

RISC++: Towards an HLS Defined RISC-V SoC

Guilherme Vareiro de Oliveira*, Vinicius Pirassoli*, Luís Miguel Sousa*, Nuno Paulino*

* INESC TEC, Faculty of Engineering, University of Porto, Porto, Portugal
{guilherme.a.cruz, vinicius.pirassoli, luis.m.sousa, nuno.m.paulino}
@inesctec.pt

Abstract—The relevance of heterogeneous architectures has significantly increased over the last decade due to stagnation of performance scaling. Concurrently, increased performance-energy tradeoff requirements driven by the growth of edge computing, with a large focus on Artificial Intelligence (AI) inference, further motivates efforts towards hardware customization. In this context, the open RISC-V Instruction Set Architecture (ISA) and its custom extension oriented paradigm are a relevant technology towards this specialization. However, customizing a processor is a lengthy process requiring Hardware description language (HDL) expertise. Furthermore, for validation and simulation purposes, implementing an Instruction Set Simulator (ISS) of the modified core may also be a necessity. This introduces the need for development of two unrelated codebases, increasing development time and effort. In this paper, we explore High-Level-Synthesis (HLS) to realize both the hardware and the respective simulator through a single codebase, which reduces design effort and simplifies specialization of a RISC-V through specification of custom instructions at high-level. We present a C++ based design of a RISC-V core, and validate it as an ISS, as well as a hardware module synthesized for an AMD Zynq UltraScale+ Field Programmable Gate Array (FPGA) through HLS, which we integrated in a System-on-Chip (SoC), and functionally validated through a state-of-the-art set of unit tests.

Index Terms—RISC-V, High-Level-Synthesis, System-on-chip

I. INTRODUCTION

Demand for computational power continues to increase, especially given the rise of edge AI which promises to find application in a wide range of domains. The requirements of AI inference, such as low latency and access to large volumes of data, have exacerbated the already existing problems in computing architectures, like Moore and Dennard breakdowns and the memory wall. To address this, focus has shifted towards specialized hardware to achieve greater energy-efficiency. In this context, RISC-V, with its modular oriented and open ISA, emerges as a pivotal enabler. By implementing custom instruction set extensions, designs can deliver optimized computation, reduced memory bottlenecks, and energy-efficient processing.

However, the development of custom processors and SoCs typically relies on Hardware description languages (HDLs) like Verilog or VHDL, where the design and debug efforts require specialized expertise. Additionally, custom core design

This work has been partially sponsored by the Portuguese Science Foundation (FCT) under research grant SFRH/BD/10002/2022, and has received funding within the Chips Joint Undertaking (Chips JU) - the Public-Private Partnership for research, development and innovation under Horizon Europe – and National Authorities under grant agreement 101096658.

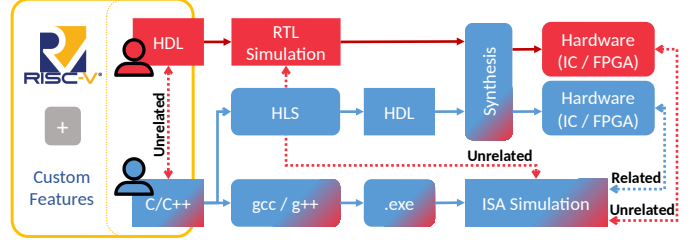


Fig. 1. Comparison of a typical HDL design flow vs. our proposed HLS-based flow.

may need to be supported by a Instruction Set Simulator (ISS) for validation and simulation purposes, which is implemented separately. Maintaining two separate codebases increases development effort, requires different skillsets, and increases the burden of verification between simulator and hardware. This can increase development time and costs, and limit accessibility for developers without an extensive hardware design background [1], [2].

Motivated by these issues, this HLS-based workflow aims to streamline the development of custom RISC-V cores and systems. This avoids the necessity of developing and maintaining two codebases, one written in a high-level programming language like C/C++ for simulation, and another in a hardware description language. This may reduce overall production costs, reduce time-to-market and avoid development problems, since the simulator and the hardware itself originate from the same code. Furthermore, since development of hardware accelerator peripherals is already well supported through HLS flows, a unified design method towards specialized SoCs can be achieved by addressing Central Processing Unit (CPU) design through the same tools.

Figure 1 illustrates the typical flow based on separate codebases, versus the use of a single code base used for ISS and HLS. Besides improving design time and maintainability, the effort of extending the RISC-V with custom instructions is also simplified, which can aid in realizing the promised gains of domain-specific hardware.

In this paper we present RISC++, a RISC-V core currently supporting the RV32IM instruction set written in C/C++. We functionally validate it as an ISS, as well as a hardware module integrated into a small SoC, deployed onto a ZCU102 UltraScale+ FPGA. We also discuss different coding strategies as they relate to synthesis optimizations, and respective hardware resource metrics for the ZCU102 as well as for the PYNQ-Z2.

TABLE I
COMPARISON OF FREQUENCY AND RESOURCE METRICS FOR
STATE-OF-THE-ART RISC-V CORES TARGETING FPGAs.

