HAL
open science

Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis

Jean-Michel Gorius, Simon Rokicki, Steven Derrien

To cite this version:

Jean-Michel Gorius, Simon Rokicki, Steven Derrien. Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis. FPT 2022 - International Conference on Field Programmable Technology, Dec 2022, Honk Kong / Hybrid, Hong Kong SAR China. pp.1-6, 10.1109/ICFPT56656.2022.9974478 . hal-03828841

HAL Id: hal-03828841

https://inria.hal.science/hal-03828841v1

Submitted on 25 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis

Jean-Michel Gorius*, Simon Rokicki*, Steven Derrien*
*Univ Rennes, Inria, CNRS, IRISA

***Abstract*—The RISC-V ecosystem is quickly growing and has gained a lot of traction in the FPGA community, as it permits free customization of both ISA and micro-architectural features. However, the design of the corresponding micro-architecture is costly and error-prone. We address this issue by providing a flow capable of automatically synthesizing pipelined micro-architectures directly from an Instruction Set Simulator in C/C++. Our flow is based on HLS technology and bridges part of the gap between Instruction Set Processor design flows and High-Level Synthesis tools by taking advantage of speculative loop pipelining. Our results show that our flow is general enough to support a variety of ISA and micro-architectural extensions, and is capable of producing circuits that are competitive with manually designed cores.**

I. Introduction

The RISC-V ecosystem is quickly growing and has gained a lot of traction in the FPGA community, as it permits free customization of both the ISA and the micro-architecture.

Retargeting a compiler to a new ISA is a widely studied problem, but automatically synthesizing the corresponding instruction set micro-architecture has received less attention. Existing tools and technique offer significant room for improvement: they either lack generality [1], [2] or operate from low-level structural models that are not fundamentally different from RTL specifications.

In the meantime, High-Level Synthesis (HLS) technology, which compiles C and C++ code directly to hardware circuits, has continuously improved. For example, several recent research results have shown how High-Level-Synthesis techniques could be extended to synthesize efficient speculative hardware structures [3], [4]. In particular, Speculative Loop Pipelining (SLP) appears as a promising approach as it can handle both control-flow and memory speculations within a classical HLS framework [5].

In this work, we bridge part of the gap between Instruction Set Processor design flows and High-Level Synthesis tools. We show how to take advantage of SLP to automatically synthesize in-order pipelined micro-architectures from Instruction Set Simulator (ISS) models written in C, focusing on the RISC-V ISA. Our contributions are the following:

- We show how SLP can serve as a foundation to perform fully automatic micro-architectural synthesis from a behavioral description of a processor, in the form of an ISS. We extend SLP to support the synthesis of in-order pipelined CPU micro-architectures and their hazard recovery logic.
- We evaluate our approach in terms of supported features (both from an ISA and micro-architectural perspective) and quality of results (performance and area). Our results show that our flow can handle complex mechanisms like branch prediction and hardware Control-Flow Integrity while providing QoR similar to manual designs.

The remainder of the paper is organized as follows. Section II provides the necessary background, while Section III describes our approach. Section IV provides additional details on the design space exploration capabilities of our speculative HLS toolchain, SLP. We evaluate our work in Section V, discuss related work in Section VI, and conclude in Section VII.

II. Background

In single-issue in-order pipelined CPUs, a new instruction starts its execution before previous instructions are entirely executed. A pipeline hazard occurs when two consecutive instructions are dependent or need the same hardware resource at the same time instant. In such a case, instructions are canceled or stalled until execution can resume. While this type of micro-architecture is well-understood, its implementation and customization using RTL-level specifications are tedious: modifying the pipeline structure often requires a complete rewrite, making Design Space Exploration (DSE) very difficult.

