Development and Verification of a Dynamically Scheduled Speculative Execution RISC-V 32I Core

Caleb Gannon
Department of Electrical & Computer Engineering
McMaster University
Hamilton, Canada
gannonc@mcmaster.ca

***Abstract*— This paper analyzes implementations of the 32I RISC-V ISA (2019 Unprivileged Spec), and the principles of pre-silicon and post-silicon verification using test program automation. The design HDL is Verilog 2001. The archetecture analyzed is an out-of-order speculative execution core, implementing Tomasulo's algorithm. Functional and structural verification workflows are developed in tandem to the design. The benefits of the open-source RISC-V tool flow are analyzed across variants of RISC-V microarchitectures.**

Keywords—RISC-V Core, Dynamic Scheduling, Speculative Execution, out-of-order, Tomasulo, Pre-Silicon, Post-Silicon, Verilog, Verification, RISC-V GNU Compiler Toolchain, Verilator, gtkwave

I. INTRODUCTION

Computer architecture projects with complex and custom designs are traditionally challenging. With mainstream commercial instruction sets, there are an array of barriers that prevent the rapid development of a novel design. The requirements to begin development are gatekept by a litany of legal and technical requirements, such as "diverse array of electronic design automation tools for simulation, synthesis, place-and route, analysis" alongside required "intellectual property (IP) blocks (e.g., standard-cell libraries, I/O cell libraries, memory generators)" [1]. The restrictive nature of commercial licencing agreements limits the ability of engineers to develop novel systems, as commercial instruction set extensions are only available based on the licensing restrictions negotiated by the project engineers. These challenges could be overcome through the development of custom instruction sets, but these solutions require the development of custom compilers, assemblers, and libraries which are tedious and complex, often extending the project timeline by "months or years" [1].

An alternative solution is to use an ISA based on principles such as open-source and modularity. Such an ISA enables the agile development of a computer architecture project by leveraging a suite of community-based development tools, which have no legal dependencies, and an ISA that is extendable and modular, enabling custom architectural solutions. The RISC-V general-purpose ISA upholds these principals. Industry standard implementations of microarchitecture may typically see "a dozen or more cores with different software stacks (e.g., an ARM application CPU, a GPU, three to five different DSPs, and a power-management core)" [2]. The RISC-V ISA targets the development of a generic ISA that could theoretically be used to implement custom designs across any of the above architectural spaces. There are also benefits in terms of the simplicity of the base RISC-V extensions. A minimal RISC-V core is "roughly half the size of equivalent ARM cores" [2]. Through its efficient but extendable nature, he RISC-V ISA demonstrates potential for use both in research and in the semiconductor industry.

The use of systematic verification in the development of RISC-V designs is critical, but faces the same challenges in RTL verification as any commercial ISA, as "microarchitectural software simulation [is] the bottleneck of the recent hardware/software co-design cycles" [3]. Given the extendable nature of the RISC-V ISA, the challenges of verifying the full design space of a complex implementation, across both pre- and post-silicon environments, are overarchingly complex. One technique is the use of FPGA based simulation accelerators, to drive a high volume of cycle-accurate transactions at multiple orders of magnitude higher simulation speeds relative to CAD software-based simulation [3]. To implement FPGA-Accelerated RTL Simulation, the RTL design is mapped to an FPGA and I/O targets are implemented as "hardware widgets that translate timing tokens to high-level transactions" [3]. The periodic results of test program transactions can be monitored, while the overall system can be driven at simulation rates approaching 3.56 MHz; "up to three orders-of-order-magnitude faster than cycle-level microarchitectural software simulators" [3]. The enormous speedup offered by FPGA-Accelerated RTL Simulation coupled with the potential for simulation inaccuracies in cycle-level software simulators demonstrate the benefits of enhanced verification techniques applicable to the RISC-V ISA as design complexity increases.