Core	ISA	Freq. (MHz)	LUTs	FFs	Mux	DSPs
Comet [3] (2019)	rv32i	80	2,032	1,503	260	0
	rv32im	70	2,910	2,244	227	3
	rv32imf	74	6,460	3,527	448	5
Rocket [4] (2016)	rv32i	76	2,253	1,154	41	0
	rv32im	76	2,570	1,275	43	2
	rv32imf	76	8,132	3,094	586	4
PicoRV [5] (2019)	rv32i	140	880	583	0	0
	rv32im	110	1,977	1,085	0	0
Toker [6] (2023)	rv32i	100	1,078	326	0	0

II. STATE-OF-THE-ART

We now review works which, to the best of our knowledge, are the only designs which are also based on HLS flows for generation of RISC-V cores, as well as reference RISC-V ISSs or similar simulators. A summary of the frequency and resource usage of representative RISC-V cores targeting FPGAs is provided in Table I.

A. RISC-V Cores

In [6] the authors implemented a RV32I core using an HLS workflow. Through AMD Vitis' block design tools, the core was integrated into an SoC with a 4KB Block Random Access Memory (BRAM). The design prioritized readability and maintainability of the C++ code, while keeping low area and complexity of the core. The SoC was tested through high-level simulation, Register Transfer Level (RTL) simulation and validated on an AMD Basys3 FPGA, where the synthesized design used 1078 Look-Up Tables (LUT), 326 Flip-Flops (FF) and 3 % of the BRAM. The achieved clock frequency was of 100 MHz, leading to a power consumption of 81 mW. Gate-level synthesis, targeting an ASIC implementation, resulted in resource requirements of 1377 D-type FF, 7415 2-NAND, 5219 2-NOR, and 1311 NOT gates.

Mantovani *et al.* present the HL5 32-bit RISC-V core, claiming it is the first HLS design through SystemC [7]. Akin to our approach, the authors aim to simplify and accelerate RISC-V core design while providing easy customization. The HL5 core implements the full RV32IM subset of the RISC-V ISA and is a pipelined in-order micro-architecture. The design was tested in simulation and validated through implementation on a Zynq SoC FPGA. It also presents an estimation of area requirements targeting a 32 nm node for their various designs using different optimization parameters. A relevant design difficulty was on how to specify the correct behavior of a true pipelined CPU through HLS. To address this, the authors resorted to concurrency mechanisms, utilizing semaphores as virtual-clock boundaries, and channels between multiple processes to enforce synchronization between pipeline stages. The design was compared with the ZERO-RISCY [8] core,

which is written at RTL level, and supports the same ISA extensions. Design variants of the HL5, for an ASIC flow, achieved operating frequencies between 700 MHz and 2 GHz, and a marginally higher Instructions per clock (IPC) versus the ZERO-RISCY's IPC of 0.23, at a significantly higher area cost. Overall, the design effort versus conventional HDL is reduced, with 2000 Lines of code (LOC) required through SystemC versus 6000 LOC of the ZERO-RISCY design, at the cost of 35 % to 50 % more area for the same performance.

Rokicki *et al.* present a C++ based core which they have open-sourced [3]. The authors explore the specification of parallel computing pipelines with HLS tools, aim to achieve reduced design times, and increased expandability, without performance or area compromises. To demonstrate this, the design was compared to the HDL based Rocket [4] and PicoRV [5] cores. The authors argue that HLS approaches cannot directly produce efficient hardware based on ISS-like code, due to their limitations in breaking loop carried dependencies which are typical in these simulator implementations. To address this, a technique is applied to enforce a true pipeline structure with an Initiation Interval (II) of one clock cycle, by explicitly encoding the structure at high-level via explicit data dependencies. With those modifications to a base ISS, they present their 5-stage pipelined RISC-V core, named *Comet*. The variants (RV32I, RV32IM, and RV32IMF) were synthesized with a target clock of 700 MHz targeting a 28 nm node. The area obtained for each of the configurations was $8168 \mu\text{m}^2$, $11\,099 \mu\text{m}^2$, and $26\,760 \mu\text{m}^2$, respectively. These area requirements are similar to those of the Rocket core, for the same clock frequency of 700 MHz. When synthesizing for a Xilinx Artix 7 FPGA, the obtained results are also comparable to the Rocket core, and both cores require more area than the PicoRV. Resorting to the Dhrystone benchmark suite [9], the IPC of Comet is comparable to the Rocket core's IPC of 0.53, and significantly higher than the PicoRV's 0.24 IPC. Additionally, a custom butterfly instruction for Fast Fourier Transform (FFT) operation was implemented. This resulted in a 31 % area increase, but a $14\times$ faster execution speed for FFTs, demonstrating fast core extensibility at high-level at reduced design effort.

Finally, Bernard Goossens presents a lengthy explanation on HLS oriented RISC-V design in [10]. This book presents incremental designs of synthesizable cores written in C code. It is a detailed primer on this type of approach, but is presented as educational material to promote adoption of novel approaches to computer architecture design leveraging modern tools. It presents single- and multi-cycle (pipelined) designs of cores supporting the mandatory RV32I subset, but addressing other extensions (namely M and F), are left as exercises, and custom instructions are not addressed. Regardless, it is a significant illustration of processor design through HLS.

B. RISC-V Instruction Set Simulators

We briefly summarize state-of-the-art RISC-V ISSs and digital system simulators, which provide simulations at different abstractions above RTL level.

