scored, the method keeps the highest up to the unit depth. Loops whose iteration count cannot be determined from the binary tie, and the tie is broken in favour of the innermost, on the assumption that it is entered most often and so saves most. That assumption can be wrong, a loop reached only when a rare branch is taken is scored as though it ran at the full nested count, but patch-all accepts the inaccuracy in exchange for needing no simulation.

Selective per-nest enumeration. The measured method replaces the static estimate with the clock cycle delta each candidate actually produces under simulation with Verilator [40]. Every eligible loop is first converted on its own and simulated, and any loop that fails or does not save cycles is dropped. Within each nest every subset of size up to the unit depth is then built and simulated, and the subset of lowest S is kept, so the loops chosen are the ones that measurably lower the clock cycle count and the group with the largest saving is kept. Measuring the delta avoids the misjudgements of the static estimate, but the method has a limitation of its own. Adding a loop shifts the surrounding code, so a loop whose end address is aligned under one selection may be misaligned under another, and the clock cycles it saves change with it. A loop's contribution is therefore not a fixed number but depends on which loops are converted alongside it, which is why the subsets are evaluated whole rather than scored one loop at a time.

4.4.6 Compiler Doloop Conversion

This work adds a compiler pass that shapes eligible loops to suit the patcher. Without it, the iteration-count derivation of Section 4.4.3 reconstructs each loop's count from the disassembled binary after the compiler has finished, which may take several instructions to compute. The pass removes that reconstruction for the loops it reaches by rewriting each into a countdown form, a counter decremented to zero, so the count is already in a register at the loop's entry and the patcher reads it directly. The same shaping enlarges the candidate set $\mathscr{L}_W$ of Section 3.4.1, since a loop whose count the assembly scan could not derive becomes convertible once the compiler has produced it. The pass is a patch to the GCC RISC-V backend, built into a dedicated toolchain container and enabled by a new `-mhwloop` flag. Compiling with the flag produces the *DoLoop* builds of a workload, against the non-DoLoop builds from the stock compiler, and since the flag is orthogonal to loop unrolling it applies on top of either the unrolled or the non-unrolled compilation, which is what gives the four builds the merge of Section 4.4.7 composes.

Shaping a loop this way also constrains the compiler. Once a loop is committed to countdown form, the optimiser can no longer restructure it in some other, possibly faster, way, so a DoLoop build can convert more loops yet run slower overall than an unconstrained build.

The patch reuses GCC's target-independent hardware-doloop pass, which already knows how to rewrite a counted loop into countdown form but stays dormant for RISC-V because the stock backend gives it no way to end a loop with a counted branch. The patch supplies that capability, gated on the flag, through two target hooks and a `doloop_end` pattern. The hooks decide which loops the pass may transform. The first, `TARGET_CAN_USE_DOLOOP_P`, returns true when the flag is set, the loop is entered at its top, and its nesting depth is at most one. The second, `TARGET_INVALID_WITHIN_DOLOOP`, rejects any loop whose body contains a call or a table jump, since those break the fall-through the unit relies on.

Besides the two hooks, the patch adds three entries to the option and machine-description files, shown in Figure 4.20. The option entry declares the `-mhwloop` flag and binds it to a target mask bit, so that `TARGET_HWLOOP` is true exactly when the flag is given. The `doloop_end` expander is the gate the doloop pass tests for, its presence under `TARGET_HWLOOP` telling the pass the

```
mhwloop
Target Mask(HWLOOP)

; riscv.md -- advertise the capability to the doloop pass
(define_expand "doloop_end"
  [(parallel [(set (pc)
                   (if_then_else
                    (ne (match_operand:SI 0 "register_operand") (const_int 1))
                    (label_ref (match_operand 1 ""))
                    (pc)))
              (set (match_dup 0) (plus:SI (match_dup 0) (const_int -1)))])]
  "TARGET_HWLOOP"
  { if (GET_MODE (operands[0]) != SImode) FAIL; })

; riscv.md -- emit the countdown loop end the pass selects
(define_insn "*doloop_end_si"
  [(set (pc) (if_then_else
               (ne (match_operand:SI 0 "register_operand" "+r") (const_int 1))
               (label_ref (match_operand 1 "" "")) (pc)))
   (set (match_dup 0) (plus:SI (match_dup 0) (const_int -1)))]
  "TARGET_HWLOOP"
  "addi\t%0,%0,-1\;bnez\t%0,%l1"
  [(set_attr "type" "branch") (set_attr "length" "8")])
```

Figure 4.20: The three entries the `-mhwloop` patch adds, one in `riscv.opt` and two in `riscv.md`. The option entry binds the flag to the `HWLOOP` target mask. The `doloop_end` expander, present only under `TARGET_HWLOOP` and restricted to a 32-bit counter, is the capability the doloop pass gates on. The `*doloop_end_si` instruction emits the countdown loop end, `addi` to decrement the counter and `bnez` to branch back while it is non-zero.

target can end a loop with a single counted-branch instruction. The matching `*doloop_end_si` instruction is what the pass actually emits, a counter register decremented by one and a branch back to the loop top while it stays non-zero, printed as the two-instruction `addi`/`bnez` sequence.

With the hooks and the pattern in place, the doloop pass runs inside the compiler's own loop optimiser, where it still sees the loop before register allocation and scheduling obscure it. It stops at the countdown form and does not emit the unit's setup directives. The assembly patcher of Section 4.4.4 recognises that form and converts it through the same emission path as a loop it found itself, so the flag is a front-end to the patcher and not a replacement.

Figure 4.21 shows the difference on one source loop, the NoUnroll build leaving the patcher to reconstruct a count from a walking pointer while the DoLoop+NoUnroll build materialises the count in a register for it to read.

4.4.7 Build Merge Across Compilation Variants

The clock cycles a hardware loop saves depend on how many loops are converted and on the saving each conversion brings. That saving grows with a loop's iteration count, since the one-off setup is diluted over more passes of the body, so a loop that runs many iterations repays its setup while others may regress. The compilation flags move both terms. Unrolling acts on the iteration count, a fully unrolled loop no longer existing for the unit to take and a partially unrolled one running

C source

```
for (i = 0; i < n; i++) {
  sum += a[i];
}
```

NoUnroll build

```
  slli a1, a1, 2  // n*4
  mv   a5, a0     // p = a
  add  a1, a0, a1 // e=a+n*4
  li   a0, 0
.L3:
  lw   a4, 0(a5)
  addi a5, a5, 4  // p++
  add  a0, a0, a4
  bne  a5, a1, .L3 // p != e
```

DoLoop+NoUnroll build

```
  slli a5, a1, 2
  addi a5, a5, -4
  srli a5, a5, 2
  addi a5, a5, 1  // cnt = n
  mv   a4, a0
  li   a0, 0
.L3:
  lw   a3, 0(a4)
  addi a4, a4, 4
  add  a0, a0, a3
  addi a5, a5, -1 // cnt--
  bnez a5, .L3    // cnt!=0
```

Figure 4.21: The same counted loop in the NoUnroll and DoLoop+NoUnroll builds, as emitted at `-O2` for this example. The NoUnroll build strength-reduces the index to a walking pointer and ends with `bne` comparing that pointer against a precomputed end address, so the patcher must reconstruct the iteration count as $(end - start)/4$. The DoLoop+NoUnroll build instead materialises the count in a register before the loop and ends with the `addi`/`bnez` countdown, which the patcher converts directly without reconstructing the count.

fewer iterations and so saving less, while the number of convertible loops depends on how many the patcher can derive an iteration count for, which the doloop pass of Section 4.4.6 aims to widen. Unrolling and the doloop pass are independent, so each workload is compiled four ways, with each of the two on or off, giving the *NoUnroll*, *Unroll*, *DoLoop+NoUnroll*, and *DoLoop+Unroll* builds.

The four builds each pass through the same patch-all and selective selection of Section 4.4.5, and the loop-converted build that runs in the fewest clock cycles becomes the base of the merge. The merge is needed because the four compilations produce different assemblies, so the build that is fastest overall need not hold the fastest version of every function, as some may run quicker in another build. Working at the assembly level, the merge starts from the base and, function by function, takes the body from whichever build runs fastest, so a function keeps its base body unless another build improves it. Only the functions whose body differs across the builds are choices to make, since one compiled identically everywhere leaves nothing to decide.

Taking a function's body from another build requires its local labels not to collide with the base's. The patcher namespaces the spliced function's numeric local labels, rewriting each `.Ln` to a per-build prefixed form, and leaves named labels such as `.LANCHOR0` untouched, since those refer to symbols in the base build's data section and renaming their references would dangle. The per-nest loop selection for each function is keyed by the function name and the loop's label, not by the back-edge instruction text, because the splice rewrites that label and a text key would no longer match the loop.

The number of whole-program assignments grows as four raised to the count of differing functions, so a program with twenty of them admits 4^{20} assignments, far beyond enumeration. The merge therefore searches with a genetic algorithm, shown in Figure 4.22, whose genome is the per-function choice of source build. It is seeded with the base alone, so the result is at least as good as that single build. Each trial assembles and simulates the merged and patched program

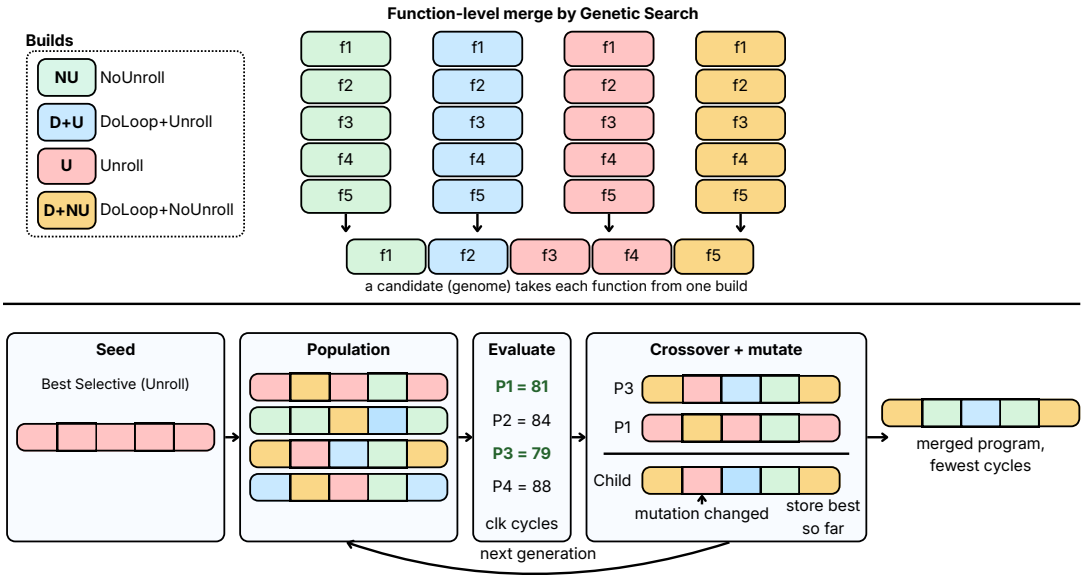


Figure 4.22: Function-level merge searched by a genetic algorithm. A candidate, the genome, names one source build for each function. The search starts from the single fastest build, evaluates each candidate by assembling and simulating it for its clock cycle count, and evolves the population by crossover and mutation of its fittest members, keeping the best assignment seen until no further improvement or a budget is reached. Colours and numbers are illustrative.

and feeds its clock cycle count back as the fitness. The best assignment is kept as the search runs, which stops once several generations pass without improving it, or a fixed budget is spent.

4.4.8 Hardware-Loop Depth Selection

The depth d of Section 3.4.2, the `HW_LOOP` parameter, sets how many levels of a nest the unit can take, so it bounds the loop selection and not only the hardware. The patch-all and selective conversion of Section 4.4.5 and the merge of Section 4.4.7 are therefore run once for each candidate depth. A workload nested two levels deep passes through the whole pipeline at depth one and again at depth two, and depth zero, with the unit removed, is a candidate as well. Each depth yields its own merged and converted program, and the one kept is chosen per benchmark by a merit that trades the clock cycles a depth saves against what the unit costs at that depth. A deeper unit tracks more nesting levels and so converts more of a deep nest, but each added level grows the unit, which occupies more area, draws more power, and can lengthen the critical path, so a clock cycle saving can be offset by the lower frequency the added logic brings. Depth zero suits a benchmark whose loops save too few clock cycles to justify even a single level of the unit, removing the unit.

The merit combines the two axes by normalising each against the reference configuration and taking their geometric mean,

$$S(d) = \sqrt{\tilde{A}(d) \cdot \tilde{C}(d)}, \qquad \tilde{A}(d) = \frac{A(d)}{A_{\text{ref}}}, \qquad \tilde{C}(d) = \frac{C(d)}{C_{\text{ref}}}, \tag{4.2}$$

so that lower is better and $S = 1$ reproduces the reference. The area $A(d)$ is the Yosys [41] cell count, a technology-independent measure of logical complexity that ranks configurations without committing to a standard-cell library, and the clock cycle count $C(d)$ is the merged result simulated with Verilator [40]. The reference is the depth-zero core, with no hardware loop unit, so $\tilde{A}$ measures each depth's area against removing the unit entirely, while C_{ref} is the lowest clock cycle count of the four software builds, so $\tilde{C}$ credits the unit only with the saving it provides.

The normalised geometric mean is scale-invariant, so the cell-count proxy and the clock cycle count enter on equal footing despite their different magnitudes. Minimising it prefers a depth that improves one axis without giving up a comparable amount on the other. The selected depth is $d^{\star} = \arg\min_d S(d)$, taken over depth zero and every level the benchmark's nests expose.

4.5 Combining the Optimisations

The three RTL transformations are analysed independently, but their edits all reach the same core and interact, so the combined variant cannot simply concatenate them. Two aspects must therefore be handled, the order in which the edits are applied and the conflicts between them.

The conflicts arise because a choice one strategy makes can invalidate an assumption another relies on. Custom-instruction fusion claims part of the custom-encoding space, and a three-input fused instruction also claims the third register-file read port. Without this coordination, pruning would treat both as unused and remove them, so the encodings and the read port that fusion occupies must be excluded from what pruning removes. Hardware loops raise the same conflict, since the loop setup instructions reuse a custom opcode, one of which is three-input, so that opcode and the read port must survive even when no ordinary instruction needs them. The dependence also runs the other way. A fused instruction that reuses an existing arithmetic block, such as the multiplier behind a multiply-accumulate, keeps that block live even when no original instruction selects it, so pruning cannot remove it. An operation that fusion absorbs into a compound instruction can leave its original encoding unused, which pruning is then free to remove.

The order follows from these dependences. Fusion settles its encodings first, since both the pruning decoder and the loop setup instructions need to know which custom opcodes are taken. Pruning is then resolved against that result, so it never removes logic a fused or loop instruction still reaches. The chosen loop depth is propagated into the pruning configuration so that the synthesis parameters agree on the unit's size. The encoding and datapath-width reductions are applied last, since they depend on which logic survived every earlier edit.

All of the RTL edits are reconciled and applied together, over a copy of the core's sources and in the fixed order above. Performing every edit once, on one workspace, keeps the strategies' analyses independent of one another while still producing a coherent combined core.

Chapter 5

Software Architecture of ARVIS

Chapter 3 defines the specialisation methodology, and Chapter 4 realises its optimisation strategies for CV32E40P. This chapter is the software counterpart to that implementation, and its concern is architectural, how ARVIS, the tool that orchestrates those strategies, is organised around the parts that vary, the target core, the optimisation strategies, the removable features, and the custom-instruction forms. ARVIS is designed to outlive a single experiment. It is demonstrated here on one core with one toolchain, but the same flow is meant to apply again, to another core, with another strategy, and an architecture that admitted those changes only by editing the orchestration would undermine that aim.

5.1 Design Goals

ARVIS is built as a structured Python program instead of a collection of ad hoc transformations. Python suits the task because the work is orchestration-heavy, invoking external compiler, simulation, and synthesis tools, parsing their textual and structured output, generating source files, and managing per-experiment directories.

Three goals shape the design. The first is that an optimisation strategy should be replaceable or extendable without disturbing the rest of the flow, since a strategy may change its analysis, for instance from a static instruction count to an execution trace, or a new strategy may be added entirely. The second is that target-specific knowledge should sit behind explicit interfaces, so that supporting a new core is a matter of implementing the operations the strategies require rather than of editing the strategies. The third is that the changes the tool makes should be inspectable, so that a user can see which features were removed, which parameter values were selected, and which custom instructions were generated, rather than only the final synthesis numbers.

The first two goals concern where knowledge lives and what may be replaced, and they motivate a hexagonal architecture, also known as ports and adapters [42]. A kernel of abstract interfaces names the roles the framework needs, and concrete implementations are supplied as adapters that depend on the kernel but not on one another. The third goal is met separately, by recording each

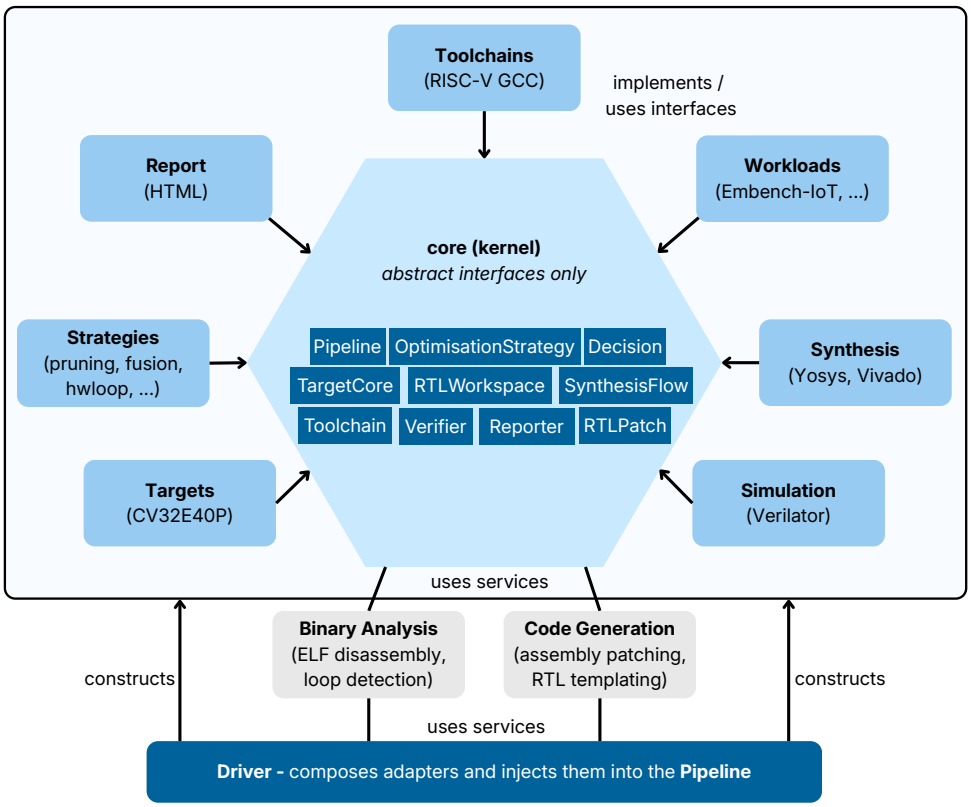


Figure 5.1: Ports-and-adapters organisation of ARVIS. The kernel defines the abstract roles, adapters implement them, and the driver composes a concrete run configuration. Dependency arrows point inward, towards the kernel.

change as it is made, and is taken up in Section 5.6. The remaining sections present this organisation, the interfaces that make it extensible, the records that cross between stages, and the flow that ties them together.

5.2 Architectural Organisation

ARVIS is built around a central kernel, shown in Figure 5.1. The kernel, drawn as the hexagon at the centre, holds only abstract interfaces and names the system's roles. The adapters arranged around it realise each role with a concrete implementation, and every one of them depends inward on the kernel, not on its neighbours. For example, the target core is one such role, an abstract interface in the kernel, and CV32E40P is the adapter that implements it. A driver below the hexagon, the command-line entry point, constructs the adapters and injects them into the pipeline, which is the only place that names concrete implementations.

The kernel coordinates a run and owns no target-specific or tool-specific knowledge. It prepares the workload, asks each selected strategy for its decisions, turns those decisions into edits through the target, and drives the downstream tools, each step expressed against an interface, not a concrete implementation.

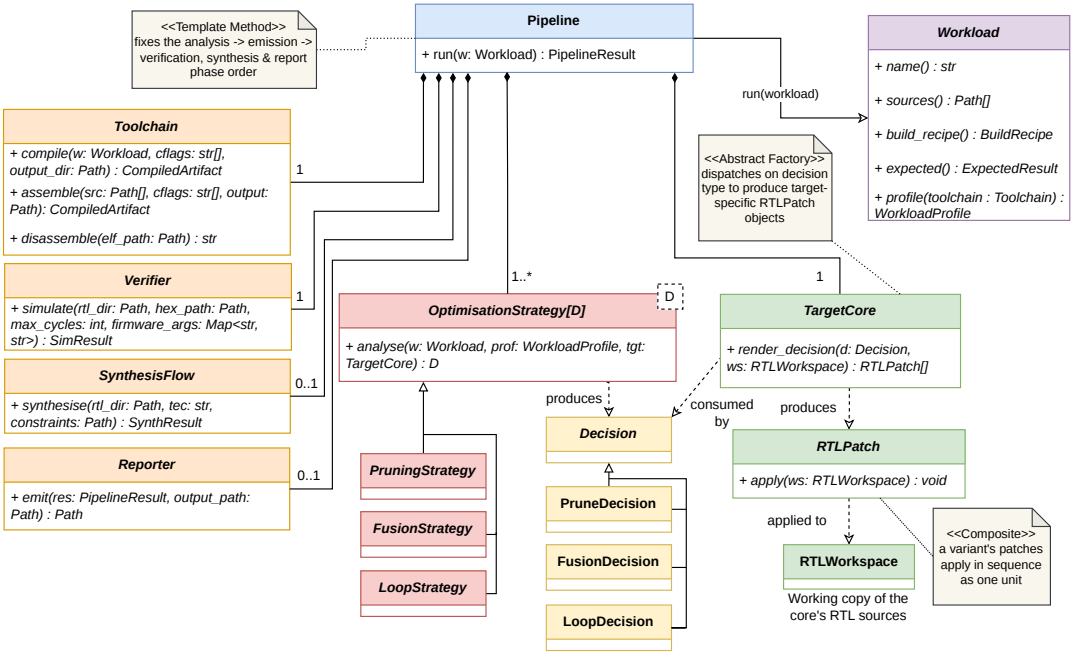


Figure 5.2: The principal kernel roles of ARVIS and how `Pipeline` composes them. Each concrete `OptimisationStrategy` (pruning, fusion, hardware loops) produces its paired `Decision`, and `TargetCore.render_decision` turns a target-agnostic `Decision` into target-specific `RTLPatch` objects applied to an `RTLWorkspace`.

Because those dependencies point only inward, every concept crossing the boundary is expressed in target-agnostic and strategy-agnostic terms, and swapping one adapter for another, or adding a new strategy, target, or tool, is a change confined to the driver, with the kernel untouched.

5.3 Interfaces and Extension Points

The kernel defines its roles as abstract base classes, and the extension points of ARVIS are exactly these interfaces. Figure 5.2 shows the roles needed to follow the flow. Besides the optimisation strategy and the target core, the kernel names a workload, a toolchain, a verifier, a synthesis flow, and a reporter, each an abstract port that the driver fills with a concrete adapter. The toolchain, verifier, synthesis flow, and reporter are uniform service interfaces with one implementation per tool, so the two that carry the architectural weight, the optimisation strategy and the target core, are the ones described below.

5.3.1 Optimisation-Strategy Interface

Each strategy follows one contract. It declares whether it applies to a given target, examines the workload, and returns a typed decision describing what should change about the core. It does not itself edit the RTL, which is a later, separate step. The pairing of a strategy with its decision is expressed in the type system, as `OptimisationStrategy[D]` is a generic abstract base

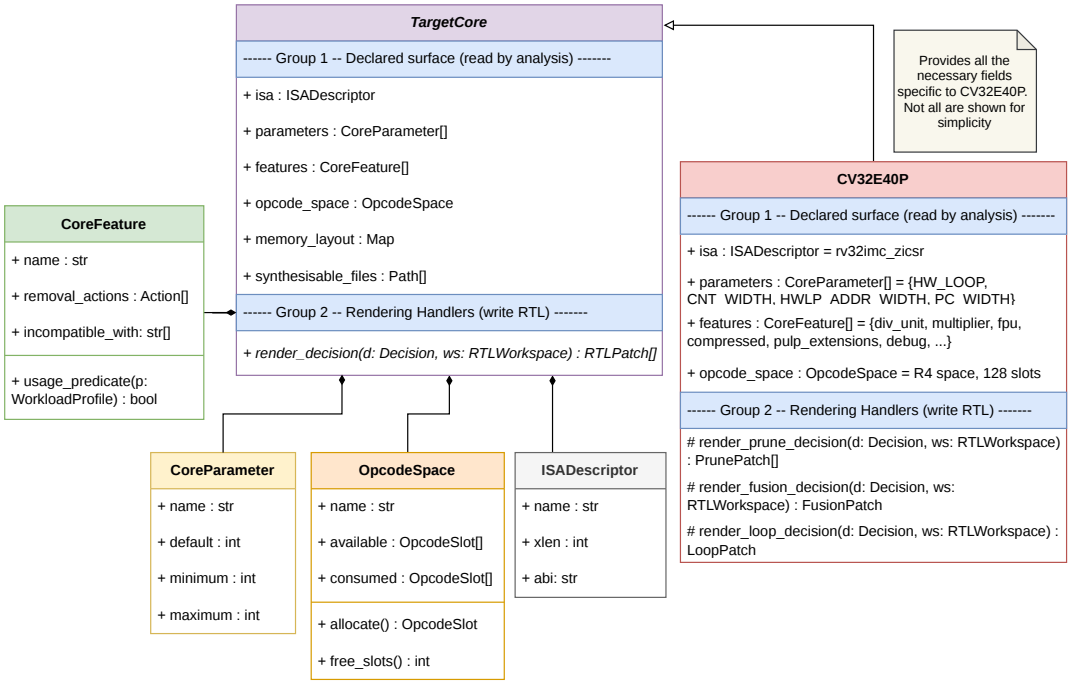


Figure 5.3: The CV32E40P target as a concrete `TargetCore`. The abstract interface exposes a declared surface that analyses read, together with the per-kind render handlers that realise their decisions. The CV32E40P subclass populates both.

parameterised by the decision type `D` it returns, bounded by the common `Decision` base. The three concrete strategies, for pruning, fusion and hardware loops, each fix `D` to their own decision type. This makes the intended pairing explicit and allows static analysis to detect mismatches, such as a pruning strategy returning a fusion decision. A new strategy is added by implementing this contract and returning its decision, with the pipeline unchanged because it drives every strategy through the same abstract interface.