The pre-silicon and post-silicon verification environments offer unique benefits and challenges for bug detection, and test program implementation. There are benefits to synergizing the two domains with a shared framework, test plan languages, and structure. A shared-language test program generator suite used by IBM targets test program generation concurrency across the pre- and post-silicon domains. Genesis-Pro is a pre-silicon test program generator, and the companion generator *Threadmill* operates using similar test-templates [4]. Post-silicon test templates can be harvested from the pre-silicon environment in order to target coverage closure, by leveraging the high cycle-count of post-silicon testing, "which allows execution of a very large number of test-cases from the harvested test templates" [4]. The use of a shared test template language allowed verification teams to operate on both the pre-and post silicon domains concurrently, using "exercisers on accelerators" to simulate a post-silicon environment during design bring up, which promoted bug-recreation and the creation of post-silicon coverage events [4]. IBM was able to achieve 100% coverage closure in post-silicon by leveraging their technologies. The effects of this philosophy are subtle, but they demonstrate the need to consider post-silicon concerns throughout the entirety of the pre-silicon development phase.

The development of a RISC-V 32I core was enhanced by the availability of open-source compilers, assemblers, linkers and the plethora of shared knowledge provided by the RISC-V community. The benefits are not specific to RISC-V, as opensource instruction sets have been attempted in the past, such as the OpenSPARC ISA [5]. The success of the RISC-V ecosystem is attributed to the range of industry and academic collaborators who have "[ratified] the RISC-V specifications and [built] the RISC-V community" [5]. The success of RISC-V has left it with a strong set of open-source development tools, such as compilers, assemblers, and synthesis tools [6]. Work has been done to develop High-Level synthesis tools that take advantage of the open-source RISC-V ISA [6]. There have been VLIW core projects that are able to exercise performance across a range of RISC-V subsets, by taking advantage of the existing RISC-V GNU Toolchain [7] and extending it by "[integrating] a dynamic instruction scheduler to overcome the shortage of a specific RISC-V VLIW compiler for the proposed VLIW architecture" [8]. These examples demonstrate the scalability and modularity of the existing open-source tools, and why the strong academic and industrial background can so successfully enable the continued growth of the RISC-V ISA.

II. The Statically Scheduled Core Design

A. *RISC-V Ecosystem*

1) *A Free and Open-Source ISA*

The RISC-V GNU Toolchain [7] was used extensively in the implementation of the RV32I core. The key components used were the compiler, linker, and de-assembler. The toolchain has the ability to support additional architectures such as "rv32i or rv64i plus standard extensions (a)tomics, (m)ultiplication and division, (f)loat, (d)ouble, or (g)eneral for MAFD" [7]. The project targets RV32I exclusively, but was developed in a parameterizable RTL style, targeting the rapid adoption of additional RISC-V modules [9]. This flexibility offered by the RISC-V GNU Toolchain is critical in the generation of executable test programs used to develop the design.

2) *A Simplified Instruction Set*

The simplified nature of the RISC-V instruction set provides a major advantage in agile development. The RTL size requirements of a basic RISC-V core are half of equivalent commercial ISA implementations [2]. The constrained instruction size alongside ISA considerations such as the *R, I, S, B, U, J* instruction formatting [10] enable a low-complexity implementation of the decoder, due to the static placement of source and destination registers within the instruction encoding [10]. While future iterations of the core may deviate and implement modules that further increase the complexity of existing modules, the RISC-V spec naturally lends itself to iteration in this way [1].

B. *Implementation Details*

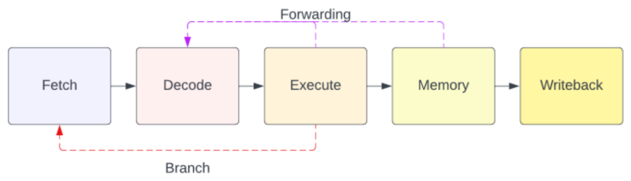


Figure 1: Initial Pipeline Flow [9]

The first iteration of the RV32I core targeted a basic 5-stage statically scheduled pipeline. The design was brought up in the free-licensed edition of Intel® Quartus® Prime [11] in tandem with the free-licensed edition of Questa*-Intel® FPGA Edition [12]. The design was initially developed in a single top-level module which held the five pipeline stages, alongside two lower-level modules which contained embedded data memory and instruction memory caches.

1) *Fetch*

The fetch module interfaced with the instruction memory. The 32-bit instructions returned from the SRAM would be latched in a register that was accessible to the later pipeline stages [9]. Inputs top the module consisted of branching controls, alongside stall controls for hazard resolution.