Spike is the reference behavioral simulator for RISC-V maintained by RISC-V International [11]. It is a C/C++ simulator which is instruction accurate, and supports the majority of ratified extensions as well as features like hardware threads (*harts*), and user, machine and supervisor modes. However, it requires forking and modification of its codebase to support custom extensions.

RISC-V VP is a virtual prototype platform for RISC-V, built on the SystemC TLM (Transaction-Level Modeling) 2.0 framework [12]. It supports a generic bus system to model SoC-like designs, and integrates cycle-accurate timing models through the SystemC-TLM framework. Use of SystemC results in a code design which is more aware of the architecture aspects of the system being simulated. The simulation is closer to hardware level versus high-level simulation, but still more abstract and faster than RTL simulation while still being cycle-accurate. Extending the RISC-V in the system with custom instructions requires modifying the SystemC model of the core, which due to the level of abstraction requires modeling transaction level behavior as well.

QEMU (Quick EMULATOR) is an open-source emulator that supports various architectures, including x86, ARM and RISC-V [13]. It provides a high-level system simulation, including hosting of operating systems and specification of device trees. It provides instruction-accurate simulation by runtime binary translation to the host machine running the QEMU executable, similar to a virtual machine. Despite this, it is one of the fastest system simulators, but is focused on providing a stable environment for cross-platform software development, rather than micro-architecture design. Modifying QEMU to support custom RISC-V requires a deep understanding of its extensive C++ codebase, in addition to then implementing new binary translation rules for execution on a host.

III. RISC-V CORE DESIGN IN C++

Our RISC++ approach is motivated by the fact that typical flows for designing and testing a new processor core require maintaining two codebases, one for the hardware core itself, and a second for some type of behavioral simulator capable of running software compiled for the core. Besides the increased development effort, as the codebases are unrelated, there is no guarantee that the correct simulation implies correct hardware implementation, or that hardware bugs can be resolved through aid of simulation. In the following sections, we will explain the code structure of the RISC++ core, as well as specific source optimizations for the HLS flow, and describe the setup for deploying and testing the core as an ISS, and on an AMD Zynq UltraScale+ FPGA.

A. RISC++ Source Code Structure

By adapting the approach from [10], our design resorts to a mix of C++ classes to model the various components of the core, and a functional C approach. The current implementation supports the RV32IM set of instructions. Figure 2 illustrates a simplified high level view of instances of these classes, which are members the `Core` class (i.e., `core.hpp`

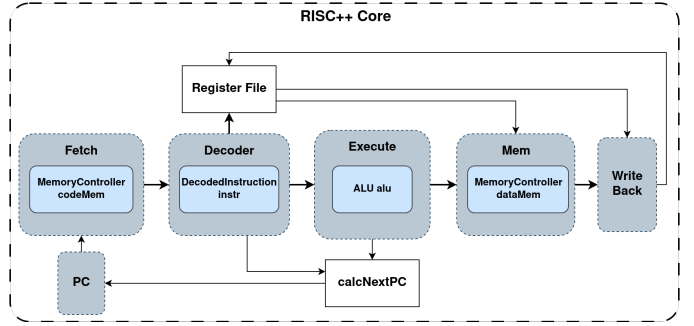


Fig. 2. Simplified overview of RISC++'s class structure (Control Unit omitted).

/ `core.cpp`). Major classes are those that model the data-path stages, i.e., `Decoder`, `ExecStage`, `MemStage`, and `WBStage`. Throughout the code, we leverage features such as the `ap_uint` template for specification of any n-bit unsigned integer to create signals such as the distinct instruction fields or control signals. We also exploit the `ap_wait` synchronization primitive to achieve specific pipeline behaviours.

The decoder stage consumes instructions fetched by one instance of the memory controller class, which is fed the *Program Counter*. It splits the instruction fields based on *opcode*, while also implementing the *IMM* field calculation, which itself depends on *opcode*, or on the specific instruction in some instances (e.g. shifts). The `ControlUnit` class generates all control signals to drive the logic in the following stages, such as Arithmetic and Logic Unit (ALU) operand selectors, and boolean control signals such as the *write enable* for the register file.

In this RV32IM version of the design, the execute stage contains only an ALU, which is driven by the `aluCtrl` signal to select an operation, the `opA` operand which is read from the `RegisterFile` class, and the `opB` operand which is either also read from the register file or is the generated immediate value. Extending this class to support the F extension for instance, would only require adding a Floating Point Unit (FPU) to the `ExecStage` class and adapting the `Decoder` class. The control unit would also need to be extended to produce the additional control signals required by this logic.

The structure described above is shown in the excerpt of the `Core_I::run` method, in Figure 3b, where some call arguments have been omitted for brevity. The loop will iterate while the decoded instruction is not a *return* from the *main* function of the executable. At the end of operation, the number of instructions executed is sent through an Advanced eXtensible Interface (AXI) interface. The `pragma` statements applied to this function implement the partition of the register file as a distributed memory, and inline all function calls.