A. Micro-architecture design tools

Several works proposed to ease micro-architecture design, either through Domain-Specific Languages [6], or HDL code generators [2]. However, these approaches

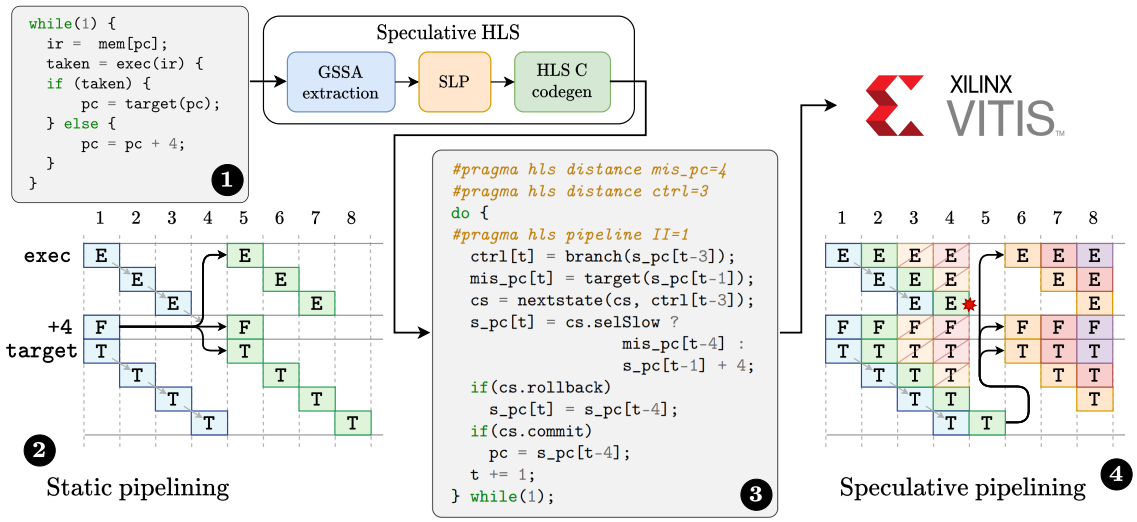

Fig. 1: Our SLP source-to-source transformation flow. The toolchain takes C code ❶ as an input and produces transformed C code ❸. ❷ and ❹ show the respective schedules of the input and output code.

either lack generality or do not raise the level of design abstraction high enough to enable DSE (see Section VI for more details). Ideally, designers should be able to operate directly from a behavioral description of the ISA, e.g., in the form of a C-based ISS description. They would then delegate the pipeline exploration and optimized HDL generation to a HLS tool.

However, state-of-the-art HLS tools fall short of expectations when synthesizing this type of program. Consider the code snippet in Part ❶ of Figure 1, which captures the PC update process within a CPU. Using this kernel as an input to an HLS toolchain such as Vitis HLS leads to a pipelined schedule (shown in ❷) with II=3; in other words, an implementation where a new instruction can be issued at best every three cycles.

The reason is that existing HLS pipelining and scheduling techniques rely on static dependency analysis, which forces the schedule to wait until the `exec` function is evaluated to pick the next value of PC. A similar problem occurs when dealing with the loop-carried dependencies involving the register file content.

B. Speculative High-Level Synthesis

This limitation can be lifted by using Speculative Loop Pipelining (SLP) [3], which produces dynamic pipelined schedules by speculating on the outcome of operations within the loop body. This work builds on an existing source-to-source transformation flow implementing SLP. The resulting flow, depicted in Figure 1, accepts C code as input and produces speculatively pipelined C code targeting an HLS toolchain. The latter can then be compiled/synthesized to obtain an RTL-level description of the processor core.

The key idea in SLP resides in rescheduling critical operations to extend their dependence distance before calling the HLS tool. The HLS static pipelining pass will harness this additional schedule slack to produce more aggressive (i.e., deeper) pipelined schedules with higher clock speeds. The output C code shown in Part ❸ of Figure 1 is produced from the input C code in Part ❶. Its corresponding execution trace is provided in Part ❹.

As shown in ❸, the output code exposes longer reuse distances through delaying elements (arrays `s_PC`, `s_IR`, etc.) and implements additional logic to handle hazards in the pipeline. This logic consists of (i) a rollback command (`cs.rollback`) for restoring past inputs and (ii) a commit command (`cs.commit`) to tag valid outputs. The hazard recovery logic is controlled by a Finite State Machine (FSM) which we do not detail here and whose active state corresponds to variable `cs`.