2) *Decode*

The decode module contains a large combinational block that determines the instruction type and operands. The embedded register file can be accessed. The parsed instruction type and operands are latched for later stages. The decode module contained hazard detection logic, using a shift register to track the location of previous instruction writes [13]. The fetch module is stalled upon detection of "data hazards, control hazards, and structural (resource) hazards" [14].

3) *Execution*

The execution module would read the latched operands and instruction type and compute the operation [10]. A shared ALU was used for both data memory addressing and arithmetic operations. The execute unit was also responsible for storing the PC counter return address, in the case of JALR operations [10]. In the case of memory access operations, the calculated address was latched to be accessed by the memory stage.

Otherwise, the memory stage was bypassed, and the result was latched to be written back to the register file in the following cycle [9]. Branch instructions were resolved in the execute stage, and the output address from the ALU was forwarded to the fetch module. Upon a failed branch prediction, a flag disables the writeback module and flushes from the pipe [9].

4) *Memory Access*

In the case of a memory access instruction, the memory stage was used to interface with the embedded data cache. A combinational logic block would interpret the type of memory access, and the latched address from the execute stage would be used to access the data memory [9].

5) *Writeback*

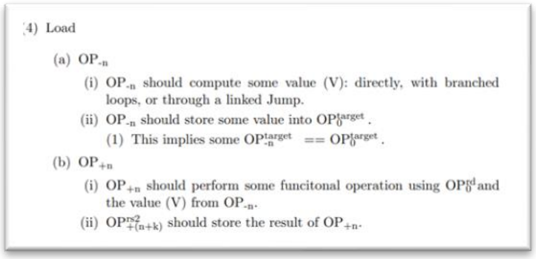
This stage takes the latched result from the execute or memory stage and writes it back into the register file This stage can be bypassed to avoid data hazards [14], and is also bypassed for *store* instructions [15].

III. Verification

A. *Long-Term Verification Planning*

Verification of hardware design is critical to achieve consistent and confident performance across a full range of legal inputs and testable programs [16]. Directed and design-based validation alone cannot provide the same level of coverage and confidence in the performance of a design as that which can be achieved through systematic verification processes [16]. Even with modern functional verification techniques, “up to 80% of designs require a respin due to a functional bug” [16]. The ability of scalable and randomized verification processes to hit corner cases and prove-out a high coverage of design behavior is a benefit to any stage of the design process, and results in more robust and spec-compliant designs [17].

A high-level verification plan which addresses pre-silicon and post-silicon concerns [4] [18]. The plan is derived from a Genesys-Pro random test generation solution implemented by IBM in the verification of their POWER7 processors [19] (Figure 6). A large challenge often faced in differentiating pre- and post-silicon test plans is the usage of differing languages and constructs in the description and implementation of the test plans [4]. To address this, the test plan documentation used a formalized mathematical language to define the test plan sequences based on the relationships between sequences of instructions which would trigger situations in both pre- and post-silicon environments that would verify the functionality of a specific opcode family [4]


(4) Load

(a) OP_{-n}

(i) OP_{-n} should compute some value (V): directly, with branched loops, or through a linked Jump.

(ii) OP_{-n} should store some value into OP_0^{target}.

(1) This implies some $OP_{-n}^{target} == OP_0^{target}$.

(b) OP_{+n}

(i) OP_{+n} should perform some funcitonal operation using OP_0^{rd} and the value (V) from OP_{-n}.

(ii) $OP_{+(n+k)}^{rs2}$ should store the result of OP_{+n}.

Figure 2: An example of a Mathematical Test Plan Descriptor

Mathematical test plan sequences (Figure 2) address each opcode family, providing high-level descriptors from which specific test plans could be generated. These descriptors model the Genesys-Pro random test generation philosophy presented by IBM [19]. Test plans can be derived from the “Test Plan Descriptors” by passing them through a CSP (constraint satisfaction problem) engine against a working model of the design [19]. This proposed workflow enables the dynamic generation of test programs that consider the RISC-V ISA implementation alongside architecture specific details. The workflow would generate a series of RISC-V instructions that stress the design in both *structural* and *functional* aspects [18].

