RAPID-X Digital ASIC Tapeout

Davi Dantas, Evan Kasky, Youssef Samwel,
Pablo Rodriguez, Timothy Ogg, Nicolas Sayed

Dept. of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

Abstract — **This paper presents from design to place and route of RAPID-X, a four-stage pipelined RISC-V processor core that implements RV32I instruction set. A main motivation to this project is the growing demand for skilled ASIC engineers and experience using Cadence tool chain. A four stage pipeline (Instruction Fetch (IF), Instruction Decode (ID), Execute and Memory/ WriteBack (WB)) comprises the micro architecture that also features hazard detection and data forwarding to prevent pipeline stalls. The design integrates split instruction and data caches generated using OpenRAM to improve memory performance. The team performed rigorous verification including Universal Verification Methodology (UVM) and full-core simulation, that achieved over 85% code coverage across every stage. RIPES simulator was also utilized to validate correct core instruction execution. For physical implementation, the tools Cadence Genus and Innovus were utilized for place and route, and static timing analysis, showing no negative slack at the desired clock frequency. Final layout meets Northrop Grumman requirements that includes 100 Million instructions per second, which proves RAPID-X architecture could be used for fabrication.**

Index Terms — **RISC-V, pipelined architecture, RTL design, verification, ASIC design flow, UVM, logic synthesis.**

I. INTRODUCTION

A. Motivation and Objectives

As computing applications continue to grow more specialized, the demand for Application-Specific Integrated Circuits (ASICs) across industries such as AI, automotive, healthcare, and defense grows proportionally. Unlike general purpose processors, ASICs can deliver optimized performance, and lower power consumption for specific workloads. This trend highlights the need for skilled ASIC engineers and custom chip designers, which is a field that is currently a bottleneck in semiconductor pipeline. This project aims to address these shortages by providing a hands on experience in ASIC development, preparing future engineers to contribute to the rapidly evolving industry. Developing domestic talent in custom hardware design is critical to strength national security, drive innovation, and achieve long term technological independence.

B. Project Scope

Our RISC-V32I core implements a 4 stage pipelined architecture, which consists of the following stages: IF, ID, EX, and MEM/WB. By combining the memory access and writeback operations into one stage, the design achieves as balance between complexity and performance. To handle instruction dependencies properly, we included a hazard detection unit that catches load/use data hazards and adds pipeline stalls when needed. Additionally, a data forward unit was implemented to reduce performance penalties by routing results form later stages back to earlier stages, which minimizes the number of stalls caused by data hazards. The core fully supports the RV32I base integer instruction set, including arithentic operations, load/store instructions, branches, and jumps. This combination of pipeline design and control logic allows for efficient instruction throughput while maintaining compliance with the RISC-V Instruction Set Architecture (ISA).

II. DESIGN METHODOLOGY

A. IC Design Flow

The development of the RAPID-X core followed a structured design flow to ensure functional correctness, and performance. Our design flow closely followed the industry-standard approach, consisting of the following stages: specification, RTL design, verification, logic synthesis, place and route, and signoff. Each phase in the design process goes through iterative refinement to ensure for a effective development cycle.

The project began with a well-defined specification sheet supplied by Northrop Grumman, the sponsor of the project. From the specifications, we began by analyzing the RISC-V32I base instruction set and established a baseline understanding required to build the core. Based on these requirements, we created block-level diagram, which helped guide us through the RTL development.

In the RTL development phase, we developed SystemVerilog modules for each of the stages in our pipeline design. Separating each phase into its own module improved the designs readability and modularity. There is a module for each of the pipeline stages, as well as other integral pieces of the core such as the register file. By using a modular design we were able to efficiently develop the core in parallel as well as allowing the verification team to immediately begin testing.

During RTL development and after its completion, we fully transitioned into the verification phase. We primarily conducted functional verification through UVM-based testbenches and direct core testing. This phase ensured that all modules worked according the the specification before moving on to synthesis.