5.3.2 Target-Core Interface

A target core is represented by an object that exposes the operations the strategies need, with the target-specific detail held behind the interface. The interface has two parts. One is the declared surface that analyses query before any edit is made, comprising the parameters the core exposes and their legal ranges, the catalogue of removable features, the custom-opcode space available for fusion, and a description of the ISA. The other is the set of rendering handlers that turn a decision into RTL edits. A concrete core is defined by supplying both. Each part of the surface is read by one analysis, pruning reads the feature catalogue, fusion the opcode space, and the loop and binary analyses the ISA and memory layout. Figure 5.3 shows CV32E40P as one such instance.

The single entry point into the rendering handlers is the target's `render_decision` method. It takes a decision and the workspace, looks at which kind of decision it is, a pruning, fusion, or loop decision, and forwards it to the matching handler that the concrete target supplies, one

handler per decision kind. That handler is the only code that knows how a decision of its kind becomes an edit to this particular core, and it returns the list of edits to apply to the workspace. Keeping this dispatch on the target lets the kernel remain ignorant of file paths, parameter names, and SystemVerilog grammar, the property the architecture exists to preserve. The external tools are also kept behind separate small interfaces. A run therefore supplies only the toolchain, verifier, and synthesis flow components it needs, so a development run that only inspects generated RTL does not need to construct the simulation or synthesis adapters.

5.4 Core Domain Models

Python is not statically typed, yet the stages of ARVIS must agree on the shape of the data they exchange. The implementation therefore moves information between stages as explicit records with a defined schema, instead of as untyped dictionaries or side effects on shared state. A record is validated where it is produced, so a malformed value is rejected at its source rather than surfacing later in an unrelated stage. Each optimisation has its own schema. A pruning decision carries the set of features found unused and the evidence behind that finding. A fusion decision carries the selected instruction patterns and the encoding assigned to each. The loop decision carries the converted loops. The remainder of this section describes the two schemas that shape the rest of the flow, the pruning feature catalogue and the fusion pattern record.

A target describes each removable feature as data, not as code. A `CoreFeature` pairs a name with the feature trigger of Section 3.2.1, here a predicate over the workload profile, and a tuple of removal actions, and may also declare features it depends on or conflicts with. The pruning analysis walks the catalogue, evaluates each trigger, and records the unused features in a `PruneDecision`.

Each removal action names a kind drawn from a closed enumeration of generic transformations, such as setting a parameter, narrowing a parameter, deleting a marked block, removing a case arm, or removing a signal. The framework provides one primitive per kind. Adding a removable feature to a core is therefore a matter of adding one catalogue record, without adding per-feature code to the pruner, because the catalogue and the pruner meet only at the kind enumeration, which both agree on and neither owns.

The custom-instruction model shows the same benefit of a shared representation. A fusion candidate records the base operations it fuses, the operand mapping, the per-occurrence immediate values, the encoding it is assigned, and the static benefit used to rank it. The same record is read by both artefact generators, one emitting the compiler machine-description entry and the other the package, decoder, and execution-path additions of Section 4.3. Holding the pattern once and deriving both sides from it keeps the fused binary and the specialised hardware aligned, instead of maintaining two separate descriptions that could diverge.

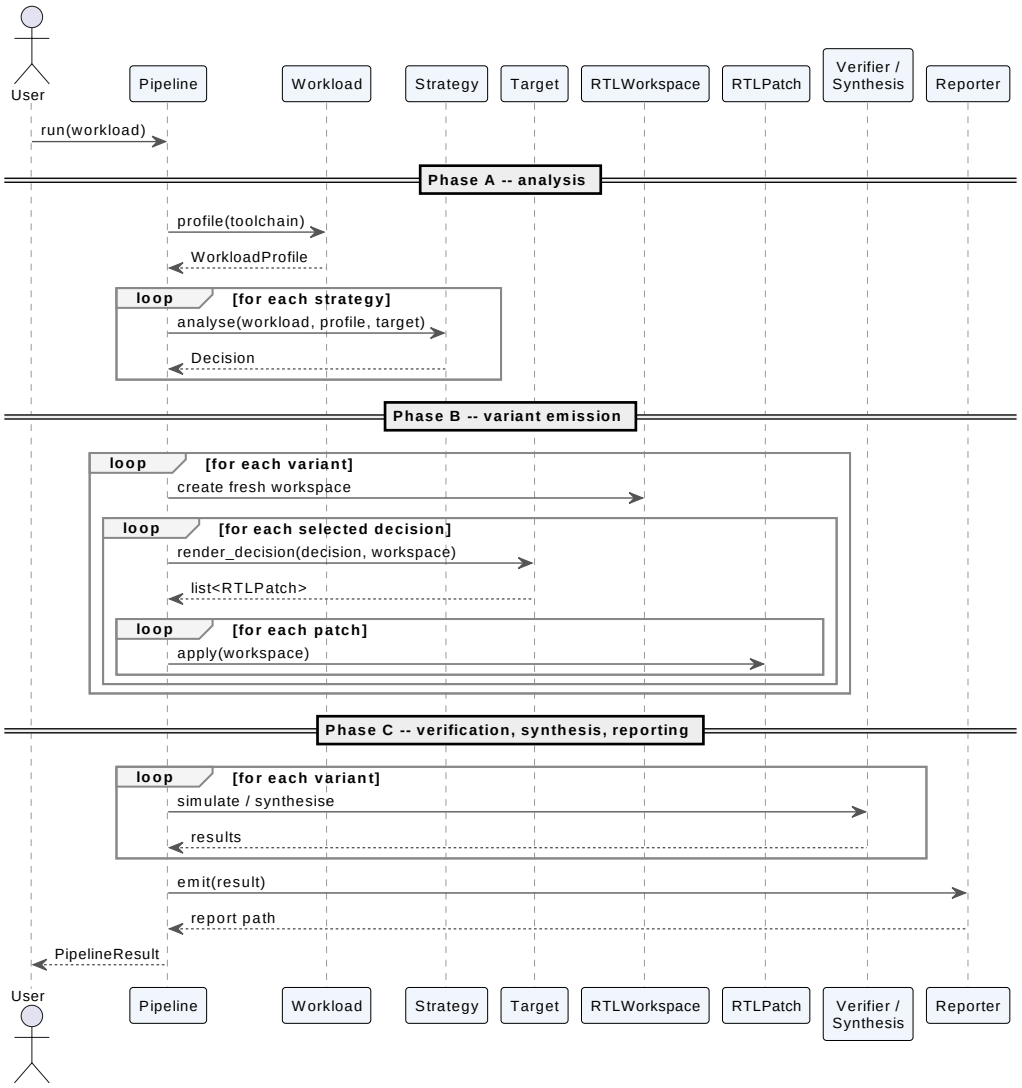


Figure 5.4: Pipeline execution sequence. The pipeline first profiles the workload and collects strategy decisions. It then emits each configured variant by rendering the selected decisions through the target and applying the resulting edits to a fresh workspace. Verification, synthesis, and reporting operate on the emitted variants. The `RTLPatch` lifeline stands for each edit a decision produces.

5.5 Specialisation Pipeline Execution

The user chooses which optimisations from the catalogue to run, from all of them down to a single one such as pruning, fusion, or the hardware loops, and which variants to emit from the resulting decisions. Each variant is a named configuration that selects a subset of the collected decisions, and the target offers a standard set as a default, from the baseline through single-strategy variants to the fully combined core, so that the evaluation can compare them. The pipeline separates analysis from emission across three phases, shown in Figure 5.4 and described below.

The first phase runs each applicable strategy once and collects its decision, driving every strategy through the same interface without knowing which concrete one it holds. The second phase emits each variant as a projection of the already-collected decisions, selecting which decision kinds take part, rendering them through the target, and applying the resulting edits to a fresh workspace, a clean copy of the core's RTL sources made for that variant so variants cannot interfere. The third phase verifies, synthesises, and reports on the emitted variants. The edits applied to a variant form an ordered list, each touching one or a small number of RTL files idempotently, so the orchestrator applies them in order without knowing how many a decision produced or which files they touch. Before the edits are rendered, a resolution step reconciles the collected decisions so that the strategies do not undo one another, in the order and with the conflict handling set out in Section 4.5, and the reconciled edits are then applied in that single pass. This is what lets edits from different strategies coexist in the combined variant while each strategy's analysis stays independent of the others.

5.6 Decision Reporting and Observability

Every change ARVIS makes is an edit to the RTL sources, so a user could in principle recover what changed by reading the generated sources or the tool's logs. Nevertheless, for each run ARVIS builds an HyperText Markup Language (HTML) report that gathers the changes into one overview, so the user can understand what was specialised at a glance rather than reconstructing it by hand.

The report shows what was changed and the evidence behind it, the features that were removed, the parameter values that were selected, the custom instructions that were generated, and the variants produced with their measured clock cycles and area estimation. An illustrative report, generated for one benchmark, is reproduced in Appendix C.

5.7 Repository and Reproducibility

ARVIS is released as an open-source Python project [43]. The repository holds the orchestration kernel, the CV32E40P target adapter, and the optimisation strategies. It also carries detailed developer documentation of the software, covering the kernel architecture, the per-strategy flows, and the steps for porting the tool to a new core, alongside the external tool versions the flow depends on.

This chapter described how ARVIS turns the methodology into a working tool. A small kernel of interfaces holds the parts that vary, the target core, the strategies, and the tools, behind explicit ports, so a strategy stays independent of the others and of the core it specialises. Each strategy returns a typed decision, and the target alone knows how that decision becomes RTL edits. A fixed pipeline collects the decisions once and replays them to emit each variant, and a report records what every variant changed. The next chapter evaluates the variants this flow produces, reporting simulation and synthesis results across a suite of benchmarks.

Chapter 6

Experimental Setup and Result Analysis of ARVIS

The previous chapters presented the methodology, its implementation on CV32E40P, and the software architecture of ARVIS. This chapter reports the results obtained on a suite of benchmarks, together with their analysis. It first fixes the baseline against which every optimisation is measured, describes the synthesis and simulation flows and the tool versions used, and defines the metrics for evaluation. The optimisation results follow.

6.1 Baseline Configuration and Hardware Implementation Flow

The original CV32E40P exposes optional features behind synthesis-time parameters, so the baseline disables the ones that would otherwise inflate the apparent gain of pruning. The FPU, the PULP extensions, and the stock hardware loops are disabled, and the number of hardware performance counters is set to one, retaining only clock cycle counting. This baseline is the reference for every result reported in this chapter.

During a specialisation run, ARVIS estimates area with Yosys [41] using its technology-independent cell count, which is fast enough to drive the per-variant search without invoking a full place-and-route. The baseline core synthesises to 38 868 generic cells under this estimate. The final physical results for the strategies are produced by two complete implementation flows, one targeting an ASIC and one an FPGA, so that each optimisation can be assessed on both.

The final physical evaluation uses LibreLane [44], an open-source ASIC flow built on OpenROAD, targeting the SkyWater Sky130A PDK at 130 nm [45] with its high-density standard-cell library, characterised at the typical-typical corner, 25 °C, and a 1.8 V supply. The flow is constrained to a 30 MHz clock and a floorplan core utilisation of 55 %, so the standard cells occupy 55 % of the core area and the remaining area is used for routing and filling. Power is estimated by OpenROAD's static power analysis, which drives a default switching activity from the primary inputs through the netlist with the clock at the target frequency, so no workload trace is supplied. The toggle rate is set to 12.5 %, with a static probability of 50 %.

Table 6.1: Baseline CV32E40P implementation results on both physical platforms.

Platform	Metric	Value
ASIC (Sky130)	Standard cell area	$268\,802\,\mu m^2$
	Core area	$394\,811\,\mu m^2$
	$\widehat{F}_{max}$	34.0 MHz
	Power ($\widehat{P}$)	7.11 mW
	$\widehat{ADP}$	$11\,612 \times 10^3\,\mu m^2$ ns
FPGA (Spartan-7)	Occupied slices	1459
	LUTs	5180
	FFs	2169
	BRAM (36 Kb)	0
	DSP blocks	5
	$\widehat{F}_{max}$	78.87 MHz
	Power ($\widehat{P}$)	66 mW
	Dynamic	50 mW
	Static	16 mW
	$\widehat{ADP}$	18 499 slice × ns

The FPGA flow targets the Xilinx Spartan-7 XC7S15 [46], synthesised with Vivado 2025.2 [47] at a 75 MHz target clock. Synthesis is run with the `AlternateRoutability` directive, retiming, and register retention enabled. Implementation then uses a performance-oriented directive configuration: `ExploreWithRemap` for logic optimisation, `ExtraTimingOpt` for placement, `AggressiveExplore` for physical optimisation and routing, and `AlternateReplication` for the post-route physical-optimisation pass. Power is estimated with Vivado's vectorless method at the same toggle rate and static probability as the ASIC flow.

Table 6.1 summarises the baseline CV32E40P implementation on both platforms, reporting cell and core area, estimated maximum clock frequency, $\widehat{ADP}$, and total power for the ASIC flow, and Look-Up Table (LUT), FF, BRAM, and DSP block counts, estimated maximum clock frequency, total power, and $\widehat{ADP}$ for the FPGA flow. Each optimisation presented in the results sections is compared against these baseline values on both platforms.

Table 6.2 lists the tools used across the ARVIS pipeline and their versions, spanning compilation, simulation, the synthesis area proxy, the RTL processing front-end, and the two physical back-ends.

6.1.1 Evaluation Metrics

Each optimisation is judged on area, timing, power, clock cycle count, and a combined figure of merit. Area is reported in the unit natural to each platform, the core area in μm^2 for the ASIC flow, and the occupied-slice, LUT, FF, BRAM, and DSP counts for the FPGA flow, of which the

Table 6.2: Tools used in the ARVIS pipeline and their versions.

Tool	Version	Purpose
GCC [37]	15.2	RISC-V cross-compiler with per-benchmark fused patterns
Verilator [40]	5.020	Cycle-accurate RTL simulation
Yosys [41]	0.63	Technology-independent synthesis (area proxy)
sv2v [48]	0.0.13	SystemVerilog to Verilog conversion
pyslang [36]	10.0	SystemVerilog AST parser for case-statement pruning
LibreLane [44]	3.0.2	ASIC place-and-route on SkyWater 130 nm
Vivado [47]	2025.2	FPGA place-and-route on Spartan-7 XC7S15
Docker [49]	29.0	Isolated per-benchmark GCC builds
objdump [35]	2.46	Binary disassembly for instruction analysis

occupied-slice count is taken as the area measure. Both flows run a single full synthesis and place-and-route against the platform's target clock. Timing is reported as the maximum clock frequency $\widehat{F}_{\max}$, estimated by re-running static timing analysis on that one implementation under a range of clock constraints. Starting from the target, the constraint is relaxed while it leaves setup violations and shortened once it passes, then refined around the boundary to locate the shortest period the implementation closes, the estimated minimum $\widehat{T}_{\min}$, with $\widehat{F}_{\max}\,[\mathrm{MHz}] = 1000/\widehat{T}_{\min}\,[\mathrm{ns}]$.

This search reuses the placement and routing produced for the target clock and only repeats the timing analysis, so $\widehat{F}_{\max}$ is an estimate rather than the result of a full timing-closure sweep. A fresh place-and-route at a different target could route the design differently and reach a different frequency, so $\widehat{F}_{\max}$ is the best frequency for the one implementation produced rather than the best the tool could ever achieve. A single place-and-route per design is used because repeating it for every candidate period across the suite, on both platforms and under several optimisations, would be prohibitively expensive.

The combined figure of merit is the Area-Delay Product (ADP). Since it multiplies the area by the shortest period the implementation closes, and that period is the reciprocal of the estimated $\widehat{F}_{\max}$, it is itself an estimate, written $\widehat{\mathrm{ADP}}$,

$$\widehat{\mathrm{ADP}} = A \cdot \widehat{T}_{\min}, \tag{6.1}$$

where A is the platform's area measure and $\widehat{T}_{\min}$ the shortest closed period, giving units of $\mu\mathrm{m}^2\cdot\mathrm{ns}$ on the ASIC flow and slice·ns on the FPGA flow. The two are not comparable in absolute terms and are only ever compared as relative changes against the corresponding baseline on the same platform.

Execution-time speedup is reported separately, since the $\widehat{\mathrm{ADP}}$ captures area against a single clock period rather than against the time a workload takes to run. The execution time of a bench-

mark is its clock cycle count divided by $\widehat{F}_{\max}$, so the speedup

$$\widehat{S} = \frac{C_{\text{base}}}{C_{\text{opt}}} \cdot \frac{\widehat{F}_{\max,\text{opt}}}{\widehat{F}_{\max,\text{base}}}, \tag{6.2}$$

accounts for both the change in clock cycles a specialisation produces and the change in $\widehat{F}_{\max}$ of the resulting core, where C is the clock cycle count. Since it depends on the estimated $\widehat{F}_{\max}$, the speedup $\widehat{S}$ is itself an estimate and carries the same target-clock caveat. Power, written $\widehat{P}$, is reported as total power and is estimated vectorlessly, using a default input toggle activity instead of a workload trace. As a result, the reported values mainly capture differences in the core structure, not benchmark-specific switching activity. On the FPGA flow the static power is fixed by the device, so only the dynamic power changes across optimisations, though every comparison still uses the total.

6.1.2 Benchmark Suite and Validation

The evaluation uses 21 benchmarks, the *Embench-IoT* suite [50], covering signal processing, cryptography, compression, and sorting, extended with CRYSTALS-Kyber [51] and CRYSTALS-Dilithium [52], the two National Institute of Standards and Technology (NIST)-standardised post-quantum cryptographic algorithms. The breadth of the suite is deliberate, since the goal is to show that ARVIS is a general tool and not one tuned to a single workload. Its optimisations are meant to help whichever benchmarks expose the structure each one targets, and its generality is expected to grow as new optimisations are added to the catalogue.

The target is the `rv32imc_zicsr` architecture with the `ilp32` Application Binary Interface (ABI), compiled at `-O3 -funroll-loops`, freestanding (`-nostdlib -nostartfiles -fno-builtin`) and linked with unused-section removal (`-Wl,-gc-sections`). The hardware-loop strategy and the full specialisation (Section 6.5) are the exceptions, each compiling every workload using four setups: with loop unrolling on or off and the doloop hint on or off (see Section 4.4 for a detailed description).

Validation is by self-checking simulation. The value the benchmark computes is compared against a reference value computed offline for that benchmark. A specialised variant is accepted only when its output matches the baseline's, which confirms that the specialisation preserved the program's result. A full Universal Verification Methodology (UVM) verification environment was outside the scope of this work, and the self-checking output comparison is the correctness criterion used throughout.

6.2 Hardware-Pruning Results

Pruning removes logic that the workload never exercises. It therefore leaves the cycle count of each benchmark unchanged and affects only the area, power, and maximum clock frequency of

Table 6.3: Pruning results on the ASIC flow, reported as variations from the baseline. Columns ΔCell and ΔCore give the standard cell and core area changes. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. Column $\widehat{S}$ gives the speedup obtained from the higher $\widehat{F}_{\max}$.

Benchmark	ΔCell	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	−24%	−23%	+39%	−35%	−45%	1.39×
aha-mont64	−32%	−32%	+34%	−41%	−49%	1.34×
nettle-sha256	−52%	−54%	+44%	−46%	−68%	1.44×
nettle-aes	−47%	−49%	+44%	−40%	−65%	1.44×
xgboost	−53%	−55%	+44%	−44%	−69%	1.44×
qrduino	−38%	−39%	+39%	−41%	−56%	1.39×
picojpeg	−38%	−39%	+38%	−41%	−56%	1.38×
nsichneu	−53%	−55%	+43%	−45%	−68%	1.43×
slre	−52%	−54%	+44%	−46%	−68%	1.44×
sglib-combined	−29%	−29%	+44%	−43%	−51%	1.44×
depthconv	−30%	−30%	+45%	−42%	−52%	1.45×
statemate	−52%	−54%	+44%	−47%	−68%	1.44×
edn	−39%	−40%	+44%	−41%	−58%	1.44×
huffbench	−52%	−54%	+45%	−43%	−68%	1.45×
crc32	−39%	−41%	+38%	−41%	−57%	1.38×
md5sum	−39%	−40%	+41%	−42%	−57%	1.41×
ud	−33%	−34%	+45%	−36%	−54%	1.45×
matmult-int	−30%	−29%	+44%	−41%	−51%	1.44×
kyber	−31%	−31%	+26%	−42%	−45%	1.26×
wikisort	−33%	−34%	+33%	−37%	−50%	1.33×
tarfind	−32%	−32%	+36%	−41%	−50%	1.36×
Average	−39%	−40%	+41%	−42%	−57%	1.41×*

*Geometric mean for $\widehat{S}$. All percentage averages are arithmetic means.

the core. Any speedup reported in this section comes from the change in $\widehat{F}_{\max}$ rather than from a reduction in executed cycles.

Tables 6.3 and 6.4 report the pruned core of each benchmark on the ASIC and FPGA flows. The Δ columns give variations relative to the baseline CV32E40P of Section 6.1. The $\widehat{S}$ column reports the corresponding execution-time speedup.

On the ASIC flow, pruning reduces standard cell area by 39 % on average and core area by 40 %. It also reduces power by 42 %, raises $\widehat{F}_{\max}$ by 41 %, and reduces $\widehat{\mathrm{ADP}}$ by 57 %. The FPGA flow follows the same trend. Pruning reduces slices by 36 %, LUTs by 44 %, FFs by 34 %, DSPs by 53 %, and power by 37 %. It also raises $\widehat{F}_{\max}$ by 27 % and reduces $\widehat{\mathrm{ADP}}$ by 49 %. Removing unused logic improves timing by reducing combinational logic, fanout, and routing pressure in the datapath.

Table 6.4: Pruning results on the FPGA flow, reported as variations from the baseline. Columns ΔSlice, ΔLUT, ΔFF, and ΔDSP give the changes in occupied slices, logic LUTs, FFs, and DSP blocks. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. Column $\widehat{S}$ gives the speedup obtained from the higher $\widehat{F}_{\max}$. BRAM is omitted because neither the baseline nor the pruned variants use block RAM.

Benchmark	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	−22%	−34%	−27%	−20%	+21%	−30%	−36%	1.21×
aha-mont64	−30%	−38%	−34%	−20%	+23%	−30%	−43%	1.23×
nettle-sha256	−46%	−52%	−38%	−100%	+33%	−48%	−59%	1.33×
nettle-aes	−45%	−48%	−33%	−100%	+29%	−42%	−58%	1.29×
xgboost	−48%	−54%	−38%	−100%	+33%	−50%	−61%	1.33×
qrduino	−38%	−42%	−32%	−40%	+26%	−38%	−51%	1.26×
picojpeg	−26%	−42%	−33%	−40%	+27%	−33%	−42%	1.27×
nsichneu	−32%	−47%	−38%	−100%	+42%	−39%	−52%	1.42×
slre	−47%	−52%	−38%	−100%	+35%	−45%	−60%	1.35×
sglib-combined	−32%	−37%	−32%	−20%	+22%	−24%	−44%	1.22×
depthconv	−38%	−45%	−33%	−20%	+23%	−35%	−50%	1.23×
statemate	−46%	−52%	−38%	−100%	+37%	−48%	−61%	1.37×
edn	−36%	−43%	−33%	−40%	+21%	−33%	−47%	1.21×
huffbench	−40%	−49%	−38%	−100%	+33%	−45%	−55%	1.33×
crc32	−33%	−46%	−34%	−40%	+27%	−30%	−47%	1.27×
md5sum	−27%	−39%	−33%	−40%	+27%	−35%	−42%	1.27×
ud	−36%	−38%	−29%	−40%	+20%	−39%	−47%	1.20×
matmult-int	−30%	−38%	−33%	−20%	+22%	−35%	−42%	1.22×
kyber	−33%	−39%	−32%	−20%	+22%	−29%	−45%	1.22×
wikisort	−34%	−39%	−28%	−40%	+23%	−36%	−46%	1.23×
tarfind	−36%	−42%	−33%	−20%	+23%	−36%	−48%	1.23×
Average	−36%	−44%	−34%	−53%	+27%	−37%	−49%	1.27×*

*Geometric mean for $\widehat{S}$. All percentage averages are arithmetic means.

The size of the gain depends on which parts of the original core remain necessary for each workload. Part of the reduction is shared across the benchmark set, since these bare metal executions do not use debug or interrupt handling and the corresponding logic is removed in every pruned variant. The remaining differences between individual benchmarks come from workload-specific pruning opportunities. The largest differences are caused by the arithmetic units that can be removed. The highest gains occur when the integer multiplier is removed and, in most cases, when the divider is removed as well. This group includes *nettle-sha256*, *nettle-aes*, *xgboost*, *nsich-neu*, *slre*, *statemate*, and *huffbench*. These benchmarks reach ASIC $\widehat{\text{ADP}}$ reductions between 65 % and 69 %, and FPGA $\widehat{\text{ADP}}$ reductions between 52 % and 61 %.