Figure 3a shows the `Core` object being instantiated in a top-level `riscpp` function which is the target function for hardware synthesis. In the current design, instruction and data memories are implemented as either AXI or BRAM interfaces. These memories are preloaded, either at synthesis or by the

```

void riscpp_top(unsigned int code_ram[CRAMSZ],
               unsigned int data_ram[DRAMSZ],
               unsigned int *instcounter) {

#pragma HLS INTERFACE bram port=data_ram
#pragma HLS INTERFACE bram port=code_ram
#pragma HLS INTERFACE mode=s_axilite port=return
#pragma HLS INTERFACE mode=s_axilite
    ↪ port=instruction_counter
#pragma HLS inline recursive

    Core riscpp(data_ram, code_ram, 0);

    riscpp.run();

    *instcounter = riscpp.instrCounter;
}

```

(a) top level function which is synthesized to hardware

```

void Core::run() {
#pragma HLS inline recursive
#pragma HLS array_partition variable=rf type=complete
main_loop: while(running) {
    instbits_t ibits = /* fetch */
        codeMem.loadWord(pc << 2);

    DecodedInst inst = /* decode */
        decoder.decode(ibits);
    ctrlUnit.calcCtrl(ibits.range(6, 0),
        ibits.range(14, 12), ibits.get_bit(30));

    pc = nextPC(pc, inst.imm, /* inc PC */
        rf[inst.rs1], /* ..ctrlUnit, exec.. */);

    regtype_t aluRes = /* execute */
        exec.execute(inst, rf, pc,
            ctrlUnit.aluctrl, /* ..ctrlUnit.. */);

    regtype_t ldRes = /* memory */
        mem.access(aluRes, rf[inst.rs2],
            ctrlUnit.load_type, ctrlUnit.store_type);

    wb.writeBack(aluRes, rf, /* wb */
        ldRes, pc, inst.rd, /* ..ctrlUnit.. */); }
}

```

(b) main loop method of the Core class

Fig. 3. Code excerpts of the top level function and main loop function of the core for RISC-V instruction execution (details omitted)

Zynq’s ARM core, before the core begins execution by calling the run method of Core class. When using BRAMs, the design benefits from lower latency, whereas AXI interfaces enable access to larger DDR-backed memory at the cost of increased latency due to the inherently slower access times of DDR compared to BRAM. Future iterations of the core will incorporate a hybrid memory architecture, combining BRAM and AXI-based interfaces to implement a cache capable memory system that balances speed and capacity.

The run function (depicted in Figure 3b) contains the main CPU loop where each processor stage is called in the appropriate sequence. We employ a `#pragma HLS inline recursive` at the top level function, which will flatten the kernel’s function hierarchy, since we have observed that it significantly improves area and performance statistics by allowing the tools to increase the amount of shared resources.

While the core can be either synthesized to hardware or compiled as an ISS, it must be hosted by different wrappers depending on the environment, where the data and code memories must be initialized and passed as arguments to the top-level function (explained in Sections III-D and III-E).

B. Performance optimizations related to HLS

As we increased the amount of logic in our design, and especially when attempting higher frequency targets, the known limitations of HLS tools to handle control-heavy data paths affected the results. In particular, the presence of variable dependencies, data-dependent control flow, and limited operation parallelism started to restrict the tool’s ability to efficiently pipeline the datapath. These issues, intrinsic to complex CPU-like architectures, demanded careful code restructuring and optimization strategies tailored to the characteristics and limitations of HLS. This section outlines the key optimizations employed to mitigate these scheduling barriers and improve timing constraints.

To achieve maximum throughput in a pipelined architecture, where a new instruction is initiated every clock cycle, the pipeline must have an II of 1. This ensures continuous instruction issue without stalls. In Vitis HLS, this can be directed through the `#pragma HLS PIPELINE II=1`, which instructs the tool to attempt scheduling the pipeline to accept a new iteration every cycle. While Vitis HLS is capable of meeting this target, it typically does so by designing a relatively shallow pipeline composed of three main stages: Fetch, followed by Decode + Execute and Memory + WriteBack. This structure simplifies the scheduling of operations and resolves most data and control dependencies. However, it does so at the cost of operating at a lower maximum frequency, as the clock period must be long enough to be capable of computing all operations within each combined stage. This trade-off illustrates a core limitation of HLS when automatically pipelining control-dominated designs. With the key takeaway being that achieving an II of 1 is possible, but frequency scaling is constrained by the compiler’s current capabilities to handle variable control dependencies.

To address the frequency limitations imposed by the enforced II=1 constraint, we adopted an exploratory approach by removing the initiation interval directive, and instead allowing Vitis HLS to synthesize the design employing its built-in default strategies, while targeting the higher desired clock frequency. By lifting the strict II requirement, the compiler could leverage more flexibility in scheduling operations and restructuring control paths to better meet timing constraints. This approach enabled us to analyze the resulting operation scheduling, and identify the specific dependencies and bottlenecks that prevented the target II of 1 cycle. The synthesized design, a 4-stage pipeline with II=3, served as a reference for understanding what dependencies were present in the datapath, guiding us in applying more targeted code-level optimizations to solve the dependencies.