After verification, the design was synthesized using Cadence Genus. To obtain a gate-level netlist optimized for performance and area, timing constraints and clock definitions were applied. This netlist was then passed through Cadence Innovus for physical implementation. Then place and route involved placing standars cells, routing wires, and intergration OpenRAM SRAM. During this process, we ensured that all the constraints were met across all paths.

The final stage of the flow includes signoff checks and static timing analysis and design rule checks. These checks are used to ensure the design meets manufacturing requirements and that it matches the original RTL.

B. *Toolchain*

We used a variety of tools throughout the design, verification, and physical implementation of the RAPID-X core:

- **Cadence Genus:** Used for RTL synthesis, generating a gate-level netlist from our SystemVerilog design. Timing constraints and cell libraries were applied to optimize for area and performance.
- **Cadence Innovus:** Used to handle place-and-route and final physical layout generation. This included integrating OpenRAM SRAM, synthesis, and routing standard cells to produce the GDSII file.
- **Cadence Xcelium:** Used for RTL simulation and UVM-based functional verification. Xcelium provided waveform traces, assertion checks, and coverage metrics used to check design correctness.
- **Vivado:** Used to validate synthesized netlists on FPGA, allowing early-stage testing of core.
- **OpenRAM:** Generated SRAM blocks for instruction and data caches. The tool produced both behavioral simulation models and physical layouts used in integration.

C. *SPECIFICATION AND ISA COMPLIANCE*

The RISC-V ISA has become increasingly popular in both academia and industry due to its robust ecosystem and open-source nature. It originated at the University of California, Berkley in 2010 and consists of several related versions for the standard that differ in capabilities and word bit-widths. [1] For this project we implemented 32-bit integer implementation of the ISA, known as RV32I. The simpler specification allowed our team to focus on learning the design flow without worrying about unnecessary complexity.

The specification of the base instruction set of RV32I consists of 40 instructions[2] – however, because the project's engineering requirements only called for implement the unprivileged portion of the instruction set, the final instruction count of our core is 38 distinct instructions. The instructions are always 32 bits wide and categorized by their encoding scheme.

D. *MEMORY HIERARCHY AND INTEGRATION*

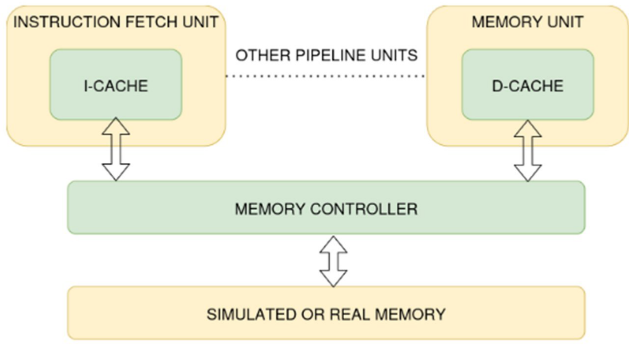


Fig. 1. Harvard memory architecture

The choices leading to the design of the memory system were critical because they needed to comply with both the engineering requirements for performance and capacity, as well as being flexible enough for the testbenches to drive the memory during simulation. To this end, the core implements a split-cache system with 8 kilobytes for the instruction cache and 8 kilobytes for the data cache. Splitting caches simplifies the design of the rest of the pipeline because each cache can be accessed independently without the need for synchronization mechanisms. The instantiated caches then are connected to a potentially larger and slower top-level memory. For simulation purposes, it is this top-level memory that is driven by the testbench. A diagram for the memory architecture major components are displayed in the figure.

The implementation of the cache memory structures initially presented a challenge because specifying memory arrays directly in RTL led to poor performance and density. However, thanks to the input from our sponsor, we utilized OpenRAM to generate the cache memory structures. OpenRAM is a framework to create layouts, netlists, routing, placement, and power models for static RAM for ASIC design. [3] The generated OpenRAM artifacts include a (non-synthesizable) behavioral description in SystemVerilog to use during simulation. The actual SRAM netlist is then utilized during place and route to interconnect the cache subsystems with their corresponding memory blocks throughout the physical design phase.