The remaining benchmarks keep more of the arithmetic datapath and therefore obtain smaller reductions. Benchmarks such as *qrduino*, *picojpeg*, *edn*, *crc32*, *md5sum*, *ud*, and *wikisort* still benefit clearly from pruning, but generally remain below the highest gain group. The lowest gains appear in benchmarks such as *dilithium*, *aha-mont64*, *sglib-combined*, *depthconv*, *kyber*, *matmult-int*, and *tarfind*, where most of the arithmetic datapath remains useful. The boundary between these two lower groups is not strict.

Beyond the arithmetic units, smaller differences come from workload-specific parameter reduction. Even when two benchmarks remove the same main arithmetic units, they can differ in how far the register file and program counter can be reduced. The register file is reduced from the full set of general purpose register entries required by the baseline to the subset used by the benchmark. The program counter width is also reduced from 32 bits to the number of bits needed to address the benchmark text section. These changes remove storage bits and the decoding, muxing, comparison, and routing logic connected to them. As a result, benchmarks that remove similar arithmetic units can still differ in area, power, and timing. For example, *xgboost* and *nettle-aes* both belong to the highest gain group, but their standard cell area reductions are 53 % and 47 %, respectively, both above the 39 % suite average. This difference comes from the larger register file and program counter reduction achieved for *xgboost*.

The difference between the ASIC and FPGA flows follows from how each technology maps the remaining logic. In the ASIC flow, removed logic translates directly into fewer standard cells and a smaller placed core. This gives synthesis and physical implementation more freedom to simplify the datapath. In the FPGA flow, LUTs and FFs measure the implemented logic, while slices measure the fabric sites occupied by that logic. A slice can remain counted as used even when only part of it is occupied. For this reason, the slice reduction is smaller than the LUT reduction, at 36 % compared with 44 % on average.

Power follows area on both platforms, but the reduction is smaller on the FPGA. The FPGA power reduction is 37 %, compared with 42 % on the ASIC flow. This happens because the device static power is fixed and remains beyond the reach of pruning, so only the dynamic component falls. This also helps explain why the ASIC flow reaches larger $\widehat{F}_{\text{max}}$ and $\widehat{\text{ADP}}$ gains, while the FPGA flow improves more moderately.

Table 6.5: Custom instruction clock cycle count results. Columns Baseline and Fused are the clock cycle counts before and after fusion. Column ΔCycles gives the change between them. Columns Unique and Static give the number of distinct fused instructions and the number of static instances of those instructions in each binary.

Benchmark	Baseline	Fused	ΔCycles	Unique	Static
dilithium	15 212 864	12 523 913	−17.7%	30	1613
aha-mont64	9 499 033	8 089 168	−14.8%	10	203
nettle-sha256	7 909 673	5 629 631	−28.8%	15	544
nettle-aes	7 775 417	6 595 486	−15.2%	6	226
xgboost	7 686 034	7 584 786	−1.3%	1	1
qrduino	6 126 711	5 806 742	−5.2%	23	575
picojpeg	5 906 433	5 545 127	−6.1%	14	152
nsichneu	5 712 793	5 712 793	0.0%	1	3
slre	5 562 707	5 520 713	−0.8%	6	18
sglib-combined	5 438 016	5 272 102	−3.1%	7	89
depthconv	5 359 346	5 415 072	+1.0%	2	5
statemate	4 830 228	4 830 224	−<0.1%	2	8
edn	4 604 541	3 583 994	−22.2%	8	211
huffbench	4 189 336	3 958 534	−5.5%	6	120
crc32	3 825 024	2 534 383	−33.7%	4	61
md5sum	3 595 351	3 224 231	−10.3%	10	147
ud	3 214 838	2 907 816	−9.6%	6	168
matmult-int	3 134 175	2 379 188	−24.1%	4	74
kyber	2 941 587	2 450 821	−16.7%	27	1563
wikisort	2 244 749	2 210 545	−1.5%	20	332
tarfind	1 627 084	1 446 350	−11.1%	4	20
Average			−10.8%		

6.3 Custom Instruction Results

Custom instructions fuse pairs of consecutive operations into single instructions. They can reduce the clock cycle count of a benchmark, but they also add logic to the specialised core. As a result, this section first analyses the clock cycle count results, then evaluates the hardware results of the fused and pruned variants.

6.3.1 Clock Cycle Count Results

Table 6.5 reports, for each benchmark, the baseline and fused clock cycle counts measured with Verilator, the clock cycle count change, the number of unique fused instructions, and the number of static fused instruction instances in the binary.

Custom instructions reduce the clock cycle count for 19 of the 21 benchmarks. The only exceptions are *nsichneu*, which is unchanged, and *depthconv*, which regresses by 1.0 %. Across the full suite, the arithmetic mean clock cycle reduction is 10.8 %. If only the benchmarks that improve are considered, the average rises to 12.0 %. The spread is wide. Benchmark *statemate* changes by only four clock cycles, *slre* improves by 0.8 %, and *crc32* reaches the largest reduction at 33.7 %.

The size of the reduction depends on where the fused patterns appear and how often they execute. Fusion removes the most cycles when the hot path is a short chain of dependent integer operations. It does little when the hot path is dominated by control flow, branching, memory movement, or code where the selected patterns are rarely executed.

The largest gains come from arithmetic chain kernels. Benchmark *crc32*, the strongest case at 33.7 %, benefits from a compact inner loop with repeated exclusive or, shift, and mask operations. Benchmark *nettle-sha256*, at 28.8 %, repeatedly executes the Secure Hash Algorithm (SHA)-256 round, where rotate, exclusive or, and add operations form dense dependence chains. In *matmult-int* and *edn*, the gains of 24.1 % and 22.2 %, respectively, mainly come from multiply-accumulate patterns, where a multiplication result is immediately used by an addition, so one fused instruction can replace the pair. The cryptographic benchmarks *dilithium*, *kyber*, and *aha-mont64* also benefit from multiply-accumulate and multiply-subtract forms, since their modular arithmetic contains products that feed directly into additions or subtractions. Their cycle reductions are 17.7 %, 16.7 %, and 14.8 %, respectively.

The static number of fused instructions does not reliably predict the cycle count savings. Benchmark *crc32* uses only four distinct fused instructions and 61 static instances, yet it leads the suite because those instances are in the dominant loop. By contrast, *wikisort* has 20 distinct fused instructions and 332 static instances, but improves by only 1.5 %, since most of those instances are not in its dominant compare and swap path. Benchmark *nsichneu* gives an even clearer warning. It contains one unique fused instruction and three static instances, but its measured clock cycle count is unchanged. For this reason, the existence of fused instructions in the binary is not sufficient for a cycle reduction.

The *depthconv* result shows the main risk of local fusion. It is the only benchmark whose fused binary executes more clock cycles. Comparing the baseline and fused disassemblies shows that the fused build combines a `mul` and an `add` before instruction scheduling can exploit the same freedom as in the baseline. In the baseline, independent loads can be scheduled between the `mul` and the `add`, hiding part of the load-use latency in the four-stage pipeline. After fusion, those two operations form a single instruction, which reduces the available scheduling freedom. As a result, some loads are placed closer to their consumers and introduce load-use pipeline stalls. The regression is small, but it shows that a locally valid fused instruction is not always beneficial once scheduling effects are considered.

Table 6.6: Custom instruction results on the ASIC flow. Each variant combines pruning and custom instructions. The Δ columns report variations from the baseline CV32E40P of Section 6.1. Columns ΔCell and ΔCore give the standard cell and core area changes. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. Column $\widehat{S}$ gives the speedup from Equation 6.2.

Benchmark	ΔCell	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	−2%	−3%	+30%	−24%	−25%	1.57×
aha-mont64	−22%	−23%	+37%	−41%	−44%	1.61×
nettle-sha256	−40%	−43%	+42%	−37%	−60%	2.00×
nettle-aes	−39%	−42%	+38%	−40%	−58%	1.63×
xgboost	−46%	−49%	+45%	−41%	−65%	1.46×
qrduino	−25%	−27%	+31%	−38%	−45%	1.39×
picojpeg	−25%	−27%	+26%	−35%	−43%	1.35×
nsichneu	−52%	−54%	+43%	−48%	−68%	1.43×
slre	−43%	−46%	+39%	−41%	−61%	1.40×
sglib-combined	−19%	−20%	+44%	−40%	−45%	1.49×
depthconv	−23%	−24%	+41%	−41%	−46%	1.40×
statemate	−45%	−49%	+43%	−38%	−64%	1.43×
edn	−29%	−31%	+44%	−40%	−52%	1.85×
huffbench	−43%	−46%	+43%	−40%	−62%	1.51×
crc32	−31%	−33%	+40%	−41%	−52%	2.11×
md5sum	−29%	−32%	+39%	−40%	−51%	1.54×
ud	−25%	−27%	+33%	−35%	−45%	1.47×
matmult-int	−21%	−22%	+32%	−42%	−41%	1.74×
kyber	−13%	−15%	+15%	−28%	−26%	1.38×
wikisort	−19%	−20%	+31%	−30%	−39%	1.33×
tarfind	−24%	−25%	+38%	−41%	−46%	1.56×
Average	−29%	−31%	+37%	−38%	−49%	1.54×*

*Geometric mean for $\widehat{S}$. All percentage averages are arithmetic means.

6.3.2 Fused and Pruned Core Hardware Results

Tables 6.6 and 6.7 report the hardware results for the fused variants after place and route on the ASIC and FPGA flows. These variants combine pruning and fusion. The hardware changes are therefore the net effect of removing unused baseline logic and adding the selected fused instruction support. The speedup is computed with Equation 6.2. It combines the clock cycle count ratio from Table 6.5 with the change in $\widehat{F}_{\max}$, so it represents estimated execution time rather than frequency alone.

On the ASIC flow, the pruned core extended with custom instructions reduces standard cell area by 29 % on average and core area by 31 %, while raising $\widehat{F}_{\max}$ by 37 %. Power falls by 38 % and $\widehat{\mathrm{ADP}}$ falls by 49 %, giving a geometric mean speedup of 1.54×. The FPGA flow follows the

Table 6.7: Custom instruction results on the FPGA flow. Each variant combines pruning and custom instructions. The Δ columns report variations from the baseline CV32E40P of Section 6.1. Columns ΔSlice, ΔLUT, ΔFF, and ΔDSP give the changes in occupied slices, logic LUTs, FFs, and DSP blocks. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. Column $\widehat{S}$ gives the speedup from Equation 6.2. BRAM is omitted because neither the baseline nor the fused variants use block RAM.

Benchmark	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	+14%	+18%	−22%	−20%	+4%	−3%	+9%	1.27×
aha-mont64	−21%	−23%	−32%	−20%	+14%	−30%	−31%	1.34×
nettle-sha256	−29%	−32%	−34%	−100%	+24%	−42%	−43%	1.74×
nettle-aes	−29%	−32%	−33%	−100%	+24%	−38%	−43%	1.47×
xgboost	−44%	−47%	−38%	−100%	+37%	−50%	−59%	1.39×
qrduino	−14%	−12%	−31%	−40%	+7%	−32%	−20%	1.13×
picojpeg	−15%	−16%	−32%	−40%	+17%	−30%	−28%	1.25×
nsichneu	−36%	−45%	−38%	−100%	+41%	−39%	−54%	1.41×
slre	−39%	−43%	−38%	−100%	+31%	−44%	−54%	1.32×
sglib-combined	−19%	−21%	−32%	−20%	+9%	−24%	−25%	1.12×
depthconv	−33%	−39%	−33%	−20%	+22%	−38%	−45%	1.21×
statemate	−40%	−43%	−38%	−100%	+37%	−47%	−56%	1.37×
edn	−22%	−27%	−33%	−40%	+13%	−33%	−30%	1.45×
huffbench	−25%	−36%	−37%	−100%	+28%	−39%	−42%	1.36×
crc32	−26%	−28%	−33%	−40%	+20%	−29%	−39%	1.82×
md5sum	−26%	−25%	−32%	−40%	+15%	−30%	−35%	1.28×
ud	−26%	−27%	−28%	−40%	+11%	−32%	−33%	1.22×
matmult-int	−24%	−31%	−33%	−20%	+15%	−32%	−33%	1.51×
kyber	−4%	+1%	−27%	−20%	+3%	−20%	−7%	1.24×
wikisort	+9%	−2%	−25%	−40%	+13%	−15%	−3%	1.14×
tarfind	−25%	−25%	−33%	−20%	+10%	−29%	−32%	1.23×
Average	−23%	−25%	−32%	−53%	+19%	−32%	−33%	1.34×*

*Geometric mean for $\widehat{S}$. All percentage averages are arithmetic means.

same direction. The fused and pruned variants reduce slices by 23 %, LUTs by 25 %, FFs by 32 %, DSP blocks by 53 %, and power by 32 %. They also raise $\widehat{F}_{\max}$ by 19 % and reduce $\widehat{\text{ADP}}$ by 33 %, giving a geometric mean speedup of 1.34×.

Comparing Tables 6.3 and 6.6 shows that, on the ASIC flow, the average core area reduction changes from 40 % with pruning alone to 31 % with pruning and fusion. Comparing Tables 6.4 and 6.7 shows the same trade-off on the FPGA flow, where the average LUT reduction changes from 44 % to 25 %. The fused instructions therefore spend part of the hardware margin opened by pruning in exchange for lower clock cycle counts.

Each distinct fused pattern adds decode and execution logic, so the number of unique patterns predicts hardware cost better than it predicts cycle reduction. Benchmark *dilithium* is the clearest high cost case, with 30 distinct patterns. Its ASIC standard cell reduction is only 2 %, and its FPGA LUT count increases by 18 % relative to the baseline. By contrast, *nsichneu* has only one distinct fused pattern and stays close to the pruning dominated result, with a 54 % core area reduction on ASIC and a 45 % LUT reduction on FPGA. Because the added logic is primarily combinational, the increase in LUT usage is larger than the increase in FF usage, as expected.

The timing effect of fusion is smaller than the area effect. The fused and pruned cores still have a higher $\widehat{F}_{\max}$ than the baseline on every benchmark in both flows, because pruning removes a large amount of unused logic before fusion adds any pattern support. However, the frequency gain is smaller than with pruning alone in most cases. This follows the expected trade-off. Fused instructions shorten dynamic execution when they are used on the hot path, but their single cycle execution logic can lengthen decode and execute paths. The result is a core that is faster than the baseline in estimated execution time, but not always faster than the corresponding pruned only core.

The speedup column combines the clock cycle changes from Table 6.5 with the frequency changes in Tables 6.6 and 6.7. Benchmark *crc32* is the strongest case on both flows, reaching 2.11× on ASIC and 1.82× on FPGA. Its 33.7 % clock cycle reduction combines with a positive timing gain. Benchmark *nettle-sha256* follows the same pattern, with a large cycle reduction and high speedup on both flows. Other strong cases depend on the platform. Benchmark *edn* is among the best on ASIC, while *matmult-int* is among the best on FPGA. In contrast, *qrduino* and *wikisort* remain among the weakest cases on both flows, since their fused binaries remove few clock cycles and depend mostly on the frequency gain.

Benchmark *depthconv* shows the opposite interaction between clock cycles and frequency. Its fused binary executes 1.0 % more clock cycles, but the estimated execution time still improves because $\widehat{F}_{\max}$ rises by 41 % on ASIC and 22 % on FPGA. This case should not be read as a successful cycle optimisation. It is a net speedup only because the hardware implementation remains faster than the baseline after pruning and fusion are applied.

Power and $\widehat{\text{ADP}}$ describe the hardware cost side of the trade-off. Power falls for every benchmark on both flows, from 24 % to 48 % on the ASIC flow and from 3 % to 50 % on the FPGA flow. The $\widehat{\text{ADP}}$ improves for every variant on the ASIC flow and for 20 of the 21 variants on the FPGA flow. On the ASIC flow, the largest reductions occur when a large area reduction and a

positive timing gain coincide, as in *nsichneu*, *xgboost*, *statemate*, *huffbench*, and *slre*. The smallest reductions occur in *dilithium* and *kyber*, where fusion adds back more logic or the frequency gain is smaller. The only case where $\widehat{\text{ADP}}$ worsens is *dilithium* on the FPGA flow. Its slice count grows by 14 %, enough to outweigh the timing improvement, so its $\widehat{\text{ADP}}$ rises by 9 %. This separates performance from hardware efficiency. A fused variant can shorten estimated execution time while still increasing area delay product when its patterns are costly.

Overall, adding custom instructions to the pruned core improves geometric mean speedup on both flows, from 1.41× to 1.54× on ASIC and from 1.27× to 1.34× on FPGA, using the pruning results of Section 6.2 as the comparison point. The gain is not uniform across benchmarks. Fusion is most effective when a small number of fused patterns appear repeatedly in the hot path, and it is weakest when many patterns must be implemented for little dynamic use. The final fused and pruned cores remain below the baseline hardware cost on average, but the extra logic must be repaid by enough cycle reduction to justify the area it consumes and the additional delay it introduces.

6.4 Hardware Loop Results

Hardware loops replace software loop control with hardware controlled iteration. In the applicable cases, the loop body no longer needs to execute the back edge branch and the associated fetch redirection. When the software loop control update is used only to count iterations, that update can also be removed from the instruction stream. The gain thus depends on whether the workload contains hot counted loops that satisfy the convertibility rules of Section 4.4.3.

This section develops as follows. First, it reports how the four compilation builds change the unpatched software baseline. Second, it shows how many loops each build exposes to the patcher. Third, it measures the clock cycle reduction from loop conversion and from the cross build merge of Section 4.4.7. Fourth, it explains the hardware-loop depth selected by the merit of Equation 4.2. Finally, it reports the post place and route cost of adding the loop unit on top of the pruned core of Section 6.2.

Each workload is compiled in four ways by combining two binary choices. The first choice is whether loop unrolling is enabled. The second is whether the hardware-loop hint of Section 4.4.6 is enabled. The *Unroll* build enables unrolling and disables the hint. The *NoUnroll* build disables unrolling. The two *DoLoop* builds enable the hint on top of each of those choices, giving *DoLoop+NoUnroll* and *DoLoop+Unroll*. Benchmarks that expose no eligible loop, namely *xgboost*, *nsichneu*, *depthconv*, and *aha-mont64*, are omitted from this section.

The four unpatched baselines are given in Table 6.8. These are software baselines, before any hardware-loop conversion is applied. Unrolling is the larger source of clock cycle variation. It lowers the non-unrolled baseline by 8.6 % on average and by much more when a hot loop dominates execution, such as 47 % for *matmult-int*, 33 % for *tarfind*, and 31 % for *edn*. This happens because unrolling runs fewer and longer iterations, which reduces the number of times the original software loop control is executed. For some benchmarks, however, unrolling increases

Table 6.8: Unpatched baseline clock cycle count of each benchmark in the four compilation builds, before any hardware-loop conversion. The builds are *NU* (NoUnroll), *U* (Unroll), *D+NU* (DoLoop+NoUnroll), and *D+U* (DoLoop+Unroll). Unrolling separates NU from U and D+NU from D+U, the hint separates NU from D+NU and U from D+U. Benchmarks *xgboost*, *nsichneu*, *depthconv*, and *aha-mont64* are omitted, as they expose no eligible loop.

Benchmark	NU	U	D+NU	D+U
dilithium	15 350 016	15 212 864	15 514 497	15 282 509
nettle-sha256	7 631 898	7 909 673	7 632 460	7 907 987
nettle-aes	7 546 322	7 775 417	7 589 727	7 967 008
qrduino	5 999 777	6 126 711	5 997 332	6 065 774
picojpeg	5 799 273	5 906 433	5 897 350	5 954 325
slre	5 504 936	5 562 707	5 515 495	5 566 649
sglib-combined	5 392 589	5 438 016	5 341 241	5 417 829
statemate	4 905 155	4 830 228	4 915 141	4 830 232
edn	6 646 785	4 604 541	6 723 804	4 594 066
huffbench	4 766 697	4 189 336	4 751 591	4 539 676
crc32	4 001 991	3 825 024	4 002 516	3 750 564
md5sum	4 376 591	3 595 351	4 136 091	3 584 628
ud	3 189 846	3 214 838	3 186 276	3 216 623
matmult-int	5 903 100	3 134 175	5 904 161	3 143 612
kyber	2 939 726	2 941 587	2 997 044	2 967 932
wikisort	2 556 132	2 244 749	2 578 128	2 237 661
tarfind	2 436 545	1 627 084	2 586 736	1 616 970

the clock cycle count, by up to 3.6 % for *nettle-sha256*, 3.0 % for *nettle-aes*, 2.1 % for *qrduino*, and 1.8 % for *picojpeg*. Smaller regressions also appear for *slre*, *sglib-combined*, *ud*, and *kyber*.

The hardware-loop hint, the other compilation axis, has a much smaller effect on the unpatched baseline. It mainly reshapes counted loops into a countdown form, so the average change is near zero and most benchmarks move by less than 1 %. A small increase is expected because the hint constrains the compiler to emit a loop form that the patcher can recognise. The largest cost appears in *tarfind*, where the countdown form adds up to 6.2 %. In the opposite direction, *md5sum* benefits from the reshaped code and falls by up to 5.5 %.

6.4.1 Compilation Flags and Loop Eligibility

A loop is convertible only when it satisfies the constraints of Section 4.4.3 and when its iteration count can be derived from the binary. Both conditions depend on the code shape emitted by the compiler. As a result, the four compilation builds expose different sets of loops before any conversion is applied. Patch all converts every convertible loop, so its count is the largest set that a build can convert at a given depth, and the count can grow when a deeper hardware-loop unit admits a nested loop that the shallower unit cannot take. Table 6.9 reports these counts.

Table 6.9: Number of loops converted by patch all in each build at each hardware-loop level (HW). The count includes loops that pass the structural eligibility checks of Section 4.4.3, whose iteration count is derivable, and that pass the static profitability filter. The builds are *NU* (NoUnroll), *U* (Unroll), *D+NU* (DoLoop+NoUnroll), and *D+U* (DoLoop+Unroll). Benchmarks *xgboost*, *nsich-neu*, *depthconv*, and *aha-mont64* are omitted, as they expose no eligible loop.

Benchmark	HW	NU	U	D+NU	D+U
dilithium	1	120	118	123	120
	2	126	124	129	127
nettle-sha256	1	12	12	13	13
nettle-aes	1	8	8	8	8
	2	12	12	12	12
qrduino	1	21	20	27	26
picojpeg	1	8	8	12	12
slre	1	1	1	1	1
sglib-combined	1	7	7	7	7
statemate	1	1	0	1	0
edn	1	14	14	17	16
	2	17	17	20	17
huffbench	1	23	19	30	26
crc32	1	5	2	5	2
	2	6	3	6	3
md5sum	1	15	15	15	15
ud	1	5	5	5	5
	2	7	7	7	7
matmult-int	1	12	12	12	12
	2	17	17	17	17
kyber	1	110	109	117	116
	2	116	115	123	122
wikisort	1	102	102	104	104
tarfind	1	3	3	3	3

The compiler hint either preserves or increases the convertible count. It gives the patcher a countdown loop whose iteration count does not need to be reconstructed from the assembly, so loops that the scan could not analyse become convertible. With unrolling held fixed, the hint raises *huffbench* from 23 to 30 loops, *qrduino* from 21 to 27, *picojpeg* from 8 to 12, *edn* from 14 to 17, and *kyber* from 110 to 117.

Unrolling has the opposite effect when it removes small counted loops that the patcher would otherwise convert. It lowers *huffbench* from 23 to 19 loops, *crc32* from 5 to 2, and *statemate* from 1 to 0. Other benchmarks expose the same number of loops in every build because their relevant loops already have a form that both flags leave convertible. This is the case for *nettle-aes*, *sglib-combined*, *md5sum*, *ud*, *matmult-int*, and *tarfind*.

Only seven benchmarks expose more than one convertible depth. This does not mean that the others contain no nesting. It means that the deeper candidate loops more often violate one of the conversion constraints, such as having a function call, having more than one exit, or failing the required loop shape. The unmodified CV32E40P configuration fixes the hardware-loop unit at two levels, while most workloads in this suite cannot use the second level. Without depth selection, those workloads would carry unused hardware.

6.4.2 Clock Cycle Savings and Loop Selection

The number of convertible loops is not the number of useful ones. Some conversions do not repay the setup instructions or may disturb code alignment. Within each build, the patcher therefore has two ways to obtain a converted binary. Patch all applies the static conversion set without per loop simulation, so it is cheap to run but may keep loops that increase the measured clock cycle count. The selective method simulates each candidate loop on its own and keeps only the conversions that pay. It thus never converts more loops than patch all, and it can fall back to the unpatched build when no loop is useful. Table 6.10 reports both methods against each build's own unpatched baseline.

The saving obtained from hardware-loop conversion depends strongly on the compilation build. At one hardware-loop level, selective conversion removes 9.6 % on *NoUnroll* and 10.0 % on *DoLoop+NoUnroll*, but only 1.6 % on *Unroll* and 2.2 % on *DoLoop+Unroll*. At two levels the same pattern holds, with 13.2 % and 13.3 % on the non-unrolled builds against 2.5 % and 2.6 % on the unrolled builds. The reason is that unrolling and hardware-loop conversion remove part of the same software overhead. Once unrolling has reduced the number of loop iterations, the hardware-loop unit has fewer dynamic back edges to remove.