B. *Interim Verification*

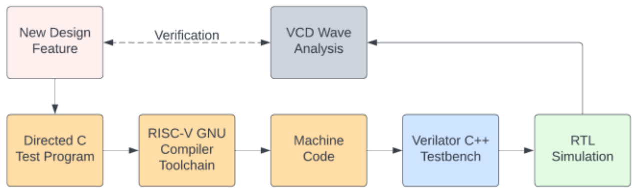


Figure 3: Interim Verification Flow [7] [20] [21]

To enable the agile development of the RV32I core, a series of directed tests were developed to stress the pipeline instruction flow, as well as the memory and execution units. These tests were written in C and compiled into RISC-V machine code through the use of the open-source RISC-V GNU Compiler Toolchain [7]. An active community of contributors maintains this toolchain, demonstrating the project benefits that are attributed to the shared knowledge and development efforts achieved by the RISC-V community. The C-code-based directed testcases are parsed by the RISC-V GNU Compiler Toolchain [7], compiled, and linked into an executable file from which the test program machine code can be extracted. The machine code is programmed into the Instruction Memory using a Verilator C++ testbench [21], and upon simulation run the resets are dropped and a clock is driven, allowing the design to execute the test case. The results are dumped to a VCD wave file, and the open-source tool *gtkwave* [20] is used to analyze the results.

IV. The Out-Of-Order-Execution Design

A. *Implementation Details*

The enhanced performance target design was a 6-stage pipeline featuring out-of-order and speculative execution, with dynamic scheduling, and a late commit, to improve execution times and guarantee precise exception handling [13]. The existing 5-stage pipeline was reconfigured to move the existing stages into Verilog modules, improving code readability and maintainability, also enabling the future development of *block-level* verification plans [18]. The original pipeline stages were updated; one pipeline stage was removed, and two new stages were added. Several design structures were added in order to support dynamic scheduling, out-of-order execution, speculative execution, and the late commit functionality [13]. These new structures include per-execution-unit Issue Queues

(Reservation Stations), a Scoreboard (which tracks pending register file writes), and a Reorder Buffer with a secondary architectural register file, enabling the restore of architectural state of the CPU upon a failed branch prediction or an interrupt or user trap (late commit) [13].

B. *New Pipeline Stages*

1) *Issue*

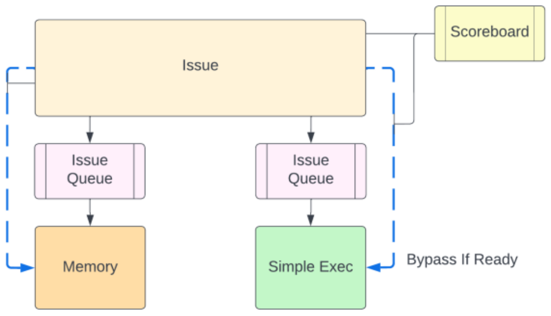


Figure 4: Scoreboard and Issue Queues [9]

The issue stage prepares the instruction for execution, by identifying whether it can begin execution immediately or if it needs to queue [9]. This stage interprets the state of the scoreboard in order to determine whether there are any RAW hazards [14] that require the instruction to queue before execution. If the instruction needs to queue it will be sent to a Reservation Station along with a *Tag* indicating from which functional unit and source register it should wait for data-forwarding-feedback-loop writeback before beginning execution [13]. If the instruction is ready to execute immediately and the execution unit is free, the issue stage will bypass the Reservation Station and latch the instruction, to begin execution in the next cycle [9]. The issue stage contains hazard detection logic, which interprets the capacity and state of the queue structures (ROB, IQ) to detect structural hazards and stall the Decode and Fetch units [14].

2) *Simple Exec*

The simple exec unit reads an instruction from the Issue Queue (or one forwarded from the Issue stage) and passes the operands to an ALU [9]. The simple execution unit resolves branch instructions. This is done using a dedicated simple comparator, and the result is immediately (within the same cycle) forwarded to the fetch unit, to minimize the delay between the issue and resolution of a branch instruction[1] [13].

3) *Memory Control*

The memory control unit interfaces with the embedded data memory and performs load/store operations [10]. It uses shift registers to store the writeback address for a given memory access in the case of sequential load operations. Unlike the original 5-stage design, the out-of-order core has parallelized the memory and simple-exec units, assigning each an issue queue [9]. This decision enables (future) superscalar execution [13].