III. RAPID-X ARCHITECTURE

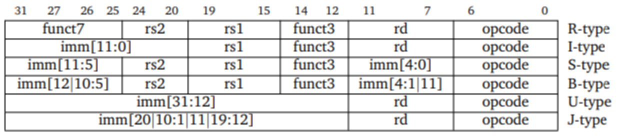
Before discussing the architecture, note that a core design covers instruction execution and I/O integration, while a CPU design also includes peripherals like I/O. The RAPID-X core uses a modified 4-stage RISC pipeline, which boosts throughput by pipelining instructions simultaneously. This allows one instruction per cycle, with speed improvements roughly proportional to the number of stages.

Our RISC pipeline consists of four stages:

- **Instruction Fetch (IF)** – Reads the next instruction from memory.
- **Instruction Decode (ID)** – Parses the current instruction and generates the corresponding control signals.
- **Execute (EX)** – Performs computations using the ALU.
- **Memory Access/Writeback (MEM&WB)** – Memory operations and stores results in the register file.

A. Decoder Unit

The decoder unit is responsible for generating the proper control signals given the instruction. The RISC ISA has 6 instruction types, each with a 7-bit opcode.



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Fig. 2. RISC-V instruction classes.

The opcode may directly specify an instruction (e.g. J/U type) or indicate a family, such as branch instructions which all use the opcode ‘1100011’. To decode further, we consider the 3-bit funct3 field that selects a specific instruction within a group, though some instructions (like add and sub) share both opcode and funct3. In these cases, a flag from the 7-bit funct7 field (with only 1 bit used in RV32I) determines whether the adder performs addition or subtraction.

The decoder converts the 32-bit instruction into a 27-bit control register that guide hardware operations. Each pipeline stage (except for instruction fetch) has its own set of control signals, which shrink as the instruction passes through the pipeline. Initially, the instruction is fetched, then decoded to generate control signals; the execute stage consumes 17 bits of the control register, while the memory/writeback stage uses the remaining 10 bits. Table 1 details the control signals.

B. Sign Extension and Immediate Decoding

Some instructions have an immediate value which can range from 7-bits to 12-bits, however for the values to be used in computations they have to be extended to 32-bit values. When extending the immediate value, we must keep the sign of the value which is typically appending the value with 1’s or 0’s to adhere to two’s complement rules. The immediate value must also be decoded since its not stored in a trivial format as shown in the diagram below,

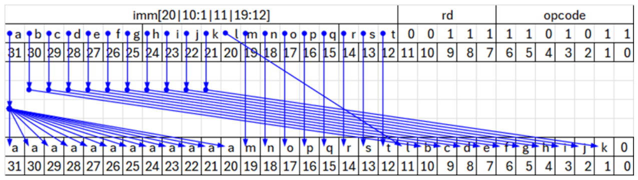


Fig. 3. Decoding of Jump Type Instruction

C. Execute Unit

The execute unit is responsible for branching and arithmetic computations. Based on the control register value, specific hardware blocks would be toggled to perform the desired computation. If branching is to occur, the execute stage creates a reset signal to clear the instruction fetch and decoder stages while also passing the new program counter address to the instruction fetch stage. The reset signal is necessary to avoid control hazards.

D. Pipeline Hazards

Hazards occur when there is a data dependency between instructions. Results of computations or memory loads are stored at the end of the pipeline but if they're needed earlier, it will introduce a data hazard. To resolve this issue, the forwarding unit will redirect the input to the execute stage the 'RS1' or 'RS2' data ports with the latest value of the source registers.

IV. VERIFICATION AND TESTING

Our team implemented a comprehensive verification and testing strategy to ensure the functional correctness and reliability of our RV32I processor design. Verification is a critical phase in the hardware development lifecycle, especially for processor architectures, where the interaction of control logic, datapath components, and instruction decoding must adhere precisely to the specifications of the Instruction Set Architecture (ISA).