The reuse distance is chosen by the SLP transformation flow. The larger the distance, the more aggressive the speculation can be. There is hence a trade-off between (i) values of II and T_{cp} and (ii) area and mispeculation recovery cost.

III. Proposed Approach

This section exposes how our toolchain infers a pipelined processor's structure from an Instruction Set Simulator. We show how this ISS model maps to the Gated-SSA (GSSA) representation [7] used by our

toolchain and illustrate transformations used to explore a wide range of micro-architectures.

A. Defining the CPU model

The input program used in the proposed approach is a simple C code implementing an ISS. The processor execution is modeled using an infinite loop that performs instruction fetching, decoding, and execution within a single loop iteration. Therefore, there is a direct correspondence between an iteration of the simulator loop and a complete CPU instruction execution.

In addition, we assume that memory and register files are represented using C arrays, whereas the decoding logic is implemented through a `switch/case` statement.

The Gated-SSA representation of the ISS is shown in Figure 2. It comprises several static loop-carried dependencies (over `pc`, `x[]` and `mem[]`) which prevents the HLS tool from fully pipelining the kernel. The representation also exposes several speculation opportunities through γ-nodes, $\gamma_{\texttt{pc}}$ and $\gamma_{\texttt{out}}$, which we aim to explore.

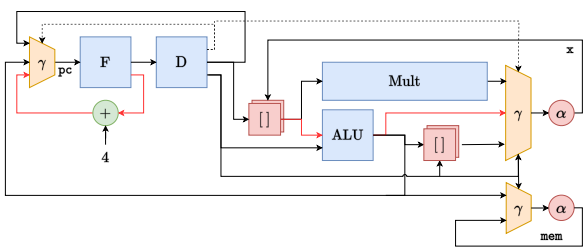

Fig. 2: GSSA representation of the full ISS model, including data memory.

B. Data hazards and memory speculation

SLP can only deal with control-flow hazards and cannot pipeline the execution stages due to RAW dependencies over the register file. From a compiler's perspective, these dependencies are akin to memory dependencies over the array that *models* the register file. This memory dependency can be speculated on using two distinct approaches described in previous work [5] by considering arrays as values and re-casting memory speculation as control-flow speculation. The following sections describe how we can leverage these techniques in our toolchain to handle data hazards in a processor pipeline.

1) Interlocking: The first approach implements pipeline interlocking by introducing runtime alias detection checks in the GSSA representation. These checks control an n-ary γ-node that selects, among p previous array values, the one containing the most recent update of the to-be-read location. This principle is illustrated in Figure 3 for $p = 1$. The modified GSSA representation exposes a speculation path with a reuse distance of 2, thanks to the additional γ-node. Thanks to this transformation, it is possible to explore different pipeline depths for the processor's execution stages.

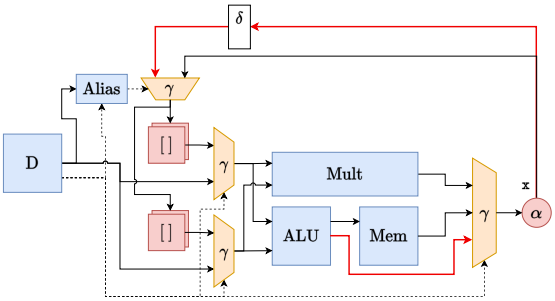

Fig. 3: Pipeline interlocking through alias check insertion. Memory access logic is omitted for the purpose of clarity.

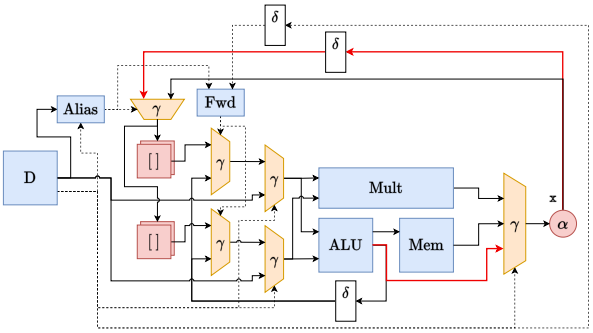

Fig. 4: Pipeline forwarding. Memory access logic is omitted for the purpose of clarity.