1) *Branching Unit:* The first dependency reported by the tool that we addressed was related to the value of the *Program Counter*. Originally, the PC was updated during the Execute stage, as its new value could depend on the result of the ALU in the case of branches or jumps. However, the PC value is needed for address resolution at instruction fetching which occurs at the first stage of the pipeline. With Execute now


```

HazardDetectionUnit::HazardDetectionUnit() {
    stalled = false;
    istall = false;
}

void HazardDetectionUnit::detectBranch(ap_uint<6> op) {
    if ((op == BRANCH || op == JALR
        || op == JAL) && !stalled) {
        istall = true;
        stalled = true;
    } else {
        stalled = false;
    }
}

void HazardDetectionUnit::detectLoad(bool lastInstWasLoad,
    rs_t rs1, rs_t rs2, rd_t lastRd) {
    if (lastInstWasLoad &&
        (rs1 == lastRd || rs2 == lastRd)) {
        istall = true;
    }
}

```

Fig. 4. Code excerpts of the Hazard Detection Unit class

being positioned at the third stage, the delay in updating the PC created a Read-after-write (RAW) dependency with a distance of two cycles.

To solve this issue, we introduced a Branching Unit into the core architecture. This unit computes the next value of the PC earlier in the pipeline, removing the dependency on the ALU output in the Execute stage. The unit is implemented as a separate combinational function that takes as input the current instruction, immediate values, and register file, and determines the next PC based on the branch type and control signals. This restructuring effectively moved the control flow decision to the second pipeline stage, breaking the dependency chain.

2) *Conditional Stalling*: Following the introduction of the Branching Unit and the restructuring of the pipeline, a new class of dependency emerged in the synthesized design. Specifically, we observed a data hazard between the result of a load instruction, which becomes available only at the MEM/WB stage, and the input operands required by the ALU in the subsequent instruction at the Execute stage. That is, a read-after-write (RAW) dependency at a distance of one cycle.

This type of dependency is well-known in traditional HDL implementations of in-order CPU datapaths and is commonly associated with memory dependence hazards, where a consumer instruction requires data that is still unavailable from a previous load. If not handled properly, this hazard can lead to incorrect computation. In our case, the HLS tool attempted to mitigate it through conservative scheduling, which means constructing a pipeline with an II of 2, even though only some types of instruction combinations lead to that data hazard.

To address this, we implemented a Hazard Detection Unit (HDU) and a Stalling Mechanism, taking inspiration from the RAW solving strategy proposed by Alonso et al. in [14]. The HDU identifies RAW hazards where the result of a load instruction is used by the next instruction, as is demonstrated by the code snippet in Figure 4. Upon detecting such a condition, our stalling mechanism introduces a pipeline bubble (a no-op instruction), which effectively delays the dependent instruction, giving enough time for the data to become available. This approach is equivalent to what the paper refers to

as *behavioral hazard stalling*.

This manual management of the pipeline guarantees correct program execution without requiring unnecessary conservative solutions. To prevent the HLS tool from introducing stalls when not needed, due to it still perceiving RAW dependencies in the regFile, we explicitly inform the tool that these are false dependencies by using the following pragma:

```

#pragma HLS dependence variable=regFile type=false
↪ direction=RAW

```

This directive suppresses the tool’s default dependency analysis between reads and writes to the regFile, which are now handled manually by our HDU and stalling logic. This is crucial for the tool to pipeline the CPU loop with an II of 1. However, while the reported II is 1, the actual effective II becomes dependent on the instruction stream. That is, if a program contains back-to-back dependencies, bubbles will be inserted, reducing throughput. Thus, the IPC of the core is now data-dependent, varying with the mix of dependent and independent instructions, just as in traditional designs.

3) *Forward Unit*: As a consequence of the manual dependency override introduced in the previous optimization, we observed that the HLS tool no longer handled even benign, intra-cycle dependencies (even when specifying it in the pragma). Specifically, instructions that relied on the result of the immediately preceding ALU operation experienced unresolved data hazards, despite being resolvable through forwarding in the same cycle. Without dependency tracking, these hazards could lead to incorrect execution due to outdated operand values being read from the register file.

To address the problem, we extended our existing HDU to also detect such ALU-to-ALU hazards, where a destination register of a preceding instruction matches the source register of the current instruction. In conjunction, we implemented a Forwarding Unit that bypasses the ALU result directly to the dependent operand input in the following instruction. This mechanism ensures that register coherence is maintained, allowing the pipeline to continue without stalling.

It is also important to note that once we manually disable dependency checking for a variable, the C-simulation may no longer accurately reflect the behavior of the synthesized hardware. The intra-cycle dependencies do not appear when running the core as an ISS or when performing a C-simulation, so to debug the behavior of the core, we chose to feed the pipeline with custom tests that provoked those intra-cycle dependencies. That way, we were able to characterize the erratic behavior and provide solutions to the pipeline architecture.

C. Compilation Flow

The RISC++ design relies on separate data and code memories allowing fetching and memory write operations within the same cycle. Therefore, we rely on the flow shown in Figure 5 to generate the contents for these memories. Programs are compiled using the standard RISC-V GNU Compiler toolchain

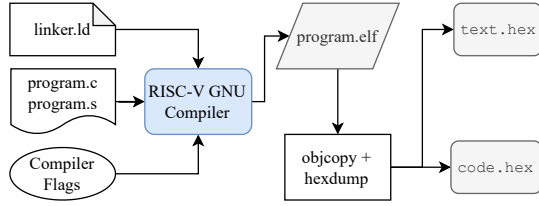


Fig. 5. Compilation flow to generate separate contents for the data and code memories.