Our verification approach was twofold: we utilized both a Universal Verification Methodology (UVM) testbench and a full-core simulation-based testing. UVM provided a robust, reusable, and modular framework for building test environments around our design. It allowed us to create constrained-random stimulus, scoreboards, monitors, and coverage models.

In addition to UVM, we developed a full-core testing using hand-crafted programs to validate the instruction set implementation at the architectural level. These programs included a wide range of test cases, from basic arithmetic and logic instructions to more complex control flow scenarios such as jumps and conditional branches. Our test also incorporated memory read/write tests and exception handling to ensure that the processor responded predictably to out-of-bounds or undefined behavior.

Together, the UVM-based verification and directed testing strategy gave us high confidence in the correctness, completeness, and robustness of our RV32I implementation. This verification effort not only validated that the processor meets the functional requirements of the RISC-V base integer instruction set (RV32I), but also established a strong foundation for future extensions, such as pipelining, cache integration, or RV32IM support. The high confidence comes from the tool XCelium which provides us the code coverage with an aim of 85% or higher

A. Universal Verification Methodology (UVM)

To perform robust functional verification of our RV32I processor, we implemented a Universal Verification Methodology (UVM) testbench. UVM is a class-based standardized SystemVerilog framework that enables scalable, reusable, and constrained random verification environments. The modular nature of UVM allowed us to thoroughly verify our processor design by systematically generating stimuli, monitoring outputs, and checking results against expected behaviors.

Our UVM environment comprised several layered components, and each encapsulated in its own class to promote modularity and reuse. The key building blocks included the **sequence**, **driver**, **monitor**, **scoreboard**, and **environment** classes.

- **Sequences and Stimulus Generation:** The stimulus for our processor was generated using UVM `sequence` classes. These sequences constructed randomized or directed instruction streams. We applied constrained-random techniques to ensure that legal instruction formats were generated according to the RV32I ISA specifications. This included constraints on opcode fields, function codes, and immediate values to create valid instruction patterns.
- **Driver:** The `driver` component received the generated instruction transactions from the sequence and drove them to the input interface of the processor design. The driver ensured that timing and protocol constraints were respected, synchronizing inputs to the design under test (DUT).
- **Monitor:** To capture the behavior of the DUT, we im-

Table 1: Control Signal Allocation Across Pipeline Stages

Stage	**Control Signal Width**	**Description**
Instruction Fetch (IF)	0 bits	Instruction is fetched from memory; no control signals used.
Instruction Decode (ID)	27 bits	Decodes the instruction and generates the full control register.
Execute (EX)	17 bits	Consumes the first 17 bits of the control register for ALU and operation control.
Memory/Writeback (MEM&WB)	10 bits	Uses the remaining 10 bits to manage memory access and register writes.

plemented `monitor` class. This passive component observed the activity of the signal in the processor output. The monitor collected instruction results and execution details, which were used in the scoreboard for comparison.

- **Scoreboard:** The `scoreboard` served to check the correctness of the DUT. It received actual outputs from the monitor and compared them against the expected output values for each instruction given the input state. By maintaining a parallel software model of the architectural state of the processor, the scoreboard could identify any mismatches and report the error.

Throughout the simulation process, detailed log files and waveform traces were used to diagnose any discrepancies between expected and actual outputs. Assertions were embedded to flag protocol violations and unexpected state transitions, accelerating the debugging process. The use of UVM significantly enhanced our ability to test corner cases, uncover latent bugs, and build confidence in the robustness of our RV32I processor design.

V. SYNTHESIS AND PHYSICAL DESIGN

Synthesis and place and route bridges the RTL designed in this project to the real world. While it may seem like these two steps would only be performed at the end of the ASIC design flow, teams often need to loop back to earlier stages to change the RTL or even the initial system level design. After and alongside verification we used Cadence Genus and Innovus for synthesizing the gate level netlist and routing the design respectively.