2) Forwarding: The second approach implements a form of operand forwarding to bypass the write operation on the array `x[]`.An additional decision layer is added to the hazard management logic, as can be seen in Figure 4. The inserted γ-nodes are controlled by the `Fwd` node and select between the previous execution's ALU result and the register file. If there is a RAW dependency, i.e., an alias, at the current iteration, and if the last iteration used the ALU, the forwarded value is selected. Otherwise, further computations are stalled, and the register file is read to get the correct operand value.

C. Supporting non-pipelined execution unit

All operations within a loop body are pipelined in the speculative HLS framework [3], even if they are not

used very frequently. This aggressive form of pipelining leads to suboptimal utilization of hardware resources. Consider, for example, the M extension of the RISC-V instruction set, which supports integer multiplication and division. Implementing these instructions through our flow will lead to a CPU datapath with a fully pipelined integer division operation. Given such instructions' low utilization rate, this can be considered a poor design decision.

To address this issue, SLP replaces such costly and rarely-used operations with a non-pipelined multi-cycle version of the same operator. A small state machine drives the resulting operation by advancing the computation at each cycle and producing the expected result after several cycles.

IV. Putting it all together

A. Design Space Exploration

Combining memory speculation strategies for the register file with speculations over $\mathtt{pc}$ leads to a large design space, in which the most effective solution is likely to intertwine speculations involving several gamma nodes. Previous work [3], [4] only handles single speculations or speculations that do not interact with one another. These approaches need to be extended to support more complex patterns to synthesize efficient pipelined processor structures. Our ISS model exposes nested speculation patterns with multiple interacting γ-nodes.

We extend the SLP flow to support these patterns and perform a static analysis over the model to determine all the speculation paths leading to a pipeline with $\mathrm{II} = 1$ (i.e., a pipeline that can reach a $\mathrm{CPI} = 1$ when no hazard occurs). We discard the least relevant configurations using profiling information. Changing the latencies of the different operators allows us to explore a wide variety of pipeline structures.

B. Expressivity

As mentioned in earlier sections, the ability to operate from an ISS model expressed in C/C++ offers more flexibility and expressivity than DSL and code generation-based approaches. In the following, we discuss several examples that illustrate this versatility. Our chosen examples address this aspect from two angles: the ability to easily extend the ISA with custom instructions and the ability to customize the micro-architecture directly from the C specification.

1) ISA extensions: We evaluate our approach on the base RV32I ISA and on the RV32IM ISA. We also implement the AES extensions described by the authors of ASSIST [1]. We directly reuse the C code provided by the authors in their paper. Area and performance results for this extension are provided in SectionV and are on par with those reported for ASSIST.

2) Micro-architectural extensions: Our approach also makes it possible to describe micro-architectural features directly at the behavioral level, as our toolchain can infer the corresponding hardware feature. In such a case, the C specification exposes both the target core's architectural and micro-architectural state.

a) Branch prediction: We extend our ISS model with a branch prediction mechanism based on a bi-modal predictor built on a 256-input Branch Target Buffer. This predictor is expressed directly at the ISS level, using less than 20 lines of C code. This model leads to a GSSA representation with an additional short speculation path, which is frequently taken as the prediction is expected to be correct most of the time. Our tool can then harness this new speculation path and reduce the number of control-flow hazards.

b) Control-Flow Integrity: We also add a Control-Flow Integrity security mechanism to protect the processor against buffer overflow attacks. The former consists of a shadow stack implemented in a separate on-chip memory. This shadow stack works as follows: whenever a call instruction is executed, the processor pushes the return address to the shadow stack. When returning from the call, the processor checks that the target jump address matches the top of the stack. If not, an exception is raised. As for branch prediction, this mechanism is entirely described at the ISS level. From a micro-architectural perspective, the CFI mechanism adds a long path in the PC update loop, although that path should rarely be taken. This path leads to additional pipeline stages to preserve clock speed. Here again, our flow can infer the modified pipelined structure automatically.