¹, and the resulting elf files are separated according to their .data and .text sections using objcopy, with the --only-section flag, followed by a call to hexdump, resulting in two hex files.

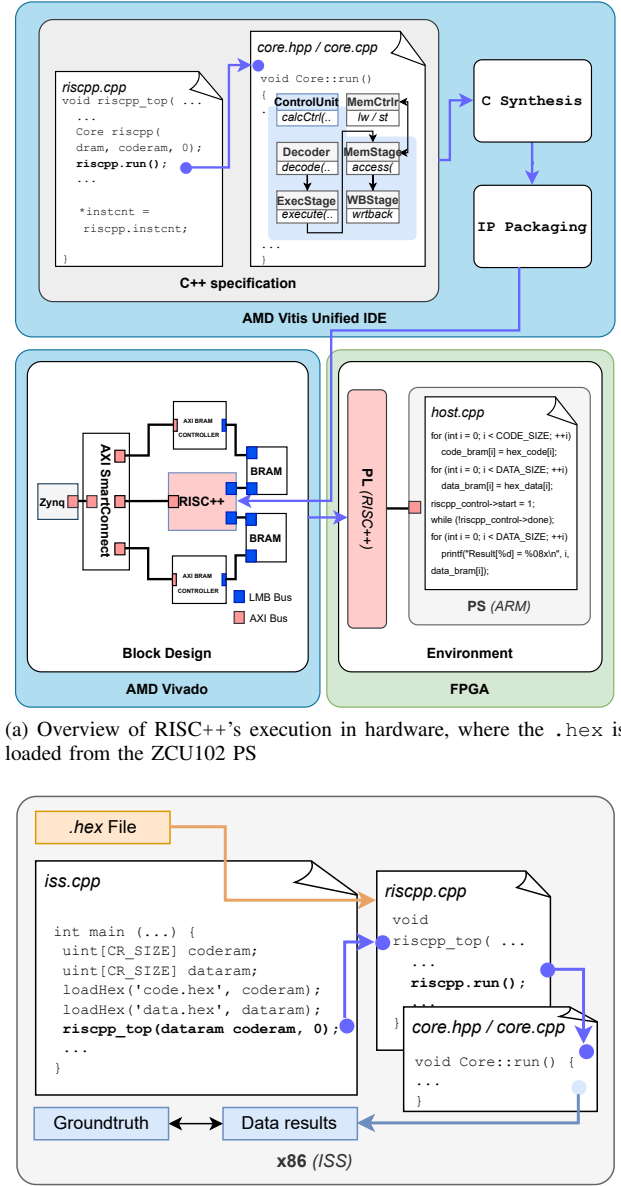
The contents of these files are then loaded into the code_ram and data_ram memories by a host program or when generating the bitstream for the FPGA. Depending on the target application, a linker script may be required to ensure that the .text and .data sections are properly organized for the objcopy step. Makefile and Python scripts have been created to automate the compilation process for programs, as well as creating the necessary files to preload the core BRAMs.

We support only the user-space RV32IM instruction subset, meaning that environment or operating system-level instructions, such as ecall, are not supported. Given this limitation, programs must be written or modified to ensure that the resulting binary adheres to the user-space instruction set specifications. For instance, we cannot resort to routines like printf to perform functional validations at this stage. Instead, since the global memory is shared between the RISC++ and the host, the core can simply dump register contents or other status data to host RAM, which is then read by the testbench code and checked for correctness.

To target our baremetal environment, the GNU C compiler and linker must be provided with specific flags. The -nostartfiles flag prevents the inclusion of standard startup code, such as crt0.o, which is responsible for initializing runtime environments in OS-based applications. Omitting these startup routines ensures that no system calls or environment setup is performed. The -static flag forces all dependencies to be statically linked, eliminating the need for dynamic linking and shared libraries, which require OS support to load at runtime. Additionally, the -Wl,-no-check-sections linker flag disables section checking, allowing the program's memory layout to be freely defined without the constraints typically enforced by an operating system.

D. Deployment of RISC++ on FPGA

The deployment of the RISC++ core follows the standard Vitis HLS to Vivado IP flow, and our target platform is an AMD Zynq UltraScale+ ZCU102 FPGA. Firstly, the core is synthesized from its C++ description in Vitis HLS, applying optimization and timing constraints. After synthesis, the design



(a) Overview of RISC++'s execution in hardware, where the .hex is loaded from the ZCU102 PS

(b) Overview of RISC++ compiled as an ISS

Fig. 6. Overview of both execution and deployment of RISC++.

is packaged as a Vivado IP block. In Vivado, we add one instance of the core IP to a small SoC containing two true dual-port BRAMs, one for code_ram and one for data_ram. The BRAMs are accessed by the RISC++ core via an LMB BRAM controller, providing fixed 1 cycle latency, and by an AXI bus by the ARM core, which starts the execution and provides debugging. These AXI controllers are connected to the Zynq MPSoC, where the ARM cores are used solely for debugging purposes. By allowing the ARM cores access to the BRAMs, we can write programs like the one demonstrated in Figure 6a that preload memory, control execution, and inspect system state, allowing seamless testing and validation of the RISC++ core.