The effect is strongest in workloads where one or a few counted loops dominate the execution time. For *matmult-int*, the selective saving is 31.3 % on *NoUnroll* but only 1.6 % on *Unroll*. For *crc32*, it is 22.9 % against 7.1 %. For *edn*, it is 23.2 % against 1.8 %. The hint changes the saving less because it mainly changes which loops are exposed, not how often the loop back edge executes. For this reason, *DoLoop+NoUnroll* and *NoUnroll* save almost the same amount at each level.

Table 6.10: Clock cycle count change of patch all (PA) and selective (Sel) against each build's own unpatched baseline, at each hardware-loop level (HW), with the number of loops the selective method drops from the patch all set (Drp). The builds are *NU* (NoUnroll), *U* (Unroll), *D+NU* (DoLoop+NoUnroll), and *D+U* (DoLoop+Unroll). Each column is measured against its own build's baseline, so the compilation effect is excluded and the figure is the loop conversion contribution on that build. The averages are taken per hardware-loop level, over the 17 benchmarks at HW=1 and the 7 with a HW=2 row. The PA and Sel entries are percentage changes. Benchmarks *xgboost*, *nsichneu*, *depthconv*, and *aha-mont64* are omitted, as they expose no eligible loop.

		NU			U			D+NU			D+U		
Benchmark	HW	PA	Sel	Drp	PA	Sel	Drp	PA	Sel	Drp	PA	Sel	Drp
dilithium	1	−2.9	−3.0	3	−0.8	−1.1	14	−4.0	−4.1	26	−1.2	−1.4	18
	2	−3.0	−3.0	8	−1.0	−1.2	4	−4.5	−4.5	1	−1.4	−1.5	24
nettle-sha256	1	−4.9	−4.9	0	−0.1	−0.1	1	−5.2	−5.2	0	+<0.1	−0.3	6
nettle-aes	1	−3.4	−4.6	1	−2.2	−2.2	0	−2.6	−2.6	1	−2.3	−2.3	2
	2	−4.8	−5.0	4	−2.4	−2.5	3	−2.8	−2.9	4	−2.5	−2.5	4
qrduino	1	−2.7	−3.4	8	−0.7	−0.7	2	−3.1	−3.1	3	+0.7	−<0.1	4
picojpeg	1	+<0.1	−0.3	2	−0.3	−0.3	0	−1.5	−1.5	2	−<0.1	−0.8	1
slre	1	+0.2	0.0	1	+0.2	0.0	1	+<0.1	0.0	1	+0.1	0.0	1
sglib-combined	1	−4.9	−4.9	0	−0.5	−0.5	2	−4.0	−5.0	1	+1.1	−<0.1	6
statemate	1	−<0.1	−<0.1	0	0.0	0.0	0	−<0.1	−<0.1	0	0.0	0.0	0
edn	1	−19.2	−23.2	1	+0.4	−1.8	1	−20.6	−23.3	4	−3.3	−3.4	3
	2	−20.9	−24.1	3	−1.4	−2.3	3	−22.0	−25.0	6	−5.0	−5.0	1
huffbench	1	−3.3	−5.6	5	+7.0	−0.3	3	−1.4	−7.9	10	−0.6	−8.4	4
crc32	1	−22.9	−22.9	0	−7.1	−7.1	1	−22.9	−22.9	0	−5.2	−5.2	1
	2	−22.9	−22.9	1	−7.1	−7.1	1	−22.9	−22.9	0	−5.2	−5.3	1
md5sum	1	−13.1	−16.7	3	−1.2	−1.9	1	−11.1	−11.1	4	−2.5	−2.7	1
ud	1	−<0.1	−<0.1	2	+<0.1	0.0	5	+<0.1	0.0	5	−<0.1	−<0.1	0
	2	−<0.1	−0.1	2	−0.1	−0.1	6	−0.1	−0.1	6	−0.1	−0.1	1
matmult-int	1	−28.1	−31.3	2	+1.0	−1.6	5	−29.2	−32.1	2	+1.0	−0.9	2
	2	−29.4	−32.6	6	−1.5	−3.1	8	−30.0	−32.4	6	−0.5	−2.3	6
kyber	1	−4.6	−4.6	16	−0.9	−1.2	8	−5.2	−5.4	15	−1.2	−1.5	27
	2	−4.6	−4.6	20	−1.0	−1.3	13	−5.2	−5.4	20	−1.2	−1.5	23
wikisort	1	−7.0	−7.5	10	−2.7	−4.3	11	−6.7	−7.0	26	−3.9	−4.0	18
tarfind	1	−30.5	−30.5	0	−5.0	−5.0	0	−39.5	−39.5	0	−5.7	−5.7	0
Average, HW=1		−8.7	−9.6		−0.8	−1.6		−9.2	−10.0		−1.4	−2.2	
Average, HW=2		−12.2	−13.2		−2.1	−2.5		−12.5	−13.3		−2.3	−2.6	

The selective step improves the result by dropping loops that are convertible but not profitable. At depth 1, it gains 1.0 percentage points over patch all on *NoUnroll*, 0.9 on *Unroll*, 0.8 on *DoLoop+NoUnroll*, and 0.8 on *DoLoop+Unroll*. The average improvement is small, but several cases show why the step is needed. On *Unroll*, patch all regresses *huffbench* by 7.0 %, while the selective step drops the harmful conversions and leaves a small saving. On *matmult-int*, patch all gains nothing on the unrolled build, while the selective step drops five loops and reaches −1.6 %. The two methods coincide when every converted loop helps, as in *tarfind*. For *slre*, the selective method keeps no loops on any build, since its only eligible loop costs clock cycles. Benchmarks *statemate* and *ud* do keep loops, but only on builds slower than their fastest baseline, so neither improves on its unpatched baseline by a useful margin. For this reason, all three are omitted from the hardware analysis that follows, where the loop unit would only add area with no clock cycle saving to repay it.

Comparing compiler unrolling with hardware-loop conversion allows two conclusions to be drawn. First, selective conversion never degrades a build because it can fall back to converting zero loops, while unrolling can increase the clock cycle count. This happens in workloads such as *nettle-sha256* and *nettle-aes*. Second, the better choice depends on the workload. In *crc32* and *dilithium*, the non-unrolled build with hardware-loop conversion reaches fewer clock cycles than the unrolled build. In *matmult-int*, *edn*, and *tarfind*, the unrolled build remains faster. This distinction matters because compiler unrolling has no direct circuit area cost, while the hardware-loop unit must repay the area, power, and timing cost it adds.

For ARVIS, the relevant value is not the best result inside one compilation build, but the lowest clock cycle count achievable after all hardware-loop stages. The merge step starts from the best single converted build and applies the genetic merge of Section 4.4.7, which can take different functions from different builds. The merged result is measured against the best of the four unpatched software baselines. This excludes the compilation effect and credits only the loop conversion and merge stages. Table 6.11 reports this result.

No single compilation reaches the lowest clock cycle count on every benchmark. The minimum is reached by *NoUnroll* for five benchmarks, *DoLoop+Unroll* for five, *DoLoop+NoUnroll* for three, and *Unroll* for one. There is a mild lean towards the non-unrolled builds, eight cases against six, because the loop unit removes the most overhead when the loop back edge still executes many times. There is also a mild lean towards the hinted builds, again eight cases against six, because the hint exposes counted loops whose iteration count the assembly scan could not derive on its own, and because the countdown form it emits needs fewer setup instructions to derive that count.

The merge of functions across the four compilations improves on the best single build for eight of the fourteen benchmarks. The largest improvements are *dilithium*, which moves from −2.6 % to −5.4 %, *sglib-combined*, from −5.0 % to −6.8 %, *kyber*, from −4.6 % to −6.3 %, *wikisort*, from −4.0 % to −5.8 %, and *qrduino*, from −3.4 % to −4.8 %. For the remaining benchmarks, the merge equals the best single build. At depth 1, it improves on the best selective result by 0.9 percentage points on average. At depth 2, the improvement rises to 1.0 percentage points because some

Table 6.11: Lowest clock cycle count reached by patch all or selective on a single build (*PA/Sel*), and by the cross build merge of Section 4.4.7, at each hardware-loop level (HW). *Build* is the compilation that reaches the *PA/Sel* minimum, one of *NU*, *U*, *D+NU*, or *D+U*. The merge composes functions across all four builds. *Swap* is the number of functions taken from a build other than the anchor, over the benchmark's total function count. The change is measured against the lowest unpatched clock cycle count over the four builds, so the compilation effect is excluded. The averages are taken per level, over the 14 benchmarks at HW=1 and the 6 with a HW=2 row. The best over depths row is the lowest each benchmark reaches across its depths. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	PA/Sel	Build	Δ	Merge	Δ	Swap
dilithium	1	14 877 274	D+NU	−2.2%	14 452 221	−5.0%	29/103
	2	14 811 574	D+NU	−2.6%	14 384 879	−5.4%	29/103
nettle-sha256	1	7 231 901	D+NU	−5.2%	7 231 901	−5.2%	0/10
nettle-aes	1	7 198 783	NU	−4.6%	7 123 728	−5.6%	3/12
	2	7 171 270	NU	−5.0%	7 098 450	−5.9%	3/12
qrduino	1	5 794 090	NU	−3.4%	5 708 900	−4.8%	3/19
picojpeg	1	5 779 448	NU	−0.3%	5 702 957	−1.7%	4/27
sglib-combined	1	5 073 394	D+NU	−5.0%	4 977 461	−6.8%	22/88
edn	1	4 438 350	D+U	−3.4%	4 438 350	−3.4%	0/12
	2	4 362 210	D+U	−5.0%	4 362 210	−5.0%	0/12
huffbench	1	4 156 327	D+U	−0.8%	4 156 327	−0.8%	0/12
crc32	1	3 086 319	NU	−17.7%	3 086 319	−17.7%	0/5
	2	3 084 973	NU	−17.7%	3 084 973	−17.7%	0/5
md5sum	1	3 486 974	D+U	−2.7%	3 486 974	−2.7%	0/9
matmult-int	1	3 083 228	U	−1.6%	3 056 228	−2.5%	1/11
	2	3 035 620	U	−3.1%	3 011 037	−3.9%	1/11
kyber	1	2 805 164	NU	−4.6%	2 754 523	−6.3%	22/75
	2	2 804 477	NU	−4.6%	2 754 523	−6.3%	22/75
wikisort	1	2 147 833	D+U	−4.0%	2 108 482	−5.8%	12/43
tarfind	1	1 524 464	D+U	−5.7%	1 524 464	−5.7%	0/7
Average, HW=1				−4.4%		−5.3%	
Average, HW=2				−6.4%		−7.4%	
Best over depths				−4.7%		−5.6%	

builds expose functions with deeper convertible loops that the merge can then use. Across the best depth of each benchmark, the suite average moves from 3.8 % for patch all to 4.7 % after selective dropping and to 5.6 % after the merge, showing that each stage contributes to the final clock cycle reduction.

The *Swap* column shows how much of each program the merge recomposes. Six of the fourteen benchmarks swap no functions, so the merged result equals the best single build. The cryptographic workloads are the opposite case. Benchmark *dilithium* takes 29 of its 103 functions from a build other than the anchor, and *kyber* takes 22 of 75. The swap count tracks the additional gain produced by the merge, not the total saving from the loop unit. Benchmark *crc32* swaps no functions but still reaches the largest saving in the suite, at 17.7 %, because one build already captures the useful loop conversion.

The number of exposed loops does not predict the achievable savings. Benchmark *crc32* converts only two to five loops, depending on the build, yet reaches a 17.7 % clock-cycle reduction against its lowest software baseline, because those loops account for a large share of the total runtime. By contrast, *wikisort* exposes around a hundred convertible loops but reaches 5.8 %, since no single converted loop dominates execution. The suite accordingly separates into three practical bands. Benchmark *huffbench* gains under 1 % because its hot paths contain no sufficiently profitable convertible counted loop. A larger group reaches roughly 5 % to 7 %, including *dilithium*, *nettle-sha256*, *nettle-aes*, *qrduino*, *sglib-combined*, *kyber*, *wikisort*, and *tarfind*. Benchmark *crc32* stands apart as the outlier. A second level adds little beyond the first for most benchmarks, with *matmult-int* and *edn* being the clearest cases where the additional level reaches one step deeper into a nest.

6.4.3 Hardware Loop Depth Selection

Clock cycle saving alone is not enough to justify the loop unit. The added hardware also costs area, can reduce $\widehat{F}_{\max}$, and can increase power. ARVIS weighs this trade-off with the merit of Equation 4.2, which selects the value of the `HW_LOOP` parameter. The clock cycle term uses the lowest achieved count at each depth from Table 6.11, measured against C_{ref}, the lowest of the four unpatched software baselines. The cost term uses the Yosys generic cell count of the core at that depth, which ARVIS obtains during the run. The merit uses area because power and final frequency are only available after synthesis and place and route.

Depth 0 is the pruned core with no hardware-loop logic. It is the normalisation point for the merit, so a merit below one means that the cycle saving is estimated to repay the added area. Table 6.12 reports the deepest level exposed by each benchmark, the selected depth, and the merit value.

The merit function keeps the unit for eight benchmarks and removes it for six. The selected set is not determined by clock cycle saving alone. The unit is kept only when the saving is large enough relative to the estimated area cost. The selected benchmarks include seven with reductions

Table 6.12: Hardware loop depth selected by the merit of Equation 4.2, normalised to the pruned core with no loop unit. The clock cycle term is the reduction below the lowest of the four software baselines, so the compilation effect is excluded. *Max* is the deepest level the benchmark exposes and *Selected* is the depth chosen, where 0 removes the unit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	Max depth	Selected depth	Merit S
dilithium	2	0	1.000
nettle-sha256	1	1	0.998
nettle-aes	2	1	0.997
qrduino	1	1	0.991
picojpeg	1	0	1.000
sglib-combined	1	1	0.991
edn	2	0	1.000
huffbench	1	0	1.000
crc32	2	1	0.936
md5sum	1	0	1.000
matmult-int	2	0	1.000
kyber	2	1	0.992
wikisort	1	1	0.993
tarfind	1	1	0.998

from 4.8 % to 6.8 %, namely *qrduino*, *nettle-sha256*, *nettle-aes*, *tarfind*, *wikisort*, *kyber*, and *sglib-combined*. Benchmark *crc32* is far stronger, with a reduction of 17.7 % and the best merit value, 0.936.

The removed cases show why the area term is needed. Benchmark *dilithium* reaches 5.4 % at depth 2 and *edn* reaches 5.0 %, but the merit still selects depth 0 because the estimated area cost is not repaid. No benchmark selects a depth beyond 1. Only six benchmarks expose a second convertible level, and for those benchmarks the second level removes at most a further 1.7 % of clock cycles over the first. This is not enough to justify the extra hardware. The contrast with the unmodified CV32E40P configuration is important. The baseline fixes the hardware-loop unit at two levels, while ARVIS gives each workload only the depth its loops repay.

6.4.4 Loop and Pruned Core Hardware Results

Tables 6.13 and 6.14 report the post place and route results for each benchmark at every hardware-loop depth. Depth 0 is the pruned core of Section 6.2. It is included so that the marginal cost of adding the loop unit can be read by comparing deeper rows with depth 0. All Δ columns remain relative to the baseline CV32E40P of Section 6.1. The speedup $\widehat{S}$ combines the lowest achieved clock cycle count with the placed $\widehat{F}_{\max}$ of that specialised core. The cycle ratio is computed as the

lowest of the four unpatched software baselines divided by the lowest achieved count after patch all, selective conversion, and merge.

Table 6.13: Hardware loop results on the ASIC flow for every hardware-loop depth (HW). Depth 0 is the pruned core of Section 6.2, included so that deeper rows can be compared with the pruned core. All Δ columns are still measured against the baseline CV32E40P of Section 6.1. Columns ΔCell and ΔCore give the standard cell and core area changes. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. The speedup $\widehat{S}$ is measured against the unmodified baseline and includes pruning, the hardware-loop cycle reduction, and the placed frequency change. It is formed as the lowest of the four unpatched software baselines divided by the lowest achieved clock cycle count, multiplied by the ratio between the placed $\widehat{F}_{\max}$ of the specialised core and the baseline $\widehat{F}_{\max}$. The shaded depth is the one selected by the merit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	ΔCell	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	0	−24%	−23%	+39%	−35%	−45%	1.39×
	1	−10%	−11%	+30%	−29%	−31%	1.37×
	2	−7%	−8%	+25%	−26%	−26%	1.32×
nettle-sha256	0	−52%	−54%	+44%	−46%	−68%	1.44×
	1	−38%	−42%	+37%	−32%	−57%	1.44×
nettle-aes	0	−47%	−49%	+44%	−40%	−65%	1.44×
	1	−35%	−38%	+37%	−28%	−55%	1.45×
	2	−32%	−35%	+32%	−32%	−51%	1.40×
qrduino	0	−38%	−39%	+39%	−41%	−56%	1.39×
	1	−26%	−28%	+31%	−29%	−45%	1.38×
picojpeg	0	−38%	−39%	+38%	−41%	−56%	1.38×
	1	−27%	−29%	+34%	−35%	−47%	1.36×
sglib-combined	0	−29%	−29%	+44%	−43%	−51%	1.44×
	1	−17%	−18%	+37%	−36%	−40%	1.47×
edn	0	−39%	−40%	+44%	−41%	−58%	1.44×
	1	−27%	−29%	+38%	−30%	−48%	1.42×
	2	−24%	−26%	+32%	−33%	−44%	1.39×
huffbench	0	−52%	−54%	+45%	−43%	−68%	1.45×
	1	−40%	−43%	+38%	−33%	−59%	1.39×

Continued on next page

Table 6.13 (continued)

Benchmark	HW	ΔCell	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
crc32	0	−39%	−41%	+38%	−41%	−57%	1.38×
	1	−27%	−29%	+31%	−35%	−46%	1.60×
	2	−25%	−27%	+29%	−34%	−43%	1.57×
md5sum	0	−39%	−40%	+41%	−42%	−57%	1.41×
	1	−27%	−29%	+34%	−29%	−47%	1.38×
matmult-int	0	−30%	−29%	+44%	−41%	−51%	1.44×
	1	−12%	−13%	+35%	−27%	−35%	1.38×
	2	−10%	−11%	+30%	−32%	−31%	1.35×
kyber	0	−31%	−31%	+26%	−42%	−45%	1.26×
	1	−12%	−13%	+19%	−30%	−27%	1.27×
	2	−10%	−11%	+13%	−21%	−21%	1.21×
wikisort	0	−33%	−34%	+33%	−37%	−50%	1.33×
	1	−21%	−23%	+27%	−25%	−40%	1.35×
tarfind	0	−32%	−32%	+36%	−41%	−50%	1.36×
	1	−20%	−21%	+28%	−32%	−38%	1.36×
Average, HW=1		−24%	−26%	+33%	−31%	−44%	1.40×‡
Average, HW=2		−18%	−20%	+27%	−30%	−36%	1.37×‡
Selected speedup							1.41×*
Max. achievable speedup							1.42×†

‡Geometric mean over the benchmarks at the given depth. *Geometric mean over the merit-selected depths. †Geometric mean over the synthesis-fastest depth of each benchmark.

Table 6.14: Hardware loop results on the FPGA flow for every hardware-loop depth (HW). Depth 0 is the pruned core of Section 6.2, included so that deeper rows can be compared with the pruned core. All Δ columns are still measured against the baseline CV32E40P of Section 6.1. Columns ΔSlice, ΔLUT, ΔFF, and ΔDSP give the changes in occupied slices, logic LUTs, FFs, and DSP blocks. Columns $\Delta\widehat{F}_{\max}$, $\Delta\widehat{P}$, and $\Delta\widehat{\mathrm{ADP}}$ give the maximum frequency, total power, and area delay product changes. The speedup $\widehat{S}$ is measured against the unmodified baseline and includes pruning, the hardware-loop cycle reduction, and the placed frequency change. It is formed as the lowest of the four unpatched software baselines divided by the lowest achieved clock cycle count, multiplied by the ratio between the placed $\widehat{F}_{\max}$ of the specialised core and the baseline $\widehat{F}_{\max}$. The shaded depth is the one selected by the merit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	0	−22%	−34%	−27%	−20%	+21%	−30%	−36%	1.21×
	1	−17%	−20%	−18%	−20%	+13%	−24%	−27%	1.19×
	2	−1%	−13%	−12%	−20%	+7%	−27%	−8%	1.13×
nettle-sha256	0	−46%	−52%	−38%	−100%	+33%	−48%	−59%	1.33×
	1	−29%	−37%	−29%	−100%	+28%	−38%	−44%	1.35×
nettle-aes	0	−45%	−48%	−33%	−100%	+29%	−42%	−58%	1.29×
	1	−19%	−31%	−26%	−100%	+21%	−32%	−32%	1.28×
	2	−23%	−26%	−23%	−100%	+14%	−33%	−33%	1.22×
qrduino	0	−38%	−42%	−32%	−40%	+26%	−38%	−51%	1.26×
	1	−25%	−28%	−27%	−40%	+21%	−33%	−38%	1.27×
picojpeg	0	−26%	−42%	−33%	−40%	+27%	−33%	−42%	1.27×
	1	−24%	−28%	−27%	−40%	+19%	−26%	−36%	1.21×
sglib-combined	0	−32%	−37%	−32%	−20%	+22%	−24%	−44%	1.22×
	1	−24%	−26%	−26%	−20%	+19%	−21%	−36%	1.28×
edn	0	−36%	−43%	−33%	−40%	+21%	−33%	−47%	1.21×
	1	−25%	−32%	−27%	−40%	+19%	−30%	−37%	1.23×
	2	−10%	−26%	−24%	−40%	+17%	−21%	−23%	1.23×
huffbench	0	−40%	−49%	−38%	−100%	+33%	−45%	−55%	1.33×
	1	−33%	−40%	−32%	−100%	+26%	−38%	−46%	1.27×
crc32	0	−33%	−46%	−34%	−40%	+27%	−30%	−47%	1.27×
	1	−28%	−33%	−27%	−40%	+23%	−27%	−41%	1.49×
	2	−22%	−26%	−24%	−40%	+20%	−30%	−35%	1.46×
md5sum	0	−27%	−39%	−33%	−40%	+27%	−35%	−42%	1.27×
	1	−25%	−28%	−27%	−40%	+18%	−30%	−37%	1.22×

Continued on next page

Table 6.14 (continued)

Benchmark	HW	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
matmult-int	0	−30%	−38%	−33%	−20%	+22%	−35%	−42%	1.22×
	1	−20%	−23%	−22%	−20%	+15%	−26%	−30%	1.18×
	2	−16%	−19%	−19%	−20%	+10%	−18%	−24%	1.15×
kyber	0	−33%	−39%	−32%	−20%	+22%	−29%	−45%	1.22×
	1	−16%	−19%	−18%	−20%	+13%	−26%	−26%	1.21×
	2	−3%	−12%	−14%	−20%	+7%	−18%	−10%	1.15×
wikisort	0	−34%	−39%	−28%	−40%	+23%	−36%	−46%	1.23×
	1	−12%	−23%	−21%	−40%	+17%	−24%	−25%	1.24×
tarfind	0	−36%	−42%	−33%	−20%	+23%	−36%	−48%	1.23×
	1	−26%	−29%	−27%	−20%	+19%	−32%	−38%	1.27×
Average, HW=1		−23%	−28%	−25%	−46%	+19%	−29%	−35%	1.26×‡
Average, HW=2		−12%	−20%	−19%	−40%	+12%	−24%	−22%	1.22×‡
Selected speedup									1.28×*
Max. achievable speedup									1.28×†

‡Geometric mean over the benchmarks at the given depth. *Geometric mean over the merit-selected depths.
†Geometric mean over the synthesis-fastest depth of each benchmark.

Adding one hardware-loop level on top of the pruned core still leaves the specialised core well below the original baseline on both flows. At HW=1, the ASIC flow averages a 26 % smaller core area, a 33 % higher $\widehat{F}_{\max}$, and 31 % lower power than the baseline. The FPGA flow averages a 23 % slice reduction, a 28 % LUT reduction, a 19 % higher $\widehat{F}_{\max}$, and 29 % lower power. At HW=2, averaged only over the six benchmarks that expose a second level, the specialised core still remains below the baseline, but the remaining area and timing margin is smaller.

The marginal cost of the loop unit is clearer when each loop variant is compared with the corresponding depth-0-pruned core. Over the eight benchmarks that activate the unit at depth 1, a single level adds a substantial amount of hardware relative to the pruned core. It increases the ASIC core area by about 20 % and the FPGA LUT count by about 27 %, which appears as about a 24 % rise in occupied slices. It also lowers $\widehat{F}_{\max}$ by about 5.1 % on the ASIC flow and 4.3 % on the FPGA flow. Power rises relative to the pruned core as well, by about 18 % on the ASIC flow and 11 % on the FPGA flow.

A second level costs less than the first because most of the unit is fixed. The aligner and decoding logic are paid once at depth 1 and shared by every level. A deeper unit mainly adds per level state, including another set of loop registers, a counter, and an end address comparator. The frequency cost does not scale in the same way because it depends on whether the added logic falls on a critical path, but it is still smaller for the second level than for the first. This supports the design choice of making the depth configurable rather than fixing the unit at the upstream two level configuration.

The cost maps differently on the two implementation flows. On the FPGA, the counters, comparators, and control logic map to general LUTs, while the loop registers map to FFs. The result is more visible in LUTs than in FFs, because the combinational control dominates the added state. On the ASIC flow, the frequency penalty is slightly larger, although the reason for this difference cannot be isolated from the available results. The timing comparison between the ASIC and FPGA flows should therefore be interpreted with caution, since $\widehat{F}_{\max}$ is obtained from a single routed implementation rather than from an exhaustive place-and-route sweep, and because the two flows rely on different place-and-route mechanisms.