V. Experimental validation

To demonstrate that our proposed approach can generate competitive pipelined micro-architectures, we generate a large set of processors by exploring different speculation setups: no speculation on the register file, pipeline interlocking, or forwarding. We also modify the latency of the different operational blocks used in the ISS to explore several different pipeline depths. As baselines, we also synthesize the three Sodor pipelined cores (2-, 3-, and 5-stage pipelines) and the CV32E40P core [8]. We synthesize two configurations of the Comet processor [9], RV32I and RV32IM. The latter core is entirely designed in C++ using HLS. It is, therefore, a good baseline to understand the level of performance that can be expected from a manually optimized low-level

pipeline description in HLS. As our generated cores do not implement RISC-V CSR registers, we remove the CSR unit from the Sodor and CVE4 cores.

Our experiments target an Artix7 XC7A200TISBG-1L and use Vitis HLS 2021.2 as the HLS backend. Performance results were obtained by executing the Dhrystone benchmark, compiled using `newlibc`.

Results of the automatic design space exploration are provided in Figure 5. The leftmost part represents the results obtained for the RV32I ISA, and the rightmost part represents the results obtained with the RV32IM ISA. The generated micro-architectures are slower than the Sodor and Comet baselines for the RV32I ISA, while we are able to generate faster cores for the RV32IM target (36% faster than CVE4). Our generic approach generates extra control logic on the critical path of the RV32I cores, reducing their maximal frequency. On the other hand, the critical path of RV32IM cores is located in the multiplication/division unit. The extra logic that hinders the RV32I performance could be optimized during the SLP transformation, but this improvement is left for future work.

Regarding hardware cost, our exploration highlights smaller designs than the baselines for both RV32I and RV32IM ISAs. The discovery of such points is made possible by the design space exploration capabilities of our toolchain and by the ability of the HLS backend to infer efficient memory primitives for the register file.

The cores generated by our flow, numbered from 1 to 8, are labeled in Figure 5. Cores 1, 2, 4, and 5, which are the fastest ones, implement the forward mechanism described in Section III-B2. Cores 6, 7, and 8, which are the smallest cores, use the interlocking mechanism described in Section III-B1. Core 3, which is the core with the highest DMIPS/MHz, does not implement any memory speculation mechanism. This last result demonstrates that, even though the absence of speculation on memory aliases increases the performance score of the micro-architecture, the impact on the maximal frequency might counterbalance the benefits.

We observe that core 2 is much smaller than core 1 while having a similar performance level. In the former, the HLS backend instantiates a LUTRAM memory for the register file, increasing the pipeline's depth and the maximal frequency and significantly reducing the number of FF used.

Comparing against the different baselines highlights that we generate cores using fewer LUTs but similar amounts of FFs. We also note that the performance improvement against baseline cores for the RV32I ISA is due to the increased maximal frequency. Indeed, our approach can balance the pipeline organization to match the target timings.

VI. Related work

This section discusses the benefit of our flow w.r.t other existing approaches, both from a quantitative and qualitative point of view.

A. Synthesizing micro-architectures from DSLs

Most of the work on this topic has focused on easing design through the use of Domain-Specific Languages (e.g., LisaTek [6]). These languages provide an abstraction layer encompassing the definition of both the instruction set and the pipeline structure. Although they drastically simplify design and validation, they still operate at a structural level. In particular, the pipeline and the hazard detection logic must be described explicitly by the designer. These frameworks are convenient for customizing an existing processor through additional instructions but do not permit design space exploration.

B. Micro-architecture synthesis tools

Intermediate approaches have also been proposed [10], [11], where the user partially defines the micro-architecture by placing pipeline stage registers. The tools can then infer the corresponding hazard recovery logic. Kam et al. [12] employ elastic transformations to convert a simple low-level CPU description into a pipelined processor design.

The closest work to ours is that of Liu et al. [1], which introduces the ASSIST design flow, a tool capable of performing DSE from ISA specifications. ASSIST relies on an abstract model of a RISC-V processor pipeline to perform DSE. Although it can explore micro-architectures with varying pipeline depth, ASSIST only considers a predefined pipeline organization and relies on a fixed set of operations that carry domain-specific semantics. This fixed organization limits the achievable expressivity of the designs. For example, ASSIST uses specific operations for PC management and does not support complex operations such as division. In contrast, our flow does not require domain-specific knowledge, e.g., `pc` is a variable just like any other.