¹<https://github.com/riscv-collab/riscv-gnu-toolchain>

TABLE II
METRICS FOR RISC++, FOR RV32I AND RV32IM CONFIGURATIONS ON
ZCU102 AND PYNQ-Z2

Target	ISA	HLS Source Optim.	Freq. (MHz)	LUTs	FFs	DSPs	II
ZCU102	rv32i	Yes	100	2,378	1,274	0	1
		No	100	2,272	1,541	0	1
		No	150	3,614	1,478	0	2
		Yes	150	3,217	1,621	0	1
		No	300	1,566	1,538	0	4
	rv32im	Yes	100	3,001	1,276	3	1
		No	100	2,357	1,539	3	1
		No	150	1,648	1,470	3	2
		Yes	150	3,251	1,621	3	1
		No	300	1,591	1,610	3	5
PYNQ-Z2	rv32i	No	30	2,261	1,259	0	1
		No	50	1,629	1,342	0	2
		Yes	50	3,531	1,560	0	1
	rv32im	No	30	3,210	1,262	3	1
		No	50	2,502	2,400	3	2
		Yes	50	2,435	1,660	3	1

E. Executing RISC++ in ISS Mode

A testbench program (`src/testbench.cpp`) has been developed to manage the execution of RISC++ in ISS mode. This testbench can load `.hex` files via command line input flags, which are handled by the `CmdLineParser` class from Xilinx’s Vitis library. The code and data memories are loaded to arrays passed as arguments to the `run` method of the core object as explained. An overview of the ISS mode’s execution is demonstrated in Figure 6b. Additional features of the testbench include the ability to dump data memory to the console or a `.txt` file, as well as automatically loading certain tests and benchmarks, provided they have already been built into `.hex` files. Upon completion, the testbench prints the total number of instructions executed by the core.

IV. EXPERIMENTAL RESULTS

We have validated the compilation of the RISC++ code as an ISS through conventional compilers, as well as RTL level simulation through Vitis, and its synthesis to hardware. We evaluated the hardware resource requirements, and executed unit test binaries for both the ISS and the hardware instances.

A. Hardware Synthesis Results

We have synthesized our core using version 2024.2 of AMD’s Vitis Unified IDE, with various target clock frequencies, from 100 MHz to 400 MHz. Table II shows our synthesis results, where we achieve comparable resource utilization and maximum operating frequency relative to existing approaches shown in Table I. The data and code memories are implemented as arrays synthesized as BRAMs, with enough capacity to accommodate the data and code sections of the unit tests.

The resulting synthesis and post-route reports indicate that an II of 1 clock cycle is possible for 100 MHz. However, to achieve higher frequencies the optimizations presented in

section III-B are required to obtain an II of 1. For instance, for the RV32IM variant, we successfully synthesize the core for 100 MHz, 150 MHz, 300 MHz, and 400 MHz. However, the respective IIs are 1, 2, 5, and 8. Therefore, the 100 MHz case achieves this effective performance due to the II of 1. However, the 150 MHz case leads to an effective frequency of 75 MHz in comparison to the first case. For the remaining two cases, the effective frequencies are 60 MHz and 50 MHz respectively.

From the analysis of our synthesis results, we saw that even though the tool is capable of generating pipelines for high frequency operation without any optimization specific to HLS flows, they are not necessarily better, since without optimizations the throughput decreases as the target frequency exceeds 100 MHz. In contrast, with source optimizations, an operating frequency of up to 150 MHz with an II of 1 is achieved, while having minimal impact on resource usage. So, we conclude that manual optimizations offer better return on resource investment than tool-directed scaling.

Synthesis on both the ZCU102 and PYNQ-Z2 demonstrated consistent scalability, requiring only changes to tool constraints, validating the core’s portability across FPGAs with different BRAM and LUT footprints. However, as it is expected, clock frequencies for the PYNQ-Z2 are significantly lower due to the ZYNQ 7020’s FPGA fabric.

Finally, designs with higher IIs tend to allow better resource sharing, which in turn leads to smaller overall hardware footprints. While a low II of 1 maximizes throughput, increasing the II can reduce resource duplication by reusing functional units across multiple cycles. This trade-off between performance and resource usage is critical for optimizing area-constrained designs, where a slightly higher II can yield significant savings in resource consumption.

These results place RISC++ as a lightweight, customizable core, comparable to state-of-the-art approaches, and viable for integration in FPGA SoCs targeting edge applications.

B. Functional Validation

For validation purposes, we successfully compiled and executed the official RISC-V Tests repository [15]. This repository contains numerous unit tests designed to verify RISC-V implementations, including our target architecture: user-space RV32IM. The tests are RISC-V binaries generated by a series of configuration and setup macros. These macros configure the sections of the generated binary, and resolve to code snippets which execute the instruction under test, and verify contents of the register file or data memory after execution. The RISC-V tests are designed to return a nonzero value whenever a test fails, so verification is performed by reading data memory values with an address higher than a pre-defined value of `0x2000`. If all memory words with a higher address are null, all tests have passed.