4) *Commit*

The commit stage interfaces with the physical and architectural register files (PRF, ARF), as well as the Reorder Buffer [13]. This stage will update the PRF upon each pipeline-feedback-loop writeback. The commit stage interfaces with the issue stage, and the Reorder Buffer [9]. The commit stage has an input to indicate a failed branch prediction, and it will restore the PRF to the ARF state. This *restore* operation enables precise interrupt and exception handling, as the ARF will always hold the register file state based on the *real* instruction order [13].

C. *Out-Of-Order Components*

1) *Scoreboard*

The scoreboard operates as a series of shift registers and combinational logic which track the state of instructions pending a writeback to a destination register [9]. The scoreboard implements a shift register on a per-register-file-entry basis. An entry in the scoreboard is populated when an instruction issues that will write to a given register file entry. The scoreboard will hold the entry in a *pending* state until the instruction begins execution (the instruction may wait in an issue queue) [13]. When the instruction begins execution, the scoreboard begins to shift a tracking bit through the registers. This process allows the issue stage to know whether a data hazard exists [14]. When the tracking bit reaches the end of the shift register, the scoreboard entry is cleared once a corresponding data-forwarding-feedback-loop writeback is detected [13]. The issue stage checks the scoreboard *pending* flag to detect whether it should issue an instruction with a register file source value, or a data-forwarding-feedback-loop writeback *tag*.

2) *Issue Queues*

The issue queues are instantiated on a per-execution-unit basis, as a predisposition for superscalar operation [13]. The issue queue entries store instruction operands, tags, PC addresses (for JALR instruction), and have flags to indicate validity of operands, as well as whether the instruction is speculative, and pending a branch resolution [13]. Verilog parameters are used to set the depth of the issue queue buffer [22]. This enables design scaling, allowing designers to utilize available hardware resources efficiently and with flexibility. A combinational logic block interfaces with all entries in the issue queue, in order to identify whether a data-forwarding-feedback-loop writeback can populate any *tag* entries for pending instructions [13]. This implementation does lead to a potential for Write-after-Write hazards, as multiple queued instructions that are pending a data-forwarding-feedback-loop writeback on a given register will all receive the same writeback, regardless of their issue order [13]. A combinational logic block detects the oldest instruction in the issue queue that has a full set of valid operands, and then proceeds to latch and issue that instruction to the execution unit (if the unit is ready) [9].

[1] This implementation could be latched in case it creates a critical path in the design, however it would increase the count of speculative issue branch delay slots.

3) *Reorder Buffer*

The Reorder Buffer (ROB) implements a FIFO queue structure, enabling the late-commit of instruction execution results to the architectural register file (ARF) in an in-order fashion [13]. This enables a precise register-file *restore* in the case of failed branch prediction or system-interrupts. The commit pushes data from the physical register file (PRF) into the ARF when the head of the FIFO has seen a writeback [13]. Speculative ROB entries will not be committed to the ARF until the respective branch prediction has been resolved [9]. There is potential for branching hazards to exist in the current implementation, as branch instructions have the potential to execute out-of-order[2] [13].

4) *The Pipeline-Feedback-Loop*

The pipeline-feedback-loop is a set of connections which span several pipeline stages, forwarding execution stage writeback data [9]. A control unit for the pipeline-feedback-loop lives in the top module of the design, and co-ordinates writeback accesses between execution units [9]. The pipeline-feedback-loop control unit listens for writeback access requests and resolves multiple writeback requests by applying a negative priority to the execution unit which wrote back last [9]. The pipeline-feedback-loop control unit hands pipeline-feedback-loop access to the execution units by signaling a pipeline-feedback-loop access ACK to the corresponding execution unit[3].