The loop unit rarely gives a large marginal speedup over the pruned core. The exception is *crc32*. Its clock cycle saving is large enough to raise speedup from 1.38× to 1.60× on the ASIC flow and from 1.27× to 1.49× on the FPGA flow. For the other selected benchmarks, the removed clock cycles are often offset by the lower $\widehat{F}_{\max}$. On the ASIC flow, the clearest gains over the pruned core are *sglib-combined* and *wikisort*. On the FPGA flow, *sglib-combined*, *tarfind*, and *nettle-sha256* show modest gains. Several selected benchmarks remain almost unchanged or slightly slower than the pruned core, even though they still improve over the original baseline.

This explains the $\widehat{\text{ADP}}$ results. The hardware-loop unit improves execution time only when its clock cycle reduction is large enough to cover the frequency loss. However, $\widehat{\text{ADP}}$ does not include benchmark clock cycles. It combines area with minimum clock period. As a result, adding the loop unit usually reduces the $\widehat{\text{ADP}}$ improvement already achieved by pruning, because the unit adds area and lowers $\widehat{F}_{\max}$. Benchmark *crc32* remains the only decisive performance case, but even there the $\widehat{\text{ADP}}$ reduction is smaller than in the pruned core because the loop unit increases area and reduces $\widehat{F}_{\max}$.

The depth selection is essential. Choosing the deepest available level based only on clock cycles reaches a geometric mean speedup of 1.38× on the ASIC flow and 1.24× on the FPGA flow. The merit-selected depths reach 1.41× and 1.28×, matching the rounded geometric mean of an ideal selector that chooses the fastest placed implementation for each benchmark. The merit selects the fastest placed depth for 13 of the 14 benchmarks on the ASIC flow. The disagreement is *qrduino*, where the selected unit is only 0.6 % behind the pruned core. On the FPGA flow, the merit selects the fastest depth for 11 of the 14 benchmarks. The disagreements are *nettle-aes*, where the frequency cost of the unit is larger than expected, *kyber*, by 0.6 %, and *edn*, by 1.7 %.

The hardware-loop optimisation is thus narrow but useful. It is decisive for *crc32*, modest for a small set of benchmarks, and not worth instantiating for the rest. The important result is that ARVIS does not impose the unit uniformly. It keeps the pruned core when the estimated benefit is too small and adds only the depth that the workload is expected to repay.

6.5 Full-Specialisation Results

The full-specialisation flow applies the three RTL transformations to the same workload-specialised core. It starts from the pruning of Section 6.2, adds the custom-instruction fusion of Section 6.3, and then adds the hardware-loop unit of Section 6.4. The previous sections already evaluated

pruning alone, pruning with fusion, and pruning with hardware loops. This section thus focuses on the interaction between fusion and hardware loops, on whether the loop unit still pays off once fusion is present, and on the placed hardware cost of the full core.

The clock cycle results use the same four-build search used for hardware loops, but now each build is first compiled with the fused instruction set. The merge then takes each function body from the compilation that gives the lowest combined clock cycle count, and the loops that survive in that body are converted under the selected hardware-loop depth. The compilation-flag effects on loop eligibility and on per-build loop savings are not repeated here, since Section 6.4 already analyses them. The per-build fused baselines and the patch-all and selective savings for each build are reported in Appendix B.

The interaction analysis first considers the 17 benchmarks that expose at least one eligible loop, as in Section 6.4. The final merge, depth-selection, and hardware tables focus on the 14 benchmarks for which loop conversion remains relevant once fusion is present. Benchmarks *slre*, *statemate*, and *ud* are kept in the clock-cycle interaction table, but are omitted from the merge, depth-selection, and hardware-depth tables because the fused core without loop conversion already matches or beats their best combined result.

6.5.1 Interaction of Fusion and Hardware Loops

A first question is whether combining custom instructions with hardware loops saves as much as the two transformations do separately. The same near-additive pattern is observed across the compilation builds, so the default *Unroll* build is used as the representative case. Table 6.15 reports, relative to that build's unpatched clock cycle count, the reduction from fusion alone, the reduction from selective loop conversion alone, their arithmetic sum, and the reduction delivered by the fused core with loops at each hardware-loop depth.

Fusion and the loop unit mainly remove different instructions from the same baseline. Fusion collapses arithmetic chains, while the loop unit removes software loop-control instructions, namely the back-edge branch and, when the induction variable is used only for loop control, its self-update. These instructions are not fusion targets, since branches are not fused and a loop-control self-update has no arithmetic consumer beyond the back-edge. This explains why the combined column is close to the sum of the two columns to its left, with 13.9 % against a sum of 14.1 % on average at one level. The fusion column is fixed across loop depths because it reports the fusion-only reduction for the same build. The depth-dependent columns show what loop conversion adds on top of that fixed fused baseline.

The combined result departs from the sum by at most about two percentage points, and the departure goes in both directions. Benchmark *kyber* delivers 1.2 percentage points more than the sum predicts, with 19.1 % against 17.9 %, while *wikisort* delivers about 1.8 points less, *crc32* about 1.3 points less, and *md5sum* and *tarfind* about 0.9 points less. Every other benchmark remains within a few tenths. The main cause is the alignment of the loop-end address. The patcher forces the last instruction of a converted loop to a four-byte boundary, so when fusion changes the size

Table 6.15: Clock cycle count reduction on the default *Unroll* build, relative to that build's unpatched clock cycle count, from custom-instruction fusion alone, from selective loop conversion alone, their arithmetic sum, and the fused core with loops at each hardware-loop depth (HW). The same near-additive pattern for the other compilations is reported in Appendix B. The averages are taken per level, over the 17 benchmarks at HW=1 and the 7 benchmarks with a HW=2 row. Benchmarks *xgboost*, *nsichneu*, *depthconv*, and *aha-mont64* are omitted, as they expose no eligible loop.

Benchmark	HW	Fusion	Sel. loops	Sum	Combined
dilithium	1	−17.7%	−1.1%	−18.8%	−18.5%
	2	−17.7%	−1.2%	−18.8%	−18.6%
nettle-sha256	1	−28.8%	−0.1%	−29.0%	−28.9%
nettle-aes	1	−15.2%	−2.2%	−17.3%	−17.6%
	2	−15.2%	−2.5%	−17.6%	−17.9%
qrduino	1	−5.2%	−0.7%	−6.0%	−6.0%
picojpeg	1	−6.1%	−0.3%	−6.4%	−6.7%
slre	1	−0.8%	0.0%	−0.8%	−0.8%
sglib-combined	1	−3.1%	−0.5%	−3.5%	−3.5%
statemate	1	−<0.1%	0.0%	−<0.1%	−<0.1%
edn	1	−22.2%	−1.8%	−23.9%	−24.2%
	2	−22.2%	−2.3%	−24.5%	−24.7%
huffbench	1	−5.5%	−0.3%	−5.8%	−5.9%
crc32	1	−33.7%	−7.1%	−40.8%	−39.5%
	2	−33.7%	−7.1%	−40.9%	−40.7%
md5sum	1	−10.3%	−1.9%	−12.2%	−11.3%
ud	1	−9.6%	0.0%	−9.6%	−9.6%
	2	−9.6%	−0.1%	−9.7%	−9.7%
matmult-int	1	−24.1%	−1.6%	−25.7%	−25.8%
	2	−24.1%	−3.1%	−27.2%	−27.3%
kyber	1	−16.7%	−1.2%	−17.9%	−19.1%
	2	−16.7%	−1.3%	−18.0%	−19.2%
wikisort	1	−1.5%	−4.3%	−5.8%	−4.0%
tarfind	1	−11.1%	−5.0%	−16.1%	−15.2%
Average, HW=1		−12.4%	−1.6%	−14.1%	−13.9%
Average, HW=2		−19.9%	−2.5%	−22.4%	−22.6%

and placement of the loop body, padding can be inserted or removed relative to the unfused build. This moves the clock cycle count by a small amount.

For three benchmarks, loop conversion on top of fusion does not produce a useful reduction. Benchmark *slre* has a single eligible loop that regresses, so the selective method removes it and leaves the fused core unchanged. Benchmark *statemate* obtains only a negligible loop saving, and that saving appears in higher-clock-cycle builds. A fused build without loop conversion already reaches a lower count than any loop-converted build, so the loop unit does not need to be instantiated. A third benchmark, *ud*, shows a similar pattern, with its loops adding no clock cycle saving on top of fusion, because the selective conversion has zero effect and the tiny gain at depth 2 is within rounding tolerance. These three benchmarks are therefore kept in the interaction table, but omitted from the merge, depth-selection, and hardware-depth tables.

6.5.2 Clock Cycle Savings Across Builds

The decomposition above fixes one compilation so fusion, loops, and their sum share a single baseline. The full flow is not restricted to one build. As for hardware loops in Section 6.4, the combined flow compiles all four builds, takes each function body from the build that gives the fewest clock cycles, and converts the loops that survive in the selected body. Table 6.16 reports the lowest clock cycle count reached by the best single fused build, using either patch-all or selective, and by the cross-build merge. Both reductions are measured against the lowest fused clock cycle count over the four builds, so the fusion effect is excluded. The reported reduction is consequently the saving that loop conversion and merge add on top of fusion, which is the part that must justify the loop unit once fusion is present.

The merge beats the best single fused build on 10 of the 14 benchmarks and ties it on the rest. The gain it adds is close to that of the hardware-loop-only flow, 0.9 percentage points on average at one level against 0.8 for the loop-only merge. As in Section 6.4, the gain tends to increase with the number of functions taken from builds other than the anchor, shown in the *Swap* column. Benchmark *dilithium* gains the most, adding 3.1 points and reaching 4.2 % where its best single build reaches 1.1 %. Benchmarks *kyber*, *sglib-combined*, and *qrduino* follow with roughly 2 additional points each. When all compilations execute each function in the same number of clock cycles, or when the build with the minimum total cycle count already executes every function in fewer cycles than the alternatives, no alternative body is useful and the merge ties the best single build, as in *edn*, *huffbench*, *md5sum*, and *tarfind*.

The *Build* column also shows that fusion changes the build preference observed for hardware loops alone. The fastest single combined build is an unrolled compilation on most benchmarks, whereas the loop-only flow more often favoured no-unroll builds in Section 6.4. The reason is the relative size of the two effects. Across the 17 benchmarks of Table 6.15, fusion removes about 12 % of clock cycles on average on the default unrolled build, while the loop unit removes 1.6 %. The build choice is consequently dominated by the compilation in which fusion gives the lowest fused clock cycle count. That is usually an unrolled build, because unrolling exposes more repeated arithmetic for fusion to collapse, as shown by the appendix baseline of Table B.1.

Table 6.16: Lowest clock cycle count reached by patch-all or selective on a single fused build (*PA/Sel*), and the cross-build merge, at each hardware-loop level (HW). *Build* is the fused compilation that reaches the *PA/Sel* minimum, one of *NU*, *U*, *D+NU*, or *D+U*, while the merge composes functions across all four. *Swap* is the number of functions the merge takes from a build other than its anchor, over the benchmark's total function count. The change is against the lowest fused clock cycle count over the four builds, so the fusion effect is excluded and the figure is the saving the loop conversion and the merge add on top of fusion. The averages are taken per level, over the 14 benchmarks at HW=1 and the 6 benchmarks with a HW=2 row, and the best-over-depths row is the lowest each benchmark reaches across its depths. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	PA/Sel	Build	Δ	Merge	Δ	Swap
dilithium	1	12 391 378	U	−1.1%	12 003 289	−4.2%	32/103
	2	12 390 725	U	−1.1%	11 998 717	−4.2%	32/103
nettle-sha256	1	5 304 922	D+NU	−4.8%	5 267 842	−5.5%	1/10
nettle-aes	1	6 223 158	NU	−5.3%	6 125 948	−6.8%	3/12
	2	6 199 198	NU	−5.7%	6 090 144	−7.3%	3/12
qrduino	1	5 587 127	D+NU	−2.9%	5 474 773	−4.9%	5/19
picojpeg	1	5 512 329	U	−0.3%	5 440 825	−1.6%	4/27
sglib-combined	1	4 869 355	D+NU	−3.4%	4 770 722	−5.4%	24/88
edn	1	3 424 107	D+U	−4.2%	3 424 107	−4.2%	0/12
	2	3 357 932	D+U	−6.0%	3 357 932	−6.0%	0/12
huffbench	1	3 703 285	D+U	−6.4%	3 703 285	−6.4%	0/12
crc32	1	2 312 635	U	−8.7%	2 293 718	−9.5%	1/5
	2	2 269 515	U	−10.5%	2 250 951	−11.2%	1/5
md5sum	1	3 153 883	D+U	−1.9%	3 153 883	−1.9%	0/9
matmult-int	1	2 325 842	U	−2.2%	2 305 835	−3.1%	1/11
	2	2 278 234	U	−4.2%	2 259 637	−5.0%	1/11
kyber	1	2 340 755	NU	−4.5%	2 290 970	−6.5%	24/75
	2	2 337 993	NU	−4.6%	2 289 967	−6.6%	24/75
wikisort	1	2 146 769	D+U	−2.9%	2 105 022	−4.8%	12/43
tarfind	1	1 379 386	U	−4.6%	1 379 386	−4.6%	0/7
Average, HW=1				−3.8%		−5.0%	
Average, HW=2				−5.4%		−6.7%	
Best over depths				−4.2%		−5.4%	

Table 6.17: Loop depth selected by the merit of Equation (4.2) for the combined core, normalised to the fused core with no loop unit. The clock cycle term is the reduction the loops add on top of fusion, measured against the lowest fused clock cycle count over the four compilations, so the fusion effect is excluded. *Max* is the deepest level the benchmark exposes and *Selected* the depth chosen, where 0 removes the loop unit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	Max depth	Selected depth	Merit S
dilithium	2	0	1.000
nettle-sha256	1	1	0.997
nettle-aes	2	1	0.991
qrduino	1	1	0.991
picojpeg	1	0	1.000
sglib-combined	1	1	0.999
edn	2	0	1.000
huffbench	1	0	1.000
crc32	2	1	0.981
md5sum	1	0	1.000
matmult-int	2	0	1.000
kyber	2	1	0.991
wikisort	1	1	0.998
tarfind	1	0	1.000

This build preference explains why the combined core can fall short of the sum of the best fusion result and the best loop result. The two transformations often pull in opposite directions on the unroll flag. When they disagree, the combined flow can use only one version of a given function. It keeps the larger fusion saving, while much of the loop saving may remain in a build that fusion does not select. Benchmark *crc32* shows this effect clearly, since its loops cut 22.9 % from the no-unroll build when applied alone, but only 7.1 % from the unrolled build used in the combined result. Benchmark *nettle-aes* is the exception, because fusion and loop conversion both do best in the no-unroll build, so the combined core keeps the loop saving rather than stranding it.

6.5.3 Loop-Depth Selection

As for hardware loops alone, ARVIS must decide whether the loop unit earns its silicon once fusion is already in place. The combined core selects its loop depth with the merit of Equation (4.2), with the reference moved to account for fusion. The clock cycle term is the lowest combined clock cycle count at each depth, taken across patch-all, selective, and the merge over the four fused builds. The reference C_{ref} is the lowest fused clock cycle count without the loop unit, computed as the minimum over the four compilations. Depth 0 thus corresponds to the fused core with no loop unit, and the merit is normalised to that point. Table 6.17 reports the deepest level each benchmark exposes, the depth selected, and the merit value.

The combined merit keeps the loop unit for 7 benchmarks and removes it for 7, choosing depth 1 in every case it keeps. The kept set is *crc32*, *nettle-sha256*, *nettle-aes*, *qrduino*, *sglib-combined*, *kyber*, and *wikisort*, with *crc32* the strongest at a merit of 0.981. Compared with the loop-only selection of Section 6.4, *tarfind* no longer keeps the unit. The merits are also closer to one than in the loop-only flow, because the combined flow keeps the compilation that reaches the fewest clock cycles after applying fusion and loops together, which may not be the build where the loop unit saves most, as previously explained. As in the loop-only flow, no benchmark selects a depth beyond 1, since the second level provides too little additional benefit once the cycle reductions from the first level have been applied.

6.5.4 Combined Core Hardware Results

The clock cycle and merit results so far use the software model. The remaining question is what the costs and returns of the combined core are after place-and-route. Tables 6.18 and 6.19 report the combined core at every synthesised loop depth, with the merit-selected depth shaded. Depth 0 is a fused and pruned core with no hardware-loop unit. It is included so the marginal cost of adding the loop unit can be read by comparing deeper rows with depth 0. The hardware Δ columns remain relative to the baseline CV32E40P of Section 6.1. The speedup $\widehat{S}$ is the execution-time speedup of the whole specialised core over the unmodified baseline. At depth 0, it uses the lowest fused clock cycle count over the four compilations instead of the single fused build reported in Section 6.3.

Table 6.18: Combined results on the ASIC flow, reported as variations from the baseline CV32E40P, for every synthesised loop depth (HW). Depth 0 is a fused and pruned core with no hardware-loop unit, and deeper rows add the loop unit. The speedup $\widehat{S}$ is the speedup of the fully specialised core, with pruning, fusion, and loops included, against the unmodified baseline. It is formed as the lowest of the four software baselines divided by the lowest achieved combined clock cycle count, multiplied by the ratio between the placed $\widehat{F}_{\max}$ of the specialised core and the baseline $\widehat{F}_{\max}$. The shaded depth is the one selected by the merit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	ΔStd	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	0	−2%	−3%	+30%	−24%	−25%	1.57×
	1	+3%	+2%	+21%	−28%	−16%	1.53×
	2	+5%	+4%	+17%	−18%	−11%	1.48×
nettle-sha256	0	−40%	−43%	+42%	−37%	−60%	1.95×
	1	−34%	−37%	+33%	−31%	−53%	1.93×

Continued on next page

Table 6.18 (continued)

Benchmark	HW	ΔStd	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\text{ADP}}$	$\widehat{S}$
nettle-aes	0	−39%	−42%	+38%	−40%	−58%	1.58×
	1	−33%	−36%	+30%	−30%	−51%	1.60×
	2	−31%	−34%	+26%	−34%	−47%	1.56×
qrduino	0	−25%	−27%	+31%	−38%	−45%	1.37×
	1	−20%	−22%	+25%	−39%	−38%	1.37×
picojpeg	0	−25%	−27%	+26%	−35%	−43%	1.33×
	1	−21%	−23%	+23%	−34%	−38%	1.31×
sglib-combined	0	−19%	−20%	+44%	−40%	−45%	1.53×
	1	−14%	−15%	+32%	−26%	−36%	1.48×
edn	0	−29%	−31%	+44%	−40%	−52%	1.85×
	1	−24%	−25%	+36%	−35%	−45%	1.83×
	2	−17%	−15%	+27%	−29%	−34%	1.74×
huffbench	0	−43%	−46%	+43%	−40%	−62%	1.51×
	1	−37%	−40%	+36%	−34%	−56%	1.54×
crc32	0	−31%	−33%	+40%	−41%	−52%	2.07×
	1	−25%	−27%	+27%	−32%	−43%	2.08×
	2	−23%	−25%	+24%	−34%	−39%	2.07×
md5sum	0	−29%	−32%	+39%	−40%	−51%	1.54×
	1	−23%	−25%	+31%	−28%	−43%	1.49×
matmult-int	0	−21%	−22%	+32%	−42%	−41%	1.74×
	1	−11%	−12%	+27%	−26%	−31%	1.73×
	2	−4%	−2%	+19%	−20%	−17%	1.65×
kyber	0	−13%	−15%	+15%	−28%	−26%	1.38×
	1	−3%	−4%	+12%	−23%	−15%	1.44×
	2	+<1%	−1%	+6%	−25%	−6%	1.36×
wikisort	0	−19%	−20%	+31%	−30%	−39%	1.33×
	1	−14%	−15%	+23%	−29%	−31%	1.31×
tarfind	0	−24%	−25%	+38%	−41%	−46%	1.55×
	1	−19%	−20%	+25%	−35%	−36%	1.46×
Average, HW=1		−20%	−21%	+27%	−31%	−38%	1.56×‡
Average, HW=2		−12%	−12%	+20%	−27%	−26%	1.63×‡
Selected speedup							1.58×*
Max. achievable speedup							1.59×†

Continued on next page

Table 6.18 (continued)

Benchmark	HW	ΔStd	ΔCore	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$

[‡]Geometric mean over the benchmarks at the given depth. [*]Geometric mean over the merit-selected depths.
[†]Geometric mean over the synthesis-fastest depth of each benchmark.

Table 6.19: Combined results on the FPGA flow, reported as variations from the baseline CV32E40P, for every synthesised loop depth (HW). Depth 0 is a fused and pruned core with no hardware-loop unit, and deeper rows add the loop unit. The speedup $\widehat{S}$ is the speedup of the fully specialised core, with pruning, fusion, and loops included, against the unmodified baseline. It is formed as the lowest of the four software baselines divided by the lowest achieved combined clock cycle count, multiplied by the ratio between the placed $\widehat{F}_{\max}$ of the specialised core and the baseline $\widehat{F}_{\max}$. The shaded depth is the one selected by the merit. Beyond the four benchmarks without an eligible loop, *slre*, *statemate*, and *ud* are also omitted, since converting their loops saves no cycles.

Benchmark	HW	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	0	+14%	+18%	−22%	−20%	+4%	−3%	+9%	1.27×
	1	+17%	+18%	−12%	−20%	−3%	−9%	+21%	1.23×
	2	+21%	+25%	−12%	−20%	−10%	−9%	+34%	1.15×
nettle-sha256	0	−29%	−32%	−34%	−100%	+24%	−42%	−43%	1.70×
	1	−19%	−23%	−27%	−100%	+17%	−33%	−31%	1.70×
nettle-aes	0	−29%	−32%	−33%	−100%	+24%	−38%	−43%	1.43×
	1	−23%	−26%	−26%	−100%	+17%	−33%	−34%	1.44×
	2	−9%	−20%	−23%	−100%	+11%	−35%	−18%	1.37×
qrduino	0	−14%	−12%	−31%	−40%	+7%	−32%	−20%	1.12×
	1	+6%	−8%	−23%	−40%	+3%	−20%	+3%	1.13×
picojpeg	0	−15%	−16%	−32%	−40%	+17%	−30%	−28%	1.23×
	1	−4%	−11%	−26%	−40%	+16%	−20%	−18%	1.24×
sglib-combined	0	−19%	−21%	−32%	−20%	+9%	−24%	−25%	1.15×
	1	−9%	−17%	−25%	−20%	+14%	−15%	−20%	1.28×
edn	0	−22%	−27%	−33%	−40%	+13%	−33%	−30%	1.45×
	1	−13%	−23%	−26%	−40%	+8%	−21%	−20%	1.45×
	2	−10%	−18%	−24%	−40%	+6%	−21%	−16%	1.45×
huffbench	0	−25%	−36%	−37%	−100%	+28%	−39%	−42%	1.36×
	1	−24%	−27%	−31%	−100%	+20%	−35%	−37%	1.36×

Continued on next page

Table 6.19 (continued)

Benchmark	HW	ΔSlice	ΔLUT	ΔFF	ΔDSP	$\Delta\widehat{F}_{\max}$	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
crc32	0	−26%	−28%	−33%	−40%	+20%	−29%	−39%	1.78×
	1	−10%	−23%	−26%	−40%	+15%	−18%	−21%	1.88×
	2	−15%	−18%	−23%	−40%	+9%	−26%	−23%	1.82×
md5sum	0	−26%	−25%	−32%	−40%	+15%	−30%	−35%	1.28×
	1	−16%	−18%	−25%	−40%	+2%	−24%	−18%	1.16×
matmult-int	0	−24%	−31%	−33%	−20%	+15%	−32%	−33%	1.51×
	1	−12%	−13%	−22%	−20%	+4%	−23%	−15%	1.41×
	2	−9%	−8%	−19%	−20%	+1%	−21%	−10%	1.40×
kyber	0	−4%	+1%	−27%	−20%	+3%	−20%	−7%	1.24×
	1	+8%	+9%	−16%	−20%	−3%	−6%	+11%	1.24×
	2	+15%	+16%	−12%	−20%	−8%	−11%	+25%	1.18×
wikisort	0	+9%	−2%	−25%	−40%	+13%	−15%	−3%	1.14×
	1	+12%	+4%	−20%	−40%	+6%	−15%	+5%	1.13×
tarfind	0	−25%	−25%	−33%	−20%	+10%	−29%	−32%	1.22×
	1	−7%	−17%	−26%	−20%	+12%	−21%	−17%	1.31×
Average, HW=1		−7%	−13%	−24%	−46%	+9%	−21%	−14%	1.34×‡
Average, HW=2		−1%	−4%	−19%	−40%	+2%	−21%	−1%	1.38×‡
Selected speedup									1.35×*
Max. achievable speedup									1.36×†

‡Geometric mean over the benchmarks at the given depth. *Geometric mean over the merit-selected depths.
†Geometric mean over the synthesis-fastest depth of each benchmark.

The cost of carrying both fusion and the loop unit can be determined by comparing deeper rows with depth 0. At depth 1, the combined logic adds on average 9 % to the ASIC core area relative to the fused-and-pruned core. On the FPGA, it adds 9 % to the LUT count and 13 % to the occupied slice count relative to depth 0. It also consumes part of the timing margin created by pruning, reducing the frequency gain by about 8 % on ASIC and 5 % on FPGA. A second level adds further area and reduces the frequency gain again on the benchmarks that expose it. The combined core is consequently the largest of the four strategies. At depth 1, it still saves 21 % of ASIC core area relative to the baseline, but this is the smallest average area saving among the specialised cores. On the FPGA, the heavily fused *dilithium* and *kyber* variants even exceed the baseline resource use at depth 1, reaching 18 % LUTs and 17 % slices for *dilithium*, and 9 % LUTs and 8 % slices for *kyber*. On the ASIC flow, *kyber* remains below the baseline, while *dilithium* rises slightly above it at depth 1, with 2 % core area.