C. Pipelined CPU models for HLS

Several recent works reportedly used High-Level Synthesis to design pipelined processors, mainly targeting the RISC-V ISA [9], [13]. These attempts expose the pipeline structure and hazard detection logic within the input C code. In that respect, they do not fully utilize HLS capabilities, nor do they raise the level of

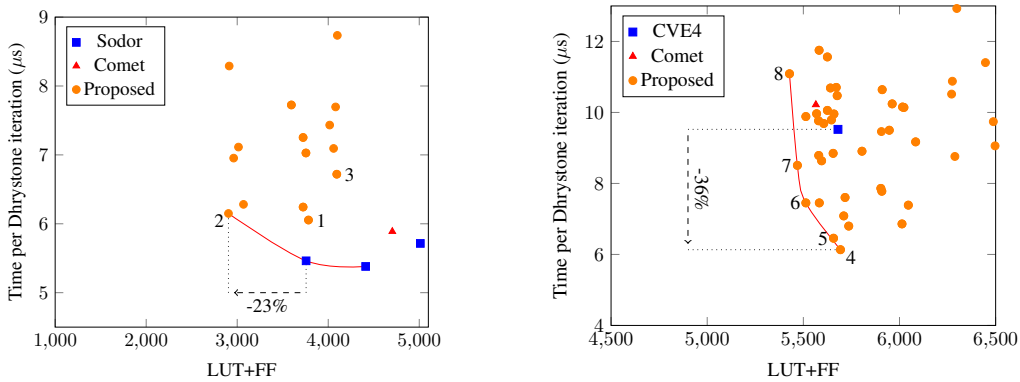

Fig. 5: Area/performance result set for 21 variants of RV32I core (left) and 105 variants of RV32IM core (right) synthesized through our approach. Results for Sodor, Comet and CVE4 are reported on the figure as baselines.

abstraction compared to RTL. Moreover, automatic DSE is impossible, as any pipeline structure changes involve profound modifications to the synthesizable C model.

D. Speculative hardware synthesis

HLS tools struggle to deal with data-dependent control flow and memory accesses. This issue is an active research topic in the HLS community, and several techniques have been proposed. Some authors have focused on dynamic scheduling [14], [15], while a smaller body of work has explored speculative scheduling [3], [4]. The latter serves as a foundation for this work.

VII. Conclusion

In this work, we show how it is possible to apply High-Level Synthesis loop pipelining techniques to the synthesis of RISC-V in-order pipelined processor microarchitectures directly from an Instruction Set Simulator written in C. The proposed approach builds on Speculative Loop Pipelining techniques and is general enough to support complex ISA and microarchitectural features, while achieving performance levels similar to RTL designs. Datapath merging to increase resource-sharing in the synthesized designs and additional control logic optimizations are left as future work.

References

[1] G. Liu, J. Primmer, and Z. Zhang, "Rapid generation of high-quality RISC-V processors from functional instruction set specifications," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[2] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, 2005, pp. 202–212.

[3] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward Speculative Loop Pipelining for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4229–4239, 2020.

[4] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative Dataflow Circuits," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–171.

[5] 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.

[6] *LISATek Processor Designer Manual*, CoWare.

[7] P. Tu and D. Padua, "Efficient building and placing of gating functions," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995, pp. 47–55.

[8] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *3rd RISCV Workshop*, 2016.

[9] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys, "What You Simulate Is What You Synthesize: Design of a RISC-V Core from C++ Specifications," in *RISC-V Workshop 2019*, 2019, pp. 1–2.

[10] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu, "Automatic pipelining from transactional datapath specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 441–454, 2011.

[11] R. Dreesen, "Generating interlocked instruction pipelines from specifications of instruction sets," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 285–294.

[12] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 434–441.

[13] P. Mantovani, R. Margelli, D. Giri, and L. P. Carloni, "HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2020, pp. 1–8.

[14] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, 2017, pp. 189–194.

[15] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, 2013.