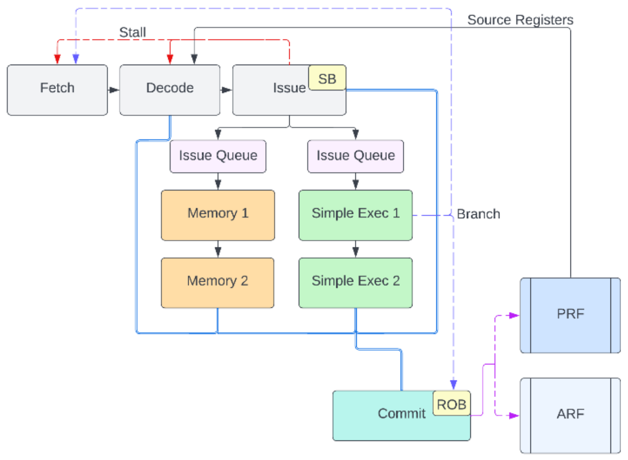


Figure 5: Out-Of-Order Core: Electrical Diagram [9]

V. Anticipated Findings

A. *Scalable Model-Based Test Program Generator*

With two Verilog design implementations of the RV32I Core, a model based test program generator would be able to benefit from test plans that implement Mathematical Test Plan Descriptors (III.A) in order to generate constrained random stimulus that targets both *structural* and *functional* elements of the two designs [19]. The functional targets would be defined in a test plan sequence and make use of the Test Plan Descriptors in order to stress inter-opcode-familiar behaviors [18] at a functional level, as these behaviors result from the definition of the RISC-V ISA definition [10]. These Test Plan Descriptors leverage pre-silicon and post-silicon concerns by maintaining a consistent language [4]. Synergy is achieved through a shared pre- and post-silicon test program generator. This is leveraged to increase the overall level of confidence in the design [4].

The verification flow can be automated, by passing the functional test program through a CSP engine that references design specific *structural* details when generating the functional test program [19]. This reduces the turnaround time in terms of verification planning and regression, as verification engineers would be able to quickly scale ISA based test plans across a variety of possible design implementations, and achieve levels of high coverage without resorting to specific and directed testing [19]. The verification engine can implement *sequence reduction algorithms*, which would parse the test program instructions for redundancies at the instruction level [23]. This automated approach maintains prominent levels of coverage but improves test program performance; without requiring manual optimization by a verification engineer.

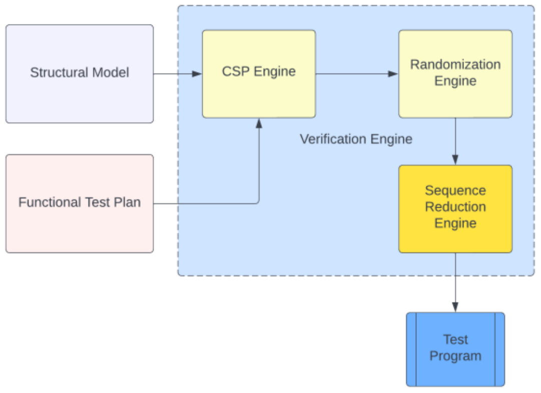


Figure 6: Scalable Test Program Generator [19]

A simulation tool such as Verilator, or Synopsys' VCS® [24] can be used to regress the test programs against the design. This provides elevated levels of design coverage and confidence [16].

VI. Conclusion

The open-source tool flows combined with the industry and academic backbone of the RISC-V ISA facilitate rapid development and verification of complex computer architecture implementation. The modular nature of the ISA lends itself to scalability, coupled with parameterization of the Verilog HDL language this enables long-term design planning. The open-source community contributes a range of verification and design tools that eliminate the need for time-consuming licensing negotiations. Modern functional verification principles can be applied to RISC-V based designs in order to

[2] This can be resolved by limiting branch instructions to in order execution or using multi-bit speculative flag to co-ordinate branch speculation identification.

[3] The ACK controls a MUX, which connects the different execution result latches to the pipeline-feedback-loop. Therefore, the data-forwarding-feedback-loop writeback occurs in the same cycle as the ACK.

research new techniques for pre- and -post silicon verification synergy, and the automation and optimization of test program generation.

References

[1] T. Ajayi, K. Al-Hawaj, A. Amarnath, S. Dai, S. Davidson, P. Gao, G. Liu, A. Rao, A. Rovinski, N. Sun, C. Torng, L. Vega, B. Veluri, S. Xie, C. Zhao, R. Zhao and Z. Zhang, "Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm," *Department of Electrical Engineering and Computer Science, University of Michigan,*, 2017.

[2] D. Kanter, "RISC-V OFFERS SIMPLE, MODULAR ISA," *The Linley Group,* 2016.

[3] D. Kim, C. Celio, D. Biancolin, J. Bachrach and K. Asanovic, "Evaluation of RISC-V RTL with FPGA-Accelerated Simulation," *University of California, Berkeley,* 2017.