A. Synthesis

To synthesize a netlist properly Cadence Genus requires a few essential files which include the RTL files, Liberty timing files, and Synopsys Design Constraints files. Within Genus there a few ”knobs” which can be turned depending on what you need to optimize such as timing, area, and power.

- **RTL Files:** Contain the detailed hardware description in SystemVerilog.
- **Liberty Timing Files:** Describe the timing and electrical characteristics of standard cells used during synthesis. This includes the pinouts of each individual cell as well as parameters such as maximum slew, capacitance, and power usage.
- **Synopsys Design Constraints (SDC) Files:** Contain constraints such as clock definitions, input/output timing, and optimization goals. This is where we set parameters such as clock speed and uncertainty that Genus will optimize towards/with.

Between synthesis and place and route there is also the concept of design for test. DFT techniques such as scan insertion and boundary scans ensure that the physical design logic matches the logic of the RTL. Without DFT it would be difficult to verify that an ASIC would be ready to manufacture, and although DFT was not implemented for the FreePDK45 process it was shown that our core could pass DFT tests on other processes such as FreePDK15. The benefits of the scan chain were outweighed by the pros of the FreePDK45 process however, so a gate level simulation was used instead to determine equivalence post place and route.

B. *Place and Route*

Once satisfied with the netlist generated by Genus, the optimized netlist is then used by Innovus to create a physical layout. Innovus uses most of the same files as Genus but with the inclusion of other files such as capTables, QRC, and lef files. Lef files in particular are very important for place and route because these files describe the physical layout of the cells as well as definitions about the dimensions of silicon and metal that make up each individual transistor. Innovus has Tempus and Voltus included within the suite which were used to perform static timing analysis and power simulations throughout the process of place and route.

The physical layout of our core went through multiple stages before being reaching the sign-off stage. This includes the initial route, pre-clock tree synthesis (pre-CTS), post-clock tree synthesis (post-CTS), post-route optimization, and eventually sign off. Each stage has incremental improvements as well as different simulation tests. Throughout the place and route process, Multi-Mode Multi-Corner (MMMC) analysis was performed, involving simulations across various operating conditions such as best and worst-case temperature and voltage levels, ensuring that the ASIC would work within the operating ranges defined for the project.

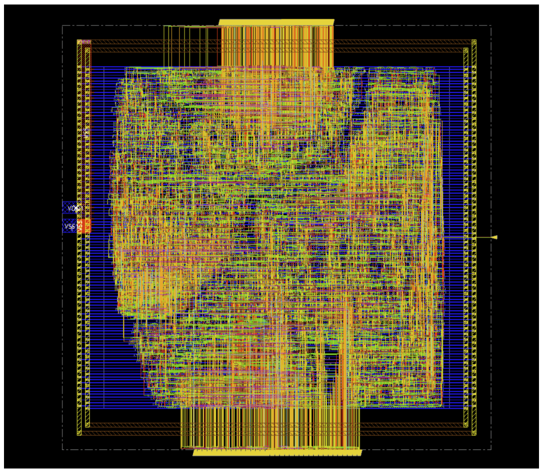


Fig. 4. Post-Clock Tree Synthesis RAPID-X Core.

Finally at the end of place and route Innovus exports a routed netlist and GDSII file which can be view in software such as Cadence Virtuoso. This final netlist can be used for an accurate gate level simulation and the GDSII file can have parasitic models generated in other software such as Cadence Quantus. There are also DRC violations which need to be fixed before a GDSII can be fabricated, but DRC issues are often fixed earlier within the place and route process.

Something to note about this project is that OpenRAM was imported into Innovus using the lef and liberty file generated by the OpenRAM SRAM compiler. This was achieved by importing the cells as macros during the floorplanning stage of place and route. To show an example of a GDS file the instruction cache for the core generated by OpenRAM can