In return for these costs, the merit-selected core delivers a geometric-mean speedup of 1.58× on ASIC and 1.35× on FPGA. Measured over the same fourteen benchmarks under the same best-

of-four baseline, this matches the fused core without loop conversion on the ASIC flow, where both reach 1.58×, and edges past it on the FPGA flow, where the combined core reaches 1.35× against 1.34×. Both stay above hardware loops with pruning, at 1.41× and 1.28×, and above pruning alone, at 1.40× and 1.25×. The loop unit's extra clock cycle savings are small and are partly offset by its frequency loss. These figures take the best of the four compilations against the lowest software baseline, as the combined tables do, so they are not directly comparable with the per-section averages reported earlier, which span the full twenty-one-benchmark suite.

The selected core averages a 24 % smaller ASIC core area, a 31 % higher $\widehat{F}_{\max}$, and 34 % lower power. On the FPGA, it averages an 11 % slice reduction, a 12 % higher $\widehat{F}_{\max}$, and 24 % lower power. The positive $\widehat{F}_{\max}$ change is inherited mainly from pruning, while fusion and the loop unit consume part of that timing margin. The selected core still reduces $\widehat{\text{ADP}}$ by 42 % on ASIC and by 20 % on FPGA, but these are the smallest reductions among the four strategies. The FPGA $\widehat{\text{ADP}}$ even increases for *dilithium*, *qrduino*, *kyber*, and *wikisort* at the selected depth, because the added slice area outweighs the frequency improvement. This separates execution-time speedup from area-delay efficiency, since every selected core still runs faster than the baseline, with *crc32* the strongest at 2.08× on ASIC and 1.88× on FPGA.

The merit function is a cell-count and clock-cycle proxy, so it cannot perfectly predict the placed speedup of each depth. Using the speedups reported in Tables 6.18 and 6.19, it selects the synthesis-fastest depth for 10 of the 14 benchmarks on ASIC and for 11 of the 14 benchmarks on FPGA. The agreement is lower than for hardware loops alone, because stacking fusion and the loop unit moves the placed area and $\widehat{F}_{\max}$ further from the approximation used by the merit. Most misses are small. The selected geometric means, 1.58× on ASIC and 1.35× on FPGA, sit just below the 1.59× and 1.36× that an oracle would obtain by choosing the synthesis-fastest depth for every benchmark. The merit function therefore recovers almost all of the achievable speedup, even though a more robust model would be needed to predict each placed result exactly.

A few isolated FPGA cases break the expected monotonic ordering in which a deeper unit lowers $\widehat{F}_{\max}$, such as *sglib-combined* and *tarfind*, which place with a higher $\widehat{F}_{\max}$ on the combined depth-one core than on the lighter depth-zero core. This is an artefact of reading the frequency from one routed design instead of a full place-and-route sweep. The cases are few, the margins are small, and the expected ordering holds elsewhere.

In summary, combining fusion and the loop unit helps only some workloads. Where meaningful fusion savings and useful loop savings fall on the same workload, as in *crc32*, the combined core improves on both fusion alone and hardware loops alone. Elsewhere, the loop unit often adds area and timing costs without repaying them once fusion is present, so the selection drops it and falls back to the fused-and-pruned core. The combined core is the largest of the four strategies and consequently has the narrowest area advantage. The best transformation mix is therefore workload dependent, which the next chapter discusses.

6.6 Best Variant per Design Objective

After presenting all variants, this section identifies, for each benchmark, which specialised core a designer should actually choose. The answer depends on the optimisation objective. Three targets are considered, ADP, the total power, and the execution-time speedup. Tables 6.20 and 6.21 give, for each benchmark and each flow, the configuration that is best for each target and the value of all three metrics at that configuration, with the loop and *All* cores taken at whichever synthesised depth is best for the metric in question instead of at the merit-selected depth, since this comparison reads the measured outcome after synthesis.

Table 6.20: Best variant per metric on the ASIC flow. For each benchmark the configuration that minimises ADP, the one that minimises power, and the one that maximises speedup are given, with the three metric values at the chosen configuration as variations from the baseline. Rows are merged where one configuration is best on more than one metric. The configurations are *Pruned*, *Fused*, *Loops*, and *All*, with *Fused*, *Loops*, and *All* each built on the pruned core, adding custom instructions, the hardware-loop unit, or both. The loop and *All* cores are taken at their best synthesised depth. The speedup $\widehat{S}$ is measured against the lowest of the four software baselines, as in Sections 6.4 and 6.5, so the *Fused* figures use the best of the four compilations and may be lower than the single-build values of Section 6.3.

Benchmark	Target	Config	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	ADP, Power	Pruned	−35%	−45%	1.39×
	Speedup	Fused	−24%	−25%	1.57×
aha-mont64	ADP, Power	Pruned	−41%	−49%	1.34×
	Speedup	Fused	−41%	−44%	1.61×
nettle-sha256	ADP, Power	Pruned	−46%	−68%	1.44×
	Speedup	Fused	−37%	−60%	1.95×
nettle-aes	ADP	Pruned	−40%	−65%	1.44×
	Power	Fused	−40%	−58%	1.58×
	Speedup	All	−30%	−51%	1.60×
xgboost	ADP, Power	Pruned	−44%	−69%	1.44×
	Speedup	Fused	−41%	−65%	1.46×
qrduino	ADP, Power, Speedup	Pruned	−41%	−56%	1.39×
picojpeg	ADP, Power, Speedup	Pruned	−41%	−56%	1.38×
nsichneu	ADP	Pruned	−45%	−68%	1.43×
	Power, Speedup	Fused	−48%	−68%	1.43×
slre	ADP, Power, Speedup	Pruned	−46%	−68%	1.44×

Continued on next page

Table 6.20 (continued)

Benchmark	Target	Config	$\Delta\widehat{P}$	$\Delta\widehat{\text{ADP}}$	$\widehat{S}$
sglib-combined	ADP, Power	Pruned	−43%	−51%	1.44×
	Speedup	Fused	−40%	−45%	1.53×
depthconv	ADP, Power, Speedup	Pruned	−42%	−52%	1.45×
statemate	ADP, Power, Speedup	Pruned	−47%	−68%	1.44×
edn	ADP, Power	Pruned	−41%	−58%	1.44×
	Speedup	Fused	−40%	−52%	1.85×
huffbench	ADP, Power	Pruned	−43%	−68%	1.45×
	Speedup	All	−34%	−56%	1.54×
crc32	ADP, Power	Pruned	−41%	−57%	1.38×
	Speedup	All	−32%	−43%	2.08×
md5sum	ADP, Power	Pruned	−42%	−57%	1.41×
	Speedup	Fused	−40%	−51%	1.54×
ud	ADP, Power	Pruned	−36%	−54%	1.45×
	Speedup	Fused	−35%	−45%	1.45×
matmult-int	ADP	Pruned	−41%	−51%	1.44×
	Power, Speedup	Fused	−42%	−41%	1.74×
kyber	ADP, Power	Pruned	−42%	−45%	1.26×
	Speedup	All	−23%	−15%	1.44×
wikisort	ADP, Power	Pruned	−37%	−50%	1.33×
	Speedup	Loops	−25%	−40%	1.35×
tarfind	ADP, Power	Pruned	−41%	−50%	1.36×
	Speedup	Fused	−41%	−46%	1.55×

Table 6.21: Best variant per metric on the FPGA flow. For each benchmark the configuration that minimises ADP, the one that minimises power, and the one that maximises speedup are given, with the three metric values at the chosen configuration as variations from the baseline. Rows are merged where one configuration is best on more than one metric. The configurations are *Pruned*, *Fused*, *Loops*, and *All*, with *Fused*, *Loops*, and *All* each built on the pruned core, adding custom instructions, the hardware-loop unit, or both. The loop and *All* cores are taken at their best synthesised depth. The speedup $\widehat{S}$ is measured against the lowest of the four software baselines, as in Sections 6.4 and 6.5, so the *Fused* figures use the best of the four compilations and may be lower than the single-build values of Section 6.3.

Benchmark	Target	Config	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
dilithium	ADP, Power	Pruned	−30%	−36%	1.21×
	Speedup	Fused	−3%	+9%	1.27×
aha-mont64	ADP, Power	Pruned	−30%	−43%	1.23×
	Speedup	Fused	−30%	−31%	1.34×
nettle-sha256	ADP, Power	Pruned	−48%	−59%	1.33×
	Speedup	Fused	−42%	−43%	1.70×
nettle-aes	ADP, Power	Pruned	−42%	−58%	1.29×
	Speedup	All	−33%	−34%	1.44×
xgboost	ADP, Power	Pruned	−50%	−61%	1.33×
	Speedup	Fused	−50%	−59%	1.39×
qrduino	ADP, Power	Pruned	−38%	−51%	1.26×
	Speedup	Loops	−33%	−38%	1.27×
picojpeg	ADP, Power, Speedup	Pruned	−33%	−42%	1.27×
nsichneu	ADP	Fused	−39%	−54%	1.41×
	Power, Speedup	Pruned	−39%	−52%	1.42×
slre	ADP, Power, Speedup	Pruned	−45%	−60%	1.35×
sglib-combined	ADP, Power	Pruned	−24%	−44%	1.22×
	Speedup	All	−15%	−20%	1.28×
depthconv	ADP, Speedup	Pruned	−35%	−50%	1.23×
	Power	Fused	−38%	−45%	1.21×
statemate	ADP, Power, Speedup	Pruned	−48%	−61%	1.37×
edn	ADP, Power	Pruned	−33%	−47%	1.21×
	Speedup	All	−21%	−20%	1.45×
huffbench	ADP, Power	Pruned	−45%	−55%	1.33×
	Speedup	All	−35%	−37%	1.36×

Continued on next page

Table 6.21 (continued)

Benchmark	Target	Config	$\Delta\widehat{P}$	$\Delta\widehat{\mathrm{ADP}}$	$\widehat{S}$
crc32	ADP, Power	Pruned	−30%	−47%	1.27×
	Speedup	All	−18%	−21%	1.88×
md5sum	ADP, Power	Pruned	−35%	−42%	1.27×
	Speedup	Fused	−30%	−35%	1.28×
ud	ADP, Power	Pruned	−39%	−47%	1.20×
	Speedup	Fused	−32%	−33%	1.21×
matmult-int	ADP, Power	Pruned	−35%	−42%	1.22×
	Speedup	Fused	−32%	−33%	1.51×
kyber	ADP, Power	Pruned	−29%	−45%	1.22×
	Speedup	All	−6%	+11%	1.24×
wikisort	ADP, Power	Pruned	−36%	−46%	1.23×
	Speedup	Loops	−24%	−25%	1.24×
tarfind	ADP, Power	Pruned	−36%	−48%	1.23×
	Speedup	All	−21%	−17%	1.31×

For ADP the pruned core is the right choice for every benchmark on the ASIC flow and for all but *nsichneu* on the FPGA flow, where the fused core reaches a marginally lower ADP. For power the pruned core is best almost everywhere, the exceptions being *nettle-aes*, *matmult-int*, and *nsichneu* on the ASIC flow and *depthconv* on the FPGA flow, where the fused core draws a fraction less, from 0.1 to about 3 percentage points. The dominance of the pruned core for ADP is the expected outcome, since pruning only removes logic and so yields the smallest core. It is not, however, always the configuration with the highest clock frequency, since on a few lightly-fused workloads the fused core edges its $\widehat{F}_{\max}$ a point or two above the pruned one, and the loop core never does, as Section 6.3 sets out, but those margins are far too small to overturn the area gap. Where area or power is the binding constraint as opposed to speed, pruning alone is therefore the configuration to deploy.

No single optimisation configuration dominates the speedup results. Across the benchmarks, the best speedup is achieved by a mix of pruned, fused, hardware-loop, and fully combined cores. On the ASIC flow the fused core is fastest for eleven benchmarks, the fully specialised *All* core for four, the pruned core for five, and the loop core once. On the FPGA flow the balance shifts towards the loop unit, the fused and *All* cores tying at seven each, the pruned core taking five and the loop core two, because adding the loop unit gives up less frequency on the FPGA than on the ASIC, so its clock cycle saving more often outweighs the frequency cost and a loop-bearing core is best on more workloads.

Four benchmarks, *aha-mont64*, *xgboost*, *nsichneu*, and *depthconv*, expose no convertible loop, so only the pruned and fused cores exist for them and the contest reduces to those two. Benchmarks

slre, *statemate*, and *ud* do expose loops, but converting them saves no cycles. Therefore, the speedup comparison is effectively between the fused and pruned cores. The fused core is fastest for *ud*, and the pruned core for *slre* and *statemate*. Fusion carries the larger share of the clock cycle reduction on this suite, so on the ASIC flow, where a workload has fusible chains but no dominant convertible loop, adding the loop unit costs more frequency than it saves and the fused core is both faster and smaller than the *All* core. On the FPGA flow this reverses for *sglib-combined*, *edn*, and *tarfind*, whose best configuration becomes the *All* core rather than the fused one. The benchmarks where the pruned core is itself the fastest are those where fusion, loops, and the *All* core do not remove enough clock cycles to compensate for their frequency cost. For *qrduino* and *picojpeg* the saving is real but modest and still smaller than the frequency the extra logic gives up, and *depthconv* behaves the same way with fusion alone, so pruning is the fastest for all of them.

Taking the pruned core in place of the speedup-optimal configuration loses 8.9 % of speedup on average on the ASIC flow and 6.5 % on the FPGA flow, but the average hides a sharp split. For the benchmarks whose speedup configuration is also the pruned core, *qrduino*, *picojpeg*, *depthconv*, *slre*, and *statemate* on ASIC, and *picojpeg*, *nsichneu*, *slre*, *depthconv*, and *statemate* on FPGA, there is no loss at all, and for those where fusion or the loop unit removes only a few clock cycles, the loss is one or two per cent. The cost is concentrated in a few workloads with a heavily fused or convertible kernel, and *crc32* dominates it, where the pruned core runs at 1.38× against the 2.08× of the fully specialised core, a 34 % speed loss on ASIC and 33 % on FPGA.

Reaching the speedup-optimal core costs area and power against the pruned reference. On the ASIC flow the speedup configuration gives up 8 percentage points of $\widehat{\text{ADP}}$ and 4 of power on average relative to the pruned core. On the FPGA flow it gives up 16 percentage points of $\widehat{\text{ADP}}$, the larger figure reflecting that the added logic takes proportionally more slice area on the FPGA than core area on the ASIC. The small power differences should be read with care, since the power figures come from a vectorless estimation that does not model the real switching activity, so a gap of a percentage point or two, such as the one that makes the fused core rather than the pruned core the power optimum for *nettle-aes* and *matmult-int*, may not survive a switching-aware measurement. The two flows agree on the fastest configuration for sixteen of the twenty-one benchmarks. The five that differ are led by the loop unit paying on the FPGA but not the ASIC, which moves *qrduino*, *sglib-combined*, *edn*, and *tarfind* to a loop-bearing core there, with *nsichneu* the lone fusion-only case where fusion's frequency gain clears the pruned core on one fabric and not the other. In nearly all of these the top two configurations sit within a couple of per cent, a margin small enough that reading $\widehat{F}_{\text{max}}$ from the first routed design instead of a full place-and-route sweep can tip the choice either way, so where it is this close the smaller core is the better choice, giving lower ADP and power at a negligible speed cost.

Overall, ARVIS delivers a speedup on every benchmark across both flows, reaching 2.08× at its best, so the methodology improves performance over the whole suite. The choice among the cores is clear when ADP or power is the objective. In those cases, the pruned core dominates because it removes logic without adding new datapath cost. The best speedup, by contrast, is less uniform and must be selected per workload. The three transformations differ markedly in how

broadly they apply. Pruning is the most general, helping every one of the twenty-one benchmarks because it only removes logic. Fusion is the next most general, carried in both the fused core and the *All* core, so a fusion-bearing core is the fastest configuration for fifteen of the twenty-one benchmarks on the ASIC flow and fourteen on the FPGA, wherever a workload exposes fusible arithmetic chains. The hardware loop is the most selective, in the fastest core for only five benchmarks on the ASIC flow and nine on the FPGA, since it repays its area on just the few workloads with a hot convertible loop.

6.7 Comparison with the State of the Art

This section positions ARVIS against the works surveyed in Chapter 2. The goal is not to repeat the survey, but to compare the results of this chapter with the closest prior approaches. Since the works use different processors, technologies, workloads, and metric sets, the numbers are compared against each work's own baseline rather than treated as a technology independent ranking.

6.7.1 Subtractive Specialisation

The pruning stage of ARVIS reduces ASIC core area by 40 % and FPGA LUTs by 44 % on average. It also raises $\widehat{F}_{\max}$ by 41 % on the ASIC flow, lowers power by 42 %, and gives geometric mean speedups of 1.41× on ASIC and 1.27× on FPGA. These reductions are below the largest gate level results in the literature, but they are obtained before synthesis through an automated RTL transformation on a pipelined RISC-V core. Gate level optimisations can still be applied after the ARVIS output by the standard synthesis and place-and-route flow, so they are complementary to the ARVIS transformations.

Cherupalli et al. [16] and Si et al. [18] report larger average area savings of 62 % and 71.0 %, respectively. Their flows work after synthesis, where removable logic can be identified at gate level. ARVIS works earlier and removes named architectural and microarchitectural structures in the source RTL. This limits the maximum reduction, but keeps the transformation visible and synthesis ready.

Compared with other RTL level work, ARVIS is more aggressive and more automated. Chaidos et al. [19] report 10.6 % area and 11.4 % power reduction on a printed Zero-Riscy core, while Raisi-Ardali et al. [20] report 8 % to 43 % area reduction on single cycle flexible electronics cores. ARVIS targets a different class of processor and reports area, frequency, power, and speed together. Its pruning contribution is therefore not the largest possible reduction, but an automated RTL-level reduction that remains compatible with gate-level optimisations and later additive specialisation.

6.7.2 Additive Specialisation

Where pruning removes unused logic, custom instruction fusion and hardware loops add workload-specific mechanisms to the core. This section compares both additive stages of ARVIS against the closest prior work.

6.7.2.1 Custom-Instruction Extensions

The closest comparison for the fusion stage is the fused and pruned core of Section 6.3. It reaches geometric mean speedups of 1.54× on the ASIC flow and 1.34× on the FPGA flow. On ASIC, the final fused core still remains 31 % smaller in core area and 38 % lower in power than the baseline, because pruning offsets part of the custom instruction cost.

CIDRE [24] is the closest automated custom instruction flow. It reports estimated speedups from 1.32× to 2.47× on a custom RV64 ASIP, with an average area overhead of 11.3 %. CIDRE reaches a higher peak, but its results are model based and depend on Synopsys ASIP Designer. ARVIS reports implemented hardware costs and keeps the flow open, although its peak speedups are lower.

Manual and accelerator based approaches can go further. RISQ-V [26] reports much larger speedups on post quantum cryptography workloads, but uses twenty nine hand designed instructions and deeply integrated accelerators for one workload family, with a 1.59× cell count increase and a 43 % frequency drop. ADVICE [23] reaches higher speedups through an HLS generated accelerator, but with a large area overhead. OpenASIP [9], ARISE [21], and Seal5 [22] are useful toolchain points, but they either rely on a different design model or stop before complete post implementation cost reporting.

6.7.2.2 Hardware Loops

The hardware loop contribution is more modest. The loop and pruned core of Section 6.4 reaches geometric mean speedups of 1.41× on ASIC and 1.28× on FPGA, with much of that gain inherited from pruning. The isolated loop benefit depends on whether a workload contains hot counted loops whose setup cost is repaid.

Prior hardware loop units report larger gains on loop dominated kernels. ZOLC [27] reports 25.5 % average speedup on media and signal processing kernels such as motion estimation and matrix multiplication, falling to about 10 % on full applications. PULP reports 37 %, but for hardware loops and post increment addressing together, where post increment alone reaches up to 20 %, so the isolated loop share is smaller [28]. Armv8.1-M LOB reports up to 70 % on a single tight Cortex-M55 kernel where branch overhead dominates, with a suite level gain near 10 % [32]. The compilation baselines also differ. PULP unrolls both its baseline and its extended build, ZOLC compiles at `-O3` and inserts the loop control by assembly post processing, and LOB compares the same core with the loop feature disabled.

In ARVIS the loop gain is isolated on top of pruning and measured over full programs against a baseline that can already include loop unrolling, so only part of the suite fits the loop dominated profile of those works, with *crc32* as the clearest example. ARVIS also does not assume the loop unit is always worth keeping. It evaluates the hardware loop depth per workload and can select depth zero when the unit does not justify its area and timing cost, comparing multiple compilation builds and using measured selection to avoid loops that are convertible but not profitable.

6.7.3 Overall Positioning

No surveyed work combines RTL level pruning, custom instruction fusion, and per workload hardware loop selection in a single workload driven process. The gap addressed by ARVIS is the combination of these dimensions in one automatic flow. The full flow takes application source code and produces synthesis-ready RTL implementations, with clock cycles, area, frequency, power, and ADP reported on both ASIC and FPGA flows. Its final speedups, choosing the fastest variant per workload, reach 1.55× on ASIC and 1.36× on FPGA, and 1.59× and 1.38× over the benchmarks that gain from more than pruning.

The comparison shows that the strength of ARVIS is the breadth of its coverage across specialisation axes, not the peak result on any single one. Gate level pruning flows can remove more area, manual instruction extensions and accelerators can deliver larger speedups, and fixed hardware loop units can perform better on loop dominated kernels. Those results usually come with a narrower workload scope, manual design effort, missing implementation metrics, or a significant area or frequency cost. The ARVIS speedups are modest compared with the most specialised prior designs, but they are obtained automatically while retaining a programmable processor structure and a complete hardware cost characterisation.

6.8 Limitations

The current results of ARVIS are subject to a few limitations. The first concerns portability beyond the current target. ARVIS currently targets only the CV32E40P core. The analysis methodology is core-independent, since it operates on the compiled binary, but the RTL transformations that realise the optimisations are specific to the CV32E40P. Porting the flow to another core therefore reuses the analysis front-end, but requires a new set of RTL transformations for the target microarchitecture. Even within the CV32E40P, although the front-end identifies unused functionality automatically, the pruning phase still requires manual effort to place the pragma markers that delimit removable regions and to write the replacement handlers before removal can run without intervention.

A second limitation concerns the scope of the additive transformations, fusion and hardware loops. The fusion strategy considers only pairs of consecutive single-cycle instructions. It does not search for hot loops or critical functions where a multi-cycle custom accelerator could yield larger gains. As a result, kernels whose hot path is built around longer instruction sequences or higher-level computations cannot benefit beyond what pairwise fusion exposes.

For hardware loops, the depth is chosen by the cell-count-and-cycle merit of Section 6.4, which does not model the placed-design frequency. As the merit-versus-synthesis comparison shows, the proxy selects the synthesis-fastest depth for most benchmarks, but not for all. A more robust merit function with frequency and area estimates would track the placed result more closely and reduce the number of cases missed by the current proxy.

Even with these limitations, ARVIS shows that automatic workload-specific specialisation is practical for a RISC-V core. Once the target-specific RTL hooks are in place, the flow can start from the workload, prune unused logic, generate workload-specific fused instructions, and fit parametrisable hardware loops. The resulting synthesisable cores improve area, power, and execution time together. The limitations above thus bound the current reach of each optimisation axis rather than the viability of the approach, and they point to clear directions for extending the tool.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation set out to determine whether an automated methodology can analyse a target program and, by combining several specialisation axes, generate a specialised RISC-V processor together with the compiler support it requires. The goal was to improve area, power, and execution time without losing the operating frequency of the original core. The work answers that question affirmatively. It shows that automatically generating a workload-specialised RISC-V processor from software is feasible, and that several implementation metrics can improve at the same time.

ARVIS takes application source code as input and produces synthesis-ready RTL implementations, each optimised for a different objective, namely ADP, power, or speedup. The generated cores are produced by pruning unused hardware, adding workload-specific fused instructions, and fitting parametrisable hardware loops. These transformations are defined within a methodology that can also host further specialisation axes.

The evaluation covers twenty-one benchmarks on both an ASIC flow, using SkyWater 130 nm, and an FPGA flow, using a Spartan-7 XC7S15. The results show that the best configuration depends on the design objective, which is why ARVIS exposes one variant per objective instead of a single output. Pruning alone gives the best ADP, with average improvements of 57 % on the ASIC flow and 49 % on the FPGA flow. It also raises $\widehat{F}_{\max}$ by 41 % and lowers power by 42 % on the ASIC flow, making it the strongest option when area or power dominates. For execution time no single core is fastest across the suite, so ARVIS selects the fastest variant per workload. On most benchmarks this is a fusion-bearing core, on a few it is the fully specialised core with pruning, fusion, and hardware loops, and where the kernel exposes little to fuse or convert it is the pruned core. Choosing the fastest variant for each workload reaches a geometric-mean execution-time speedup of 1.55× on the ASIC flow and 1.36× on the FPGA flow across all twenty-one benchmarks, and 1.59× and 1.38× over the benchmarks that gain from more than pruning, rising to 2.08× on the most amenable workload. Every variant still runs above the baseline frequency, so the clock-cycle savings are not paid back through a frequency regression.

Beyond the methodology and its results, this work contributes an openly released toolchain [43]

to a research area where reproducible implementations remain scarce. To the best of current knowledge, no comparable tool exists, since no prior flow brings several forms of processor specialisation together under a single automated process. ARVIS therefore helps close this gap.

Finally, this work supports the broader case that automatic, workload-driven specialisation is a practical path toward the rapid development of dedicated processors, and one that is worth continuing to pursue.

7.2 Future Work

The limitations of Section 6.8 point to the most immediate directions for future work. The clearest is to broaden how ARVIS modifies the RTL. The analysis front-end is largely core-independent, but the transformations themselves are written for the CV32E40P and rely on source parsing and handcrafted handlers. This is also why the pruning phase still requires manual pragma markers. Replacing these parser-driven rewrites with more flexible mechanisms could reduce the manual effort and make porting to other cores cheaper.