[4] A. Adir, A. Nahir, G. Shurek, A. Ziv, C. Meissner and J. Schumann, "Leveraging Pre-Silicon Verification Resources for the Post-Silicon Validation of the IBM POWER7 Processor," *IBM Research,* 2011.

[5] S. L. Harris, D. Chaver, L. Piñuel, J. Gomez-Perez, M. H. Liaqat, Z. L. Kakakhel, O. Kindgren and R. Owen, "RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education," *31st International Conference on Field-Programmable Logic and Applications,* 2021.

[6] J.-M. Gorius, S. Rokicki and S. Derrien, "Design Exploration of RISC-V Soft-Cores through Speculative High-Level Synthesis," *HAL Open Science,* 2022.

[7] RISC-V GNU Compiler Toolchain, [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain. [Accessed 11 12 2022].

[8] N. M. QUI, C. H. LIN and P. CHEN, "Design and Implementation of a 256-Bit RISC-V-Based Dynamically Scheduled Very Long Instruction Word on FPGA," *IEEE Access,* 2020.

[9] mac-risc-v, "RISCV-RV32I-CPU: RTL Repo," 2022. [Online]. Available: https://github.com/mac-risc-v/RISCV-RV32I-CPU/tree/main/rtl.

[10] A. Waterman, K. Asanovi´c and S. Inc., *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213,* RISC-V Foundation, 2019.

[11] Intel, "Intel® Quartus® Prime Lite Edition Design Software Version 20.1.1 for Windows," Intel®, 22 11 2020. [Online]. Available: https://www.intel.com/content/www/us/en/software-kit/660907/intel-quartus-prime-lite-edition-design-software-version-20-1-1-for-windows.html.

[12] Intel, "Questa*-Intel® FPGA Edition Software," Intel®, [Online]. Available: https://www.intel.ca/content/www/ca/en/software/programmable/quartus-prime/questa-edition.html. [Accessed 11 12 2022].

[13] School of Electrical and Computer Engineering, "T10 Advanced Processors: Out-of-Order Execution," Cornell University, 28 11 2016. [Online]. Available: https://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-T10-ap-ooo.pdf.

[14] A. Sari and I. Butun, "A Highly Scalable Instruction Scheduler Design based on CPU Stall Elimination," *IEEE,* 2021.

[15] S. S. Omran and H. S. Mahmood, "VHDL PROTOTYPING OF A 5-STAGES PIPELINED RISC PROCESSOR FOR EDUCATIONAL PURPOSES," *14TH MIDDLE EASTERN SIMULATION & MODELLING MULTICONFERENCE* , 2014.

[16] A. B. Mehta, ASIC/SoC Functional Design Verification, Los Gatos, California: Springer International Publishing, 2018.

[17] A. Meyer, "Why Functional Verification is Needed," in *Principles of Functional Verification*, Burlington, MA, Elsevier, 2003, p. 2.

[18] Mac-Risc-V, "RISC-V Discovery: Test Cases," [Online]. Available: https://mac-riscv.atlassian.net/wiki/spaces/MR/pages/1638408/RISC-V+Discovery+Test+Cases. [Accessed 11 12 2022].

[19] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov and A. Ziv, "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification," *Functional Verification and Testbench Generation,* 2004.

[20] GTKWave, "Welcome to GTKWave," SourceForge, [Online]. Available: https://gtkwave.sourceforge.net/. [Accessed 12 12 2022].

[21] J. Bennett, "High Performance SoC Modeling with Verilator," EMBECOSM, February 2009. [Online]. Available: https://www.embecosm.com/appnotes/ean6/embecosm-or1k-verilator-tutorial-ean6-issue-1.html.

[22] IEEE, "IEEE Standard Verilog® Hardware Description Language," IEEE Computer Society, 28 September 2001. [Online]. Available: https://inst.eecs.berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf. [Accessed 12 12 2022].

[23] J. Yan, H. Zhou, X. Deng, P. Wang, R. Yan, J. Yan and J. Z. , "Efficient testing of GUI applications by event sequence reduction," *Elselvier,* 2019.

[24] Synopsys, "VCS," Synopsys, [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html. [Accessed 12 12 2022].