To ensure compatibility with our design, the test source code was modified to eliminate all environment calls and OS-dependent operations. Additionally, all print or output writing commands were replaced with instructions to load results into

data memory, and a `_start.S` main file was added. These modifications to the repository borrow from the approach in [10]. The design passes all self-contained tests for the RV32I and RV32M subsets, for the ISS execution, as well as the execution on FPGA. In addition, we developed several additional minor tests, such as testing if data dependencies were being correctly resolved, in RISC-V assembly. Validating functionality and characterizing pipeline behavior, throughout the implementation process.

V. ONGOING WORK

Currently, we are characterizing performance through benchmark suites such as MiBench [16] and PolyBench [17], and simultaneously exploring new optimization logic to target a 200 MHz design with `II` equal to 1, when targeting the ZCU102. In future work, we will improve the integration of the core into the SoC, specifically its peripheral interface capabilities, which are currently hindered by the HLS compilers' assumptions regarding bus latency, which compromise the `II` of the core. We are also adding UART and GPIO connectivity, as well as implementing AXI interfaces between the core and other accelerators or memory interfaces. Lastly, we are trying to enhance the flexibility of our core implementation by exploring the composability of RISC-V extensions, allowing the user to instantiate the core at the top-level function as an object, and through template parameters, select which extensions are implemented.

VI. CONCLUSION

We have presented a C++ based design for a RISC-V processor supporting the RV32IM set of instructions, and demonstrated its functional correctness as an ISSs through conventional compilation, and its deployment to hardware via HLS. Our main motivation is to reduce the required time for processor core design, especially in the context of implementing custom instructions for acceleration. With these goals in mind, we plan on further exploring the usage of class inheritance mechanisms and other features presented by C++ and supported by Vitis HLS, and leveraging their advantages for SoC development.

During our literature review, we have identified that HLS based approaches for processor design are under-explored, but that existing works, which focus entirely on the RISC-V ISA, produce competitive results versus conventional hardware design. The underlying HLS limitations related to datapath complexity are still a challenge for performance scaling in this type of flows, and there have been recent efforts focused on improving HLS workflows to address these issues for processor design [18]–[20]. However, we were capable of demonstrating that, by removing some control from the tool and manually solving dependencies through high-level specification, an improvement in throughput is possible, as well as having better control over the synthesized pipeline. As a result, we achieved comparable synthesis results for resource usage and a higher throughput than other HLS implementations.

Finally, on-going work on RISC++ will aim to address two limitations identified in the state-of-the-art. Namely, only one approach presented the integration of the core to a small test SoC, and no approaches considered equipping the core with existing interfaces for external hardware accelerators, including support for multiple clock domains [21]–[23].

REFERENCES

- [1] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.
- [2] A. Takach, "High-Level Synthesis: Status, Trends, and Future Directions," *IEEE Design & Test*, vol. 33, pp. 116–124, 2016.
- [3] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.
- [4] "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2016-17, April 2016. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>
- [5] Clifford Wolf, "PicoRV32: A Size-Optimized RISC-V CPU Core," 2019. [Online]. Available: <https://github.com/YosysHQ/picorv32>
- [6] O. Toker, "A High-Level Synthesis Approach for a RISC-V RV32I-Based System on Chip and Its FPGA Implementation," *Engineering Proceedings*, vol. 58, p. 72, 11 2023.
- [7] P. Mantovani, R. Margelli, D. Giri, and L. P. Carloni, "HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis," 2020, pp. 1–8.
- [8] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flammann, and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.
- [9] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984. [Online]. Available: <https://doi.org/10.1145/358274.358283>
- [10] B. Goossens, *Guide to Computer Processor Architecture*. Springer, 2023.
- [11] "Spike RISC-V ISA Simulator." [Online]. Available: <https://github.com/riscv-software-src/riscv-isa-sim>
- [12] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020.
- [13] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USENIX Association, 2005, p. 41.
- [14] T. Alonso, G. Sutter, S. López-Buedo, and J. E. López De Vergara, "Enhancing conditional stalling to boost performance of stream-processing logic with RAW dependencies," 2023. [Online]. Available: <http://www.ieee.org/publications>
- [15] RISC-V International, "RISC-V Tests: A Test Suite for RISC-V Compliance," 2024. [Online]. Available: <https://github.com/riscv-software-src/riscv-tests>
- [16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.
- [17] M. J. Reisinger, "PolyBench/C benchmark suite (version 4.2.1 beta)." [Online]. Available: <https://github.com/MatthiasReisinger/PolyBenchC-4.2.1>
- [18] G. Liu, J. Primmer, and Z. Zhang, "Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications," in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.
- [19] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis," *IEEE Micro*, vol. 42, no. 5, pp. 99–107, 2022.
- [20] J.-M. Gorius, S. Rokicki, and S. D., "Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis," in *International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–6.

- [21] S. Machetti, P. D. Schiavone, T. C. Müller, M. Peón-Quirós, and D. Atienza, “X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.05548>
- [22] M. Damian, J. Oppermann, C. Spang, and A. Koch, “Scaie-v: an open-source scalable interface for isa extensions for risc-v processors,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, p. 169–174.
- [23] B. Green, D. Todd, J. C. Calhoun, and M. C. Smith, “TIGRA: A Tightly Integrated Generic RISC-V Accelerator Interface,” in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 779–782.