One possible direction is to delegate part of the RTL editing process to Artificial Intelligence (AI) agents instead of relying only on fixed parsers and handwritten transformations. Recent work has shown progress in automated Verilog generation, including multi-modal models that combine natural language with visual design intent [53] and autonomous coding agents that plan over a task graph and verify their output with AST-based waveform tracing [54]. In such a flow, ARVIS would still make the specialisation decisions, deciding what to remove, fuse, or convert, while the agent would help implement the corresponding edits in the target core sources.

A second direction follows from the complementary nature of the netlist-level pruning works discussed in Section 6.7.1. ARVIS operates before synthesis and produces synthesis-ready RTL. Gate-level optimisation is therefore still applied afterwards by the normal synthesis and place-and-route flow. Chaining a dedicated netlist-level pruning pass after the ARVIS output is a natural extension, since it would target reductions at a stage that the current flow does not modify directly.

Finally, the specialisation axes implemented in this work mainly target computation, but the dominant bottleneck of many embedded workloads is memory. Extending the methodology to target the memory hierarchy and other non-datapath bottlenecks would widen the range of workloads that the flow can accelerate. The same applies to richer additive transformations, such as multi-cycle accelerators or function-level accelerators, instead of instruction-pair fusion alone. These extensions would move ARVIS from a processor specialisation flow toward a more complete system specialisation framework.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965, Reprinted in Proceedings of the IEEE, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[2] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. SC-9, no. 5, pp. 256–268, Oct. 1974. DOI: 10.1109/JSSC.1974.1050511.

[3] L. Wang and K. Skadron, "Implications of the power wall: Dim cores and reconfigurable logic," *IEEE Micro*, vol. 33, no. 5, pp. 40–48, Sep. 2013. DOI: 10.1109/MM.2013.74.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, San Jose, California, USA: ACM, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[5] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, Feb. 2019. DOI: 10.1145/3282307.

[6] RISC-V International. "The RISC-V instruction set manual, volume I: Unprivileged architecture." Version 20260120: Official Release. [Online]. Available: https://docs.riscv.org/reference/isa/v20260120/unpriv/unpriv-index.html.

[7] C. Celio, P. Dabbelt, D. Patterson, and K. Asanović, "The renewed case for the reduced instruction set computer: Avoiding ISA bloat with macro-op fusion for RISC-V," University of California, Berkeley, Tech. Rep., Jul. 2016, arXiv:1607.02318v1 [cs.AR]. [Online]. Available: https://arxiv.org/abs/1607.02318.

[8] E. Hussein, B. Waschneck, and C. Mayr, "Automating application-driven customization of ASIPs: A survey," *Journal of Systems Architecture*, vol. 148, p. 103 080, Feb. 2024. DOI: 10.1016/j.sysarc.2024.103080.

[9] K. Hepola, T. R. Arachchige, J. Multanen, and P. Jääskeläinen, "Automatically retargeting hardware and code generation for RISC-V custom instructions," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 33, no. 10, pp. 2852–2861, Oct. 2025. DOI: 10.1109/TVLSI.2025.3586902.

[10] European Electronic Components and Systems. "ECS strategic research and innovation agenda 2026, chapter 2.1: Edge computing and embedded artificial intelligence," Accessed: May 12, 2026. [Online]. Available: https://ecssria.eu/2026_2.1.

[11] S. Wallentowitz, F. K. Gürkaynak, and L. Benini. "Open letter: Open-source chips for europe," Accessed: May 12, 2026. [Online]. Available: https://open-source-chips.eu/.

[12] D. A. Patterson, "50 years of computer architecture: From the mainframe CPU to the domain-specific TPU and the open RISC-V instruction set," in *IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2018, pp. 50–53. DOI: 10.1109/ISSCC.2018.8310168.

[13] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. Cambridge, MA, USA: Morgan Kaufmann, 2017, ISBN: 978-0-12-812275-4.

[14] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, IEEE, Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[15] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, Jul. 2020. DOI: 10.1145/3361682.

[16] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke processors for applications with ultra-low area and power constraints," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, Toronto, ON, Canada: ACM, Jun. 2017, pp. 41–54. DOI: 10.1145/3079856.3080247.

[17] N. Bleier, J. Sartori, and R. Kumar, "Property-driven automatic generation of reduced-ISA hardware," in *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2021, pp. 349–354. DOI: 10.1109/DAC18074.2021.9586090.

[18] Q. Si, P. Chowdhury, R. Sreekumar, and B. Carrion Schafer, "Application specific approximate behavioral processor," *IEEE Transactions on Sustainable Computing*, vol. 8, no. 2, pp. 165–179, Apr. 2023. DOI: 10.1109/TSUSC.2022.3222117.

[19] P. Chaidos, G. Armeniakos, S. Xydis, and D. Soudris, "A bespoke design approach to low-power printed microprocessors for machine learning applications," in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025, pp. 1–5. DOI: 10.1109/ISCAS56072.2025.11044307.

[20] A. Raisiardali, K. Iordanou, J. Kufel, K. Gudimetla, K. Myny, and E. Ozer, "Flexing RISC-V instruction subset processors to extreme edge," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*, New York, NY, USA: ACM, 2025, pp. 1147–1159. DOI: 10.1145/3725843.3756036.

[21] A. Hager-Clukas, P. van Kempen, and S. Wallentowitz, *ARISE: Automating RISC-V instruction set extension*, 2025. arXiv: 2508.07725 [cs.AR].

[22] P. van Kempen, M. Salmen, D. Mueller-Gritschneder, and U. Schlichtmann, "Seal5: Semi-automated LLVM support for RISC-V ISA extensions including autovectorization," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2024, pp. 335–342. DOI: 10.1109/DSD64264.2024.00052.

[23] Q. Si and B. Carrion Schaefer, "ADVICE: Automatic design and optimization of behavioral application specific processors," in *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23)*, Knoxville, TN, USA: ACM, Jun. 2023, pp. 327–332. DOI: 10.1145/3583781.3590214.

[24] E. Rezunov, N. Zurstraßen, L. M. Reimann, and R. Leupers, "Automatic microarchitecture-aware custom instruction design for RISC-V processors," in *Proceedings of the 2025 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2025. DOI: 10.1109/ICCAD66269.2025.11240781.

[25] C. Xiao and E. Casseau, "Exact custom instruction enumeration for extensible processors," *INTEGRATION, the VLSI Journal*, vol. 45, no. 3, pp. 263–270, Jun. 2012. DOI: 10.1016/j.vlsi.2011.11.011.

[26] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 239–280, 2020. DOI: 10.13154/tches.v2020.i4.239-280.

[27] N. Kavvadias and S. Nikolaidis, "Elimination of overhead operations in complex loop structures for embedded microprocessors," *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 200–214, Feb. 2008. DOI: 10.1109/TC.2007.70790.

[28] M. Gautschi et al., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017. DOI: 10.1109/TVLSI.2017.2654506.

[29] OpenHW Group. "CV32E40P user manual." Version v1.8.3, 15 July 2024, Accessed: May 13, 2026. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/cv32e40p_v1.8.3/.

[30] L. Codrescu et al., "Hexagon DSP: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, Mar. 2014. DOI: 10.1109/MM.2014.12.

[31] Arm Limited, *Armv8-M architecture reference manual*, Arm DDI 0553B.j, Dec. 2019. [Online]. Available: https://developer.arm.com/docs/ddi0553/bj.

[32] Arm Limited, *Lighter yet better: Powering edge through Armv8.1-M features*, White Paper, Version 1.0, Non-Confidential, Arm document 107564_0100_01_en, Issue 01, 2021. Accessed: May 13, 2026. [Online]. Available: https://developer.arm.com/documentation/107564/0100/Low-Overhead-Branch--LOB--performance-analysis/.

[33] lowRISC. "Ibex RISC-V core documentation," Accessed: May 13, 2026. [Online]. Available: https://ibex-core.readthedocs.io/en/latest/.

[34] OpenHW Group. "CVA6 RISC-V core documentation," Accessed: May 17, 2026. [Online]. Available: https://cva6.readthedocs.io/en/latest/.

[35] Free Software Foundation. "GNU binutils documentation." Version 2.46, Accessed: May 17, 2026. [Online]. Available: https://sourceware.org/binutils/docs/binutils/.

[36] M. Popoloski. "slang: SystemVerilog compilation, elaboration, analysis, tooling," Accessed: May 17, 2026. [Online]. Available: https://sv-lang.com/.

[37] Free Software Foundation. "GCC, the GNU compiler collection." Version 15.2.0, Accessed: May 18, 2026. [Online]. Available: https://gcc.gnu.org/.

[38] Free Software Foundation. "Machine descriptions." Chapter in the GNU Compiler Collection (GCC) Internals manual, version 15.2.0, Accessed: May 18, 2026. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html.

[39] Chipyard. "The RISC-V ISA Simulator (Spike)," Accessed: May 18, 2026. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Software/Spike.html.

[40] W. Snyder, "Verilator: Open simulation growing up," in *DVClub Bristol*, 2013. [Online]. Available: https://www.veripool.org/papers/Verilator_Open_Simulation_DVClub13_pres.pdf.

[41] YosysHQ. "Yosys open SYnthesis Suite." Version 0.63, Accessed: May 25, 2026. [Online]. Available: https://yosyshq.net/yosys/.

[42] A. Cockburn. "Hexagonal architecture," Accessed: May 25, 2026. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/.

[43] P. Fidalgo. "ARVIS: Automated RISC-V intelligent specialisation." Source code repository, revision d59673e, Accessed: Jun. 1, 2026. [Online]. Available: https://github.com/PauloFidalgo/arvis.

[44] FOSSi Foundation. "LibreLane: An open-source digital ASIC implementation flow," Accessed: Jun. 1, 2026. [Online]. Available: https://librelane.org/.

[45] SkyWater PDK Authors. "SkyWater open source PDK (Sky130)," Accessed: Jun. 2, 2026. [Online]. Available: https://skywater-pdk.readthedocs.io/.

[46] AMD, *Spartan 7 FPGAs data sheet: DC and AC switching characteristics (DS189)*, Document ID: DS189; Revision 1.11, Oct. 18, 2024. [Online]. Available: https://docs.amd.com/r/en-US/ds189-spartan-7-data-sheet.

[47] AMD. "Vivado design suite user guide: Release notes, installation, and licensing (UG973)." Document ID: UG973; Version 2025.2 English, Accessed: Jun. 2, 2026. [Online]. Available: https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license.

[48] Z. Snow. "sv2v: SystemVerilog to Verilog." Version v0.0.13, Accessed: Jun. 4, 2026. [Online]. Available: https://github.com/zachjs/sv2v.

[49] Docker, Inc. "Docker." Docker Engine 29.4.3, Accessed: Jun. 4, 2026. [Online]. Available: https://docs.docker.com/.

[50] D. Patterson et al., "Embench IOT 2.0 and DSP 1.0: Modern embedded computing benchmarks," *Computer*, vol. 58, no. 5, pp. 37–47, 2025. DOI: 10.1109/MC.2024.3511352.

[51] J. Bos et al., "CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[52] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, *CRYSTALS – dilithium: Digital signatures from module lattices*, Cryptology ePrint Archive, Paper 2017/633, 2017. [Online]. Available: https://eprint.iacr.org/2017/633.

[53] K. Chang et al., "Natural language is not enough: Benchmarking multi-modal generative AI for verilog generation," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24, Newark Liberty International Airport Marriott, New York, NY, USA: Association for Computing Machinery, 2025, ISBN: 9798400710773. DOI: 10.1145/3676536.3676679. [Online]. Available: https://doi.org/10.1145/3676536.3676679.

[54] C.-T. Ho, H. Ren, and B. Khailany, "VerilogCoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (AST)-based waveform tracing tool," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'25/IAAI'25/EAAI'25, AAAI Press, 2025, ISBN: 978-1-57735-897-8. DOI: 10.1609/aaai.v39i1.32007. [Online]. Available: https://doi.org/10.1609/aaai.v39i1.32007.

Appendix A

Hyperparameter Tuning

Alongside the three RTL transformations of the main text, this work implements and evaluates a hyperparameter-tuning mechanism, a search over a core's configurable size parameters that selects, per workload, the value preferred under a chosen merit. On CV32E40P the only such parameter is the instruction-prefetch First-In First-Out (FIFO) depth. The full infrastructure for the search is in place, but its measured benefit proved marginal, the shipped default depth of two being preferred for almost every benchmark and the few that depart gaining around one per cent in clock cycles. The study is therefore reported here rather than in the main evaluation, where it would carry weight out of proportion to its effect. The main-text results are all reported at the default depth of two. This appendix gives the methodology of the search, its application to the prefetch buffer, and the per-benchmark merit values.

A.1 Methodology

A core may expose parameters whose value sets the size of a resource without changing its architectural behaviour, such as the depth of a prefetch buffer or the capacity of a cache. Since this tuning acts on a size rather than on the instruction set or the datapath, it applies to any core configuration, including one that already carries the pruning, fusion, and hardware-loop modifications, and is therefore run as the last step of the flow, on the final specialised RTL rather than on the stock core. Sizing the resource on the already-pruned, fused, and loop-converted core is what makes the choice meaningful, since the clock cycle behaviour the depth must suit is the behaviour of the core that will actually be built, after the other transformations have changed its stall pattern. These differ from the resizing parameters of pruning, whose correct value is fixed by functional sufficiency. A wider buffer or a larger cache does not change what the program computes, but it can change how many clock cycles the program takes, and the preferred value is workload-sensitive because it depends on the program's dynamic behaviour and on the metric used to compare configurations. A tunable parameter therefore admits different values, and those values have impact in the performance they yield and the area they cost. The task is therefore not to derive the only correct value from the binary, but to select the value preferred under the chosen metric for the workload at hand.

That choice can be made analytically or empirically. The analytical approach studies the role of each parameter in the microarchitecture, models its effect on a target metric, and predicts a suitable configuration from that model. The empirical approach instead treats the core as a black box, evaluates candidate configurations, measures each against a target metric, and selects the lowest-scoring result. Analytical methods are efficient when accurate models are available, but many parameter interactions are difficult to characterise precisely. The empirical approach captures those interactions directly and scales to multi-parameter searches, at the cost of evaluating each candidate individually. This work adopts the empirical approach.

The problem this defines is one of discrete optimisation. Let each tunable parameter i admit a finite set of candidate values V_i, so that a configuration is a point

$$\theta \in \Theta = \prod_i V_i \tag{A.1}$$

in the Cartesian product of those sets. A merit function $S(\theta)$ scores each configuration, and the goal is the configuration that minimises it,

$$\theta^\star = \arg\min_{\theta \in \Theta} S(\theta). \tag{A.2}$$

The evaluation loop associated with this optimisation problem is shown in Figure A.1. Each candidate configuration is applied to the RTL, evaluated through the downstream flow, and scored before the search either proposes another candidate or returns the selected configuration.

The two design decisions this framing isolates are the choice of merit function, which fixes what a good configuration means, and the choice of search strategy, which fixes how Θ is explored.

A merit function maps the metrics produced by a configuration to the single scalar that Equation A.2 minimises. How it is defined and what it takes as inputs depend on the objective of the specialisation. When the goal is to reduce execution time alone, the merit draws only on the metrics that bear on speed and leaves area out, whereas when the goal is to improve area and performance together, a composite that combines both is the appropriate choice. The form of any such composite expresses the trade-off sought between its axes.

The search strategy decides which configurations of Θ are evaluated and in what order, then returns the one of lowest merit. The space has a size which is the product $\prod_i |V_i|$ of the per-parameter candidate counts and so grows multiplicatively with each parameter added, so the appropriate strategy depends on a configurable budget that defines the number of configurations the flow is willing to evaluate in full. When the size is at or below this budget, every configuration can be evaluated and the global minimum over Θ is found directly by exhaustive search, at a cost of $\mathcal{O}(\prod_i |V_i|)$ evaluations. When the size exceeds that budget, the role of the strategy changes, since it must spend a limited evaluation budget on the configurations most likely to improve on the best result found so far. Any adaptive search that proposes configurations from the results of earlier evaluations is admissible, and the one best suited depends on the structure of the space.

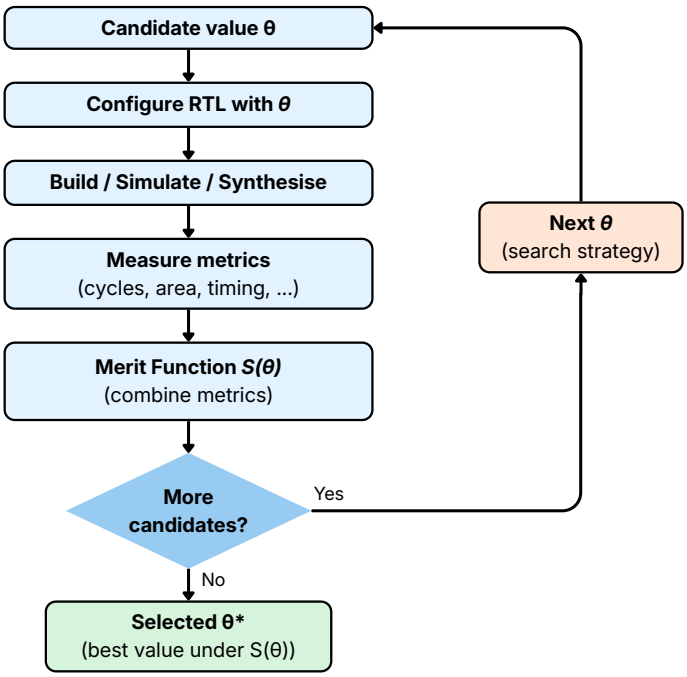


Figure A.1: Hyperparameter-tuning evaluation loop. A candidate configuration θ is applied to the RTL, built, simulated, and synthesised. The resulting metrics are combined by the score function $S(\theta)$. The search strategy either proposes another candidate or returns the selected configuration $\theta^{\star}$.

A.2 Application to the Prefetch Buffer

The hyperparameter methodology treats CV32E40P as a parameterised design over which a search is performed. Before the search, the core's parameters are audited to determine which of them form a performance-versus-area trade-off space, since only those are hyperparameters in the sense intended.

CV32E40P exposes different parameters, but most are not tuning variables in this work. Parameters such as `FPU`, `COREV_PULP`, `NUM_MHPMCOUNTERS`, `PC_WIDTH`, and the introduced `ENABLE_*` are handled by pruning because the workload fixes a functionally sufficient value. They either decide whether a unit is needed or how large a structure must be for the workload to execute correctly. Values below that requirement are invalid, while larger values mainly preserve unnecessary hardware. They therefore define the pruning configuration C of Section 4.2, not a performance search space.

The `FIFO_DEPTH` parameter is different. All supported values leave the core functionally valid, but they change how far the fetch front-end can run ahead of the pipeline and how much storage is added. This can affect both clock cycle count and area, so `FIFO_DEPTH` is the parameter exposed to the tuning search.

The target core uses a prefetch buffer between the instruction memory interface and the fetch stage. Its role is to keep the fetch front-end supplied while the downstream pipeline stalls, by running ahead of the program counter and holding the fetched instructions until the fetch stage

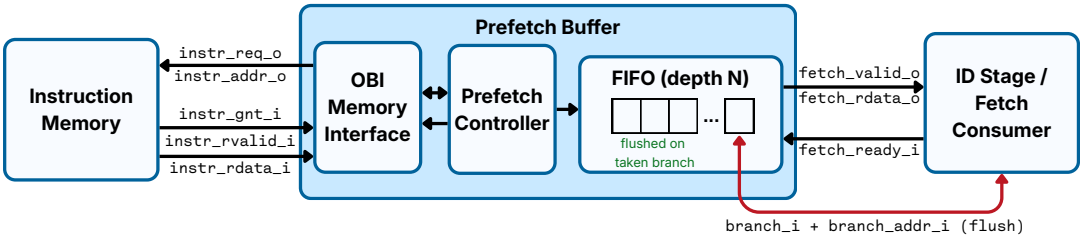


Figure A.2: Prefetch buffer architecture in CV32E40P. The buffer sits between the instruction memory interface and the fetch stage. The prefetch controller issues addresses ahead of the program counter through the OBI-compatible memory interface, and responses are buffered in the FIFO until the fetch stage consumes them. The depth N, set by the `FIFO_DEPTH` parameter, bounds the total number of in-flight requests plus buffered words. A taken branch from the decode stage flushes both the FIFO and the outstanding requests.

consumes them. It can also hold multiple outstanding memory requests, which would absorb a multi-cycle memory latency. The buffer is implemented as a FIFO whose depth, set by the `FIFO_DEPTH` parameter, bounds the total of in-flight instruction-memory requests and words already buffered. The depth is configurable, with the upstream default set to 2, which the designers also specify as the smallest permitted value. Figure A.2 shows the buffer's internal structure and its handshakes with the surrounding logic.

The handshakes shown describe the buffer's steady-state operation. The prefetch controller asserts a memory request on `instr_req_o` together with the next sequential address on signal `instr_addr_o` whenever the fetch path wants instructions and the total of words already buffered plus responses in flight is below the depth. The memory accepts the request on the same clock cycle that it raises `instr_gnt_i`, after which the controller can issue a new address on the following clock cycle. The response arrives later on `instr_rvalid_i` with the data on `instr_rdata_i`, and the FIFO captures it on the same clock cycle. On the consumer side, the fetch stage pulls one word from the FIFO whenever `fetch_ready_i` is high, with `fetch_valid_o` signalling that `fetch_rdata_o` carries a valid word.

The depth can affect clock cycle count because it bounds how much memory work the buffer can run in parallel with the downstream pipeline. The buffer banks work during any stall that backpressures fetch without redirecting the program counter, since while the execute stage is stalled the decode stage cannot accept a new instruction and `fetch_ready_i` to the prefetch buffer goes low, so the FIFO stops popping while the program counter keeps advancing sequentially. On CV32E40P these stalls come from the instructions that occupy the execute stage for more than one clock cycle, the loads and stores that wait for the memory response after issuing the address, the MULH state machine, and the iterative divider, together with the load-use data hazard, where an instruction in decode reads a register a load in execute has not yet produced and is held until the value is ready. Ordinary data dependencies between arithmetic instructions do not stall, since the core forwards a result from execute to a following instruction, so it is the multi-cycle operations and the load-use case that create the stalls the buffer exploits. The prefetch controller continues to assert `instr_req_o` as long as the fetch path still wants instructions and there is room in the

buffer, so requests issued during the stall accumulate as outstanding transactions and, when their responses arrive on `instr_rvalid_i`, fill the FIFO up to the depth.

A deeper FIFO therefore acts as a slack reservoir. More memory requests can be in flight during such a stall, and more responses land in the FIFO before the stall resolves. When the stalling instruction completes, `fetch_ready_i` returns high and the fetch stage streams the buffered words to the decode stage at one per clock cycle until the buffer drains, without waiting for fresh memory responses. A shallower FIFO saturates quickly, performs less work during the stall, and forces the front end to wait for new responses once the stall ends. Branches do not benefit from depth in the same way. When the pipeline redirects to a new target on a taken branch, the controller flushes the FIFO and discards the outstanding responses by tracking how many `instr_rvalid_i` pulses must be ignored. As a consequence, depth influences the size of the work that gets discarded on a branch. The clock cycle savings come from the stall-overlap mechanism, and the optimum depth depends on the workload's density of stalling instructions, the typical instruction-memory latency, and the relative weight that the chosen merit function places on clock cycles against area.

The space has a single parameter, so a configuration θ is one depth value. The upstream designers specify a minimum depth of 2, so the space $\Theta = \{2,3,4,5\}$ is the set of valid buffer depths the search explores. The evaluation budget below which the flow enumerates exhaustively is a configurable parameter of ARVIS, left at its default of 10 configurations in this work. With $|\Theta| = 4$ the prefetch space falls under it, so the exhaustive search enumerates it in full.

The prefetch depth trades area against clock cycles, so the merit function combines both axes by the composite of Equation 4.2, the geometric mean of the normalised cell-count area and clock cycle count, with the reference taken at the default depth of two. The area $A(\theta)$ is the Yosys [41] cell count and the clock cycle count $C(\theta)$ comes from simulating each candidate with Verilator [40]. Both are deliberately lightweight, since the cell count does not replace post-place-and-route area and the clock cycle count does not capture any change in maximum frequency, and applying the full implementation flow to every candidate would make the sweep impractical.

A.3 Per-Benchmark Merit Values

Figure A.3 plots the composite merit at each prefetch-FIFO depth for every benchmark, normalised so depth 2 is 1, with each curve labelled by the depth the search selects. Because pruning leaves both the clock cycle count and the per-depth area change untouched, the same values apply to the unmodified core and to the pruned core. Depth 2 is selected for sixteen of the benchmarks, which confirms it as the right general-purpose default and the reason the upstream core ships with it. The five that depart gain little, *dilithium* and *nettle-sha256* settling at depth 4 with clock cycle reductions of 3.4 % and 1.0 % over depth 2, and *kyber*, *crc32*, and *nettle-aes* at depth 3 with reductions of 2.1 %, 1.3 %, and 0.8 %. Almost every curve rises monotonically with depth, since the added storage costs area while the deeper buffer recovers no clock cycles on a workload whose stalls the default depth already covers, and only the five workloads with a denser pattern of multi-cycle stalls bend below 1 at all, and then by around one per cent. The flatness of the sweep is the

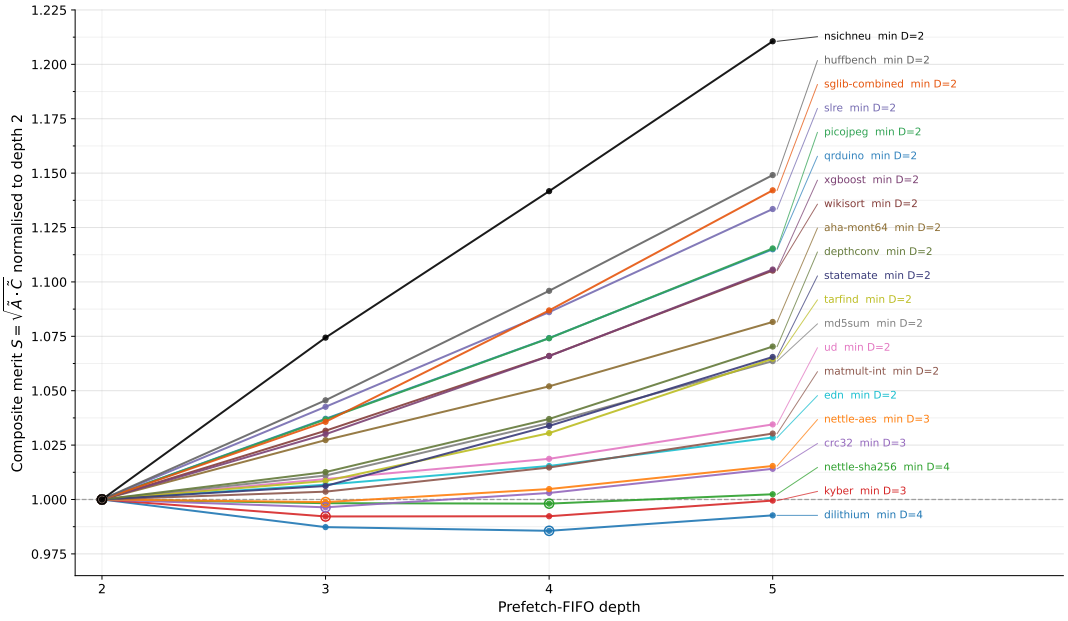


Figure A.3: Composite merit $S = \sqrt{\tilde{A} \cdot \tilde{C}}$ against prefetch-FIFO depth for every benchmark on the baseline core, normalised so depth 2 is 1. Each curve is labelled with the depth the search selects for that benchmark. Most curves rise with depth, so the default depth of two is selected, and the few that dip below 1 do so by around one per cent.

result, since the search infrastructure is general and would select a workload-specific depth where one paid, but on this core the knob is weak and the marginal gain on a single narrow axis is why the study is reported here rather than in the main evaluation.

In summary, the prefetch depth is a genuine tuning axis that ARVIS exposes and searches automatically through the same build, simulate, and synthesise loop used for the other strategies, but on CV32E40P its leverage is small. Keeping the shipped default of two is already the merit-optimal choice for sixteen of the twenty-one benchmarks, and the five that prefer a deeper buffer recover at most a few per cent of clock cycles at a measurable area cost, so the net effect on the specialised core is negligible. The depth is therefore fixed at two throughout the main evaluation, and the value of the mechanism is in the infrastructure it demonstrates, a search that would size this or any future size parameter on the final specialised RTL for a core whose parameters carried more weight, rather than in the gain it delivers on this particular target.

Appendix B

Combined Clock Cycle Savings per Compilation Build

This appendix gives the per-build data behind the combined-core clock cycle count results of Section 6.5, the counterpart to the hardware-loop data of Section 6.4. Table B.1 lists the fused-and-pruned unpatched clock cycle count of each benchmark in the four compilation builds, the value the loop conversion then tries to reduce further. Table B.2 reports, for each build and at each loop depth (HW), the clock cycle count change of patch-all (PA) and selective (Sel) against that build's own fused baseline, together with the number of loops the selective method drops from the patch-all set (Drp).

The patch-all convertible-loop counts are not repeated here, since loop eligibility depends only on the compilation and is therefore identical to the fusion-free counts of Table 6.9. Only the selective set differs, because fusion shortens a body and so shifts which of its loops repay conversion, and that difference is captured by the Drp column. The cross-build merge is not repeated either, as it is given against the software baseline in Table 6.16.

Table B.1: Fused-and-pruned unpatched baseline clock cycle count of each benchmark in the four compilation builds, before any loop conversion. The builds are *NU* (NoUnroll), *U* (Unroll, the default), *D+NU* (DoLoop+NoUnroll), and *D+U* (DoLoop+Unroll).

Benchmark	NU	U	D+NU	D+U
dilithium	12 992 397	12 523 913	13 417 659	12 635 370
nettle-sha256	5 575 130	5 629 631	5 576 816	5 652 122
nettle-aes	6 572 400	6 595 486	6 664 001	6 668 428
qrduino	5 763 432	5 806 742	5 825 337	5 754 226
picojpeg	5 582 389	5 545 127	5 738 056	5 530 595
slre	5 485 576	5 520 713	5 499 613	5 520 492
sglib-combined	5 040 903	5 272 102	5 071 925	5 289 409
statemate	4 905 096	4 830 224	4 915 213	4 830 231
edn	6 394 322	3 583 994	6 146 692	3 573 593
huffbench	4 662 799	3 958 534	4 557 903	3 983 371
crc32	3 480 612	2 534 383	3 481 412	2 535 526
md5sum	4 133 393	3 224 231	4 151 278	3 214 475
ud	2 945 236	2 907 816	2 947 186	2 909 616
matmult-int	5 307 036	2 379 188	5 323 306	2 438 008
kyber	2 494 044	2 450 821	2 554 867	2 465 036
wikisort	2 484 184	2 210 545	2 588 613	2 217 855
tarfind	2 294 875	1 446 350	2 520 508	1 466 285

Table B.2: Clock cycle count change, in percent, of patch-all (PA) and selective (Sel) for the combined core against each build's own fused-and-pruned baseline, at each loop depth (HW), with the number of loops the selective method drops from the patch-all set (Drp). The builds are *NU* (NoUnroll), *U* (Unroll, the default), *D+NU* (DoLoop+NoUnroll), and *D+U* (DoLoop+Unroll). Each column is against its own build's baseline, so the fusion effect is excluded and the figure is the loop unit's contribution on top of fusion. The averages are taken per loop depth, over the 17 benchmarks at HW=1 and the 7 with a HW=2 row.

		NU			U			D+NU			D+U		
Benchmark	HW	PA	Sel	Drp	PA	Sel	Drp	PA	Sel	Drp	PA	Sel	Drp
dilithium	1	−3.1	−3.2	3	−1.1	−1.1	0	−4.8	−4.8	0	−1.7	−1.7	0
	2	−3.4	−3.5	8	−1.1	−1.1	1	−5.2	−5.2	1	−1.7	−1.7	14
nettle-sha256	1	−3.0	−4.1	1	−0.1	−0.1	1	−4.9	−4.9	0	+0.9	−0.6	7
nettle-aes	1	−4.3	−5.3	2	−2.8	−2.8	0	−1.6	−3.0	2	−2.8	−2.8	0
	2	−5.6	−5.7	5	−2.8	−2.9	3	−1.8	−3.3	5	−3.0	−3.1	3
qrduino	1	−1.8	−2.7	7	−0.1	−0.8	1	−1.7	−4.1	5	+1.4	−0.1	6
picojpeg	1	+1.3	−0.2	3	−0.6	−0.6	0	−2.5	−2.5	0	−<0.1	−<0.1	0
slre	1	+0.2	0.0	1	+<0.1	+<0.1	1	+<0.1	0.0	1	+<0.1	0.0	1
sglib-combined	1	−0.3	−3.4	1	−0.1	−0.4	1	−1.8	−4.0	2	−0.1	−0.2	5
statemate	1	−<0.1	−<0.1	0	0.0	0.0	0	−<0.1	−<0.1	0	0.0	0.0	0
edn	1	−24.2	−24.2	0	−0.4	−2.6	2	−22.7	−25.6	3	−4.2	−4.2	0
	2	−25.0	−25.0	2	−2.4	−3.2	2	−24.5	−27.4	5	−5.8	−6.0	1
huffbench	1	−4.0	−5.4	6	−0.4	−0.4	0	−4.5	−5.3	8	+8.6	−7.0	6
crc32	1	−24.0	−24.0	0	−8.7	−8.7	0	−24.2	−24.2	0	−5.4	−5.4	0
	2	−25.9	−26.3	1	−10.5	−10.5	0	−24.2	−24.2	0	−8.2	−8.2	0
md5sum	1	−14.7	−14.7	2	−1.1	−1.1	2	−11.1	−11.1	3	−1.6	−1.9	2
ud	1	−<0.1	−<0.1	2	+<0.1	0.0	5	−<0.1	−<0.1	5	−<0.1	−<0.1	1
	2	−0.1	−0.2	1	+<0.1	−0.2	5	−0.1	−0.2	5	−<0.1	−0.1	2
matmult-int	1	−33.7	−33.8	2	−1.3	−2.2	4	−32.5	−33.6	2	+1.2	−1.1	1
	2	−33.3	−36.8	7	−1.3	−4.2	7	−11.6	−36.2	6	−1.2	−4.9	5
kyber	1	−5.7	−6.1	19	−2.9	−2.9	0	−7.2	−7.8	19	−3.0	−3.6	19
	2	−5.7	−6.3	24	−3.0	−3.0	0	−7.2	−8.0	10	−3.1	−3.7	24
wikisort	1	−4.7	−4.8	19	−2.1	−2.6	20	−4.0	−6.6	19	−2.7	−3.2	26
tarfind	1	−31.4	−31.4	0	−4.6	−4.6	0	−41.3	−41.3	0	−5.2	−5.2	0
Average, HW=1		−9.0	−9.6		−1.6	−1.8		−9.7	−10.5		−0.9	−2.2	
Average, HW=2		−14.2	−14.8		−3.0	−3.6		−10.7	−14.9		−3.3	−3.9	

Appendix C

Illustrative ARVIS Decision Report for a Specialised RISC-V Core

This appendix reproduces the HTML decision report that ARVIS generates for each run, here for the `crc32` workload discussed in Section 5.6. It records the evidence drawn from the workload, the pruning decisions and narrowed parameter widths, the custom instructions generated, the hardware loops converted on each compilation build with their clock cycle counts, and the variants produced with their measured clock cycles, area, and depth-selection merit. It is included to show the observability the tool provides, not as a result in itself. The report is shown across the following three pages.

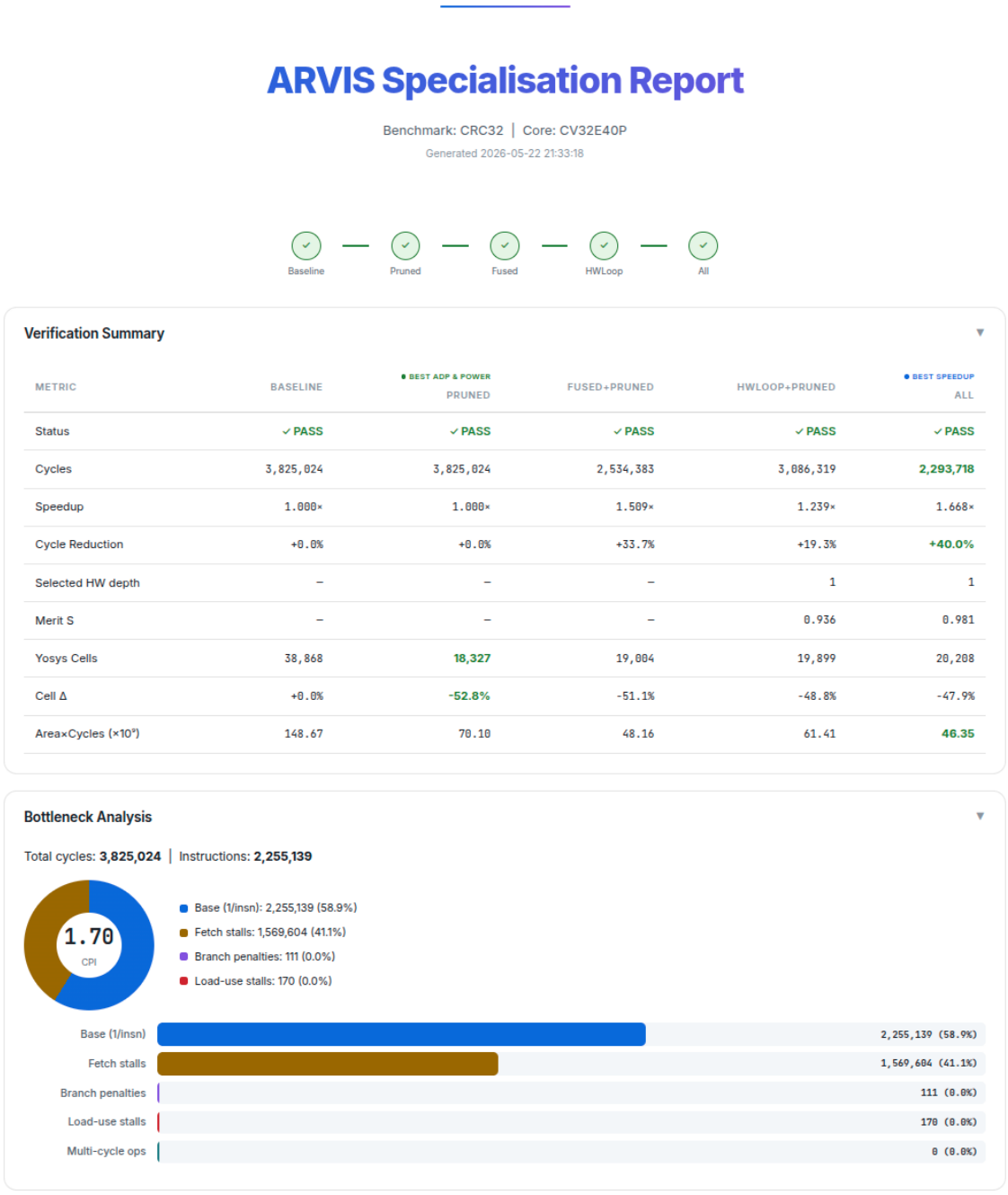


Figure C.1: ARVIS decision report for `crc32`, part 1 of 3, the verification summary and bottleneck analysis.

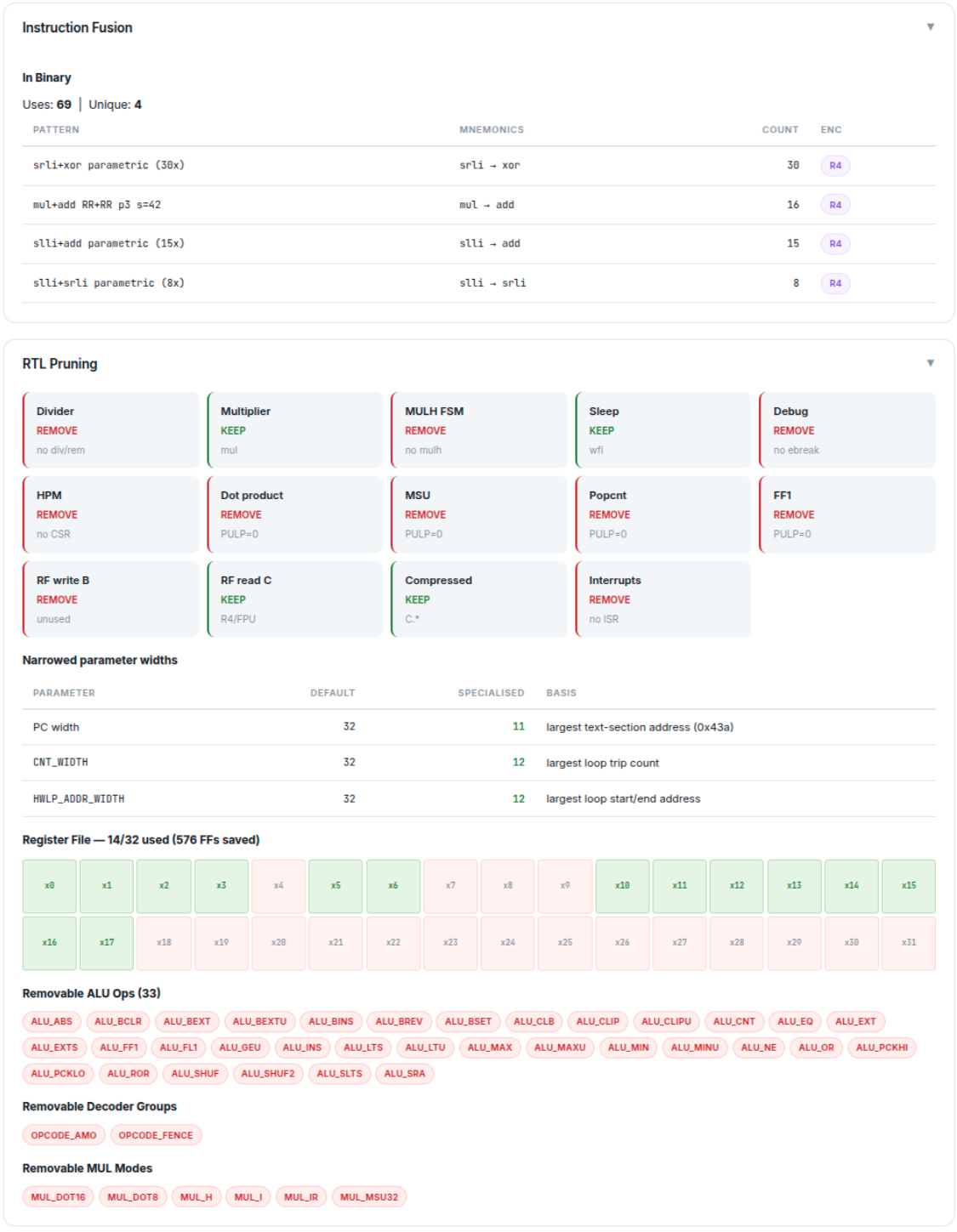
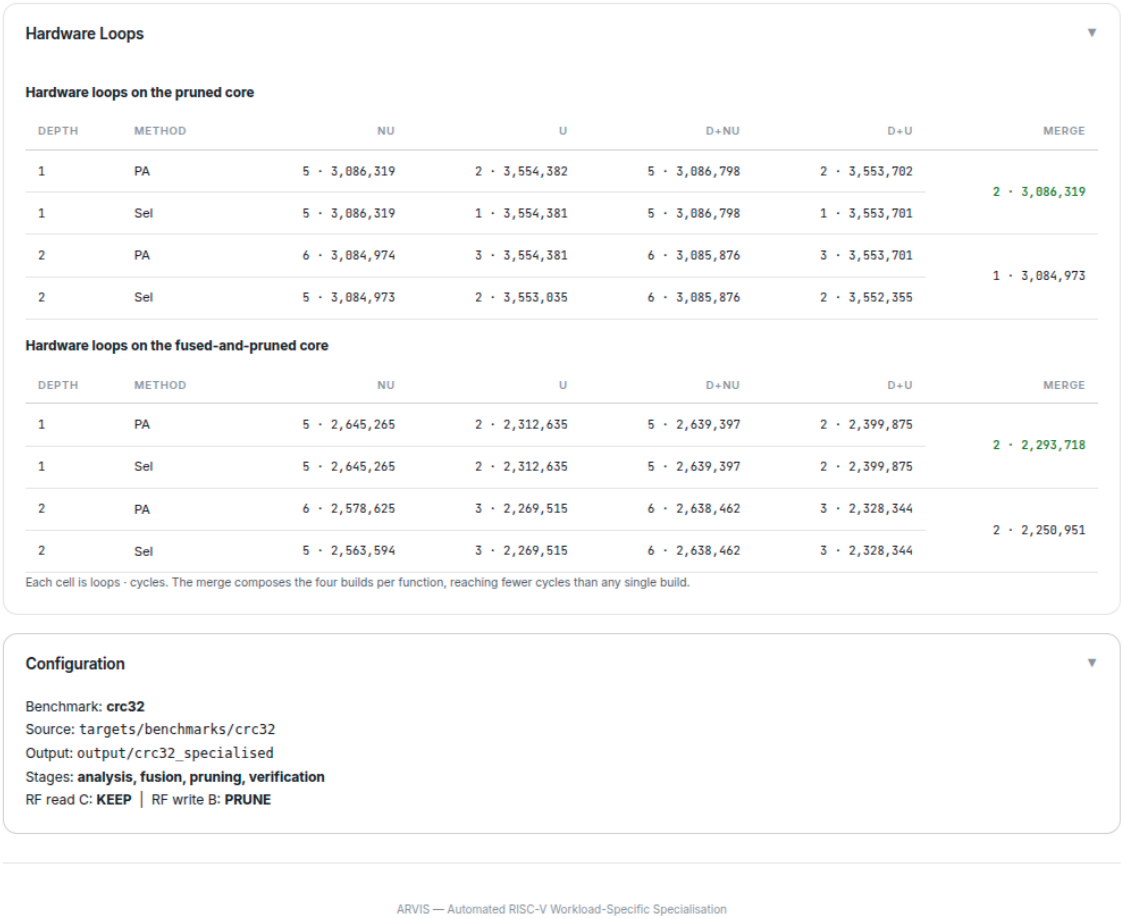


Figure C.2: ARVIS decision report for `crc32`, part 2 of 3, the instruction fusion and RTL pruning decisions.



Hardware Loops

Hardware loops on the pruned core

DEPTH	METHOD	NU	U	D+NU	D+U	MERGE
1	PA	5 · 3,086,319	2 · 3,554,382	5 · 3,086,798	2 · 3,553,702	2 · 3,086,319
1	Sel	5 · 3,086,319	1 · 3,554,381	5 · 3,086,798	1 · 3,553,701	
2	PA	6 · 3,084,974	3 · 3,554,381	6 · 3,085,876	3 · 3,553,701	1 · 3,084,973
2	Sel	5 · 3,084,973	2 · 3,553,035	6 · 3,085,876	2 · 3,552,355	

Hardware loops on the fused-and-pruned core

DEPTH	METHOD	NU	U	D+NU	D+U	MERGE
1	PA	5 · 2,645,265	2 · 2,312,635	5 · 2,639,397	2 · 2,399,875	2 · 2,293,718
1	Sel	5 · 2,645,265	2 · 2,312,635	5 · 2,639,397	2 · 2,399,875	
2	PA	6 · 2,578,625	3 · 2,269,515	6 · 2,638,462	3 · 2,328,344	2 · 2,250,951
2	Sel	5 · 2,563,594	3 · 2,269,515	6 · 2,638,462	3 · 2,328,344	

Each cell is loops · cycles. The merge composes the four builds per function, reaching fewer cycles than any single build.

Configuration

Benchmark: **crc32**
Source: `targets/benchmarks/crc32`
Output: `output/crc32_specialised`
Stages: **analysis, fusion, pruning, verification**
RF read C: **KEEP** | RF write B: **PRUNE**

Figure C.3: ARVIS decision report for `crc32`, part 3 of 3, the hardware-loop conversion per build and the run configuration.

Appendix D

Disclosure of Usage of Generative Artificial Intelligence

The author, Paulo Miguel de Jesus Coutinho dos Santos Fidalgo, acknowledges the use of the following generative artificial intelligence tools in the development of this dissertation. These technologies were utilized under strict supervision, and all AI-generated outputs were critically reviewed and verified for accuracy. The author accepts full responsibility for the validity and originality of the content, confirming that the use of these tools aligns with all institutional policies and standards of academic integrity.

Generative AI tools used: Claude.

Activities supported by Generative Artificial Intelligence Tools:[1]

Conceptualization

- ☐ Idea generation
- ☐ Defining the research objective
- ☐ Formulating research questions and hypotheses
- ☑ **Feasibility assessment and risk evaluation**
- ☐ Hypodissertation viability assessment
- ☐ Simulated debate and argument testing

Literature Review

- ☐ Search and Discovery
- ☐ Literature summarization and syndissertation
- ☐ Concept Mapping and Systematization
- ☐ Market/patent landscape analysis
- ☐ Gap Identification and Novelty Evaluation
- ☐ Cross-lingual literature comprehension

Methodology

- ☐ Experimental Design Optimization
- ☐ Development of experimental or research protocols
- ☐ Methodological Instrument Selection

[1] This list extends the GAIDeT taxonomy.

Software Development and Automation

- ☐ Code Generation
- ☑ **Debugging and Repair**
- ☐ Algorithm design
- ☐ Refactoring and Optimization
- ☐ Process automation
- ☑ **Code documentation and comment generation**

Data Management

- ☐ Data collection
- ☐ Validation
- ☐ Data cleaning
- ☐ Data curation and organization
- ☐ Data analysis
- ☐ Quantitative Plotting and Charting
- ☐ Reproducibility and rerun checks
- ☐ Data labeling and annotation assistance
- ☐ Synthetic data generation
- ☐ De-identification and anonymization support
- ☐ Audio-to-text transcription and diarization

Visuals and Multimedia

- ☐ Synthetic Asset Generation
- ☐ Image Enhancement and Editing
- ☐ Diagrammatic and Schematic Design

Writing and Editing

- ☐ Drafting Text
- ☑ **Polishing and Editing**
- ☐ Abstract and Executive Summary Generation
- ☐ Formulation of conclusions
- ☑ **Tone Adjustment**
- ☐ Translation
- ☐ Citation formatting and bibliography management
- ☐ Press releases and outreach materials
- ☐ Title Generation

Ethics Review

- ☐ Bias analysis and discrimination assessment
- ☐ Ethical risk analysis
- ☐ Monitoring compliance with ethical standards
- ☐ Data confidentiality monitoring

Quality Assurance

- ☐ Simulated Peer Review
- ☐ Consistency checking against field trends
- ☐ Identification of limitations
- ☐ Publication strategy and journal selection
- ☐ Publication support

Additional comment: Claude was used during this dissertation to facilitate the debugging process of both the Python code and the RTL that were developed, helping to find the root causes of the failures that appeared. During the writing of the document, Claude was used to check the correct use of verb tenses, correct writing errors, and help with LaTeX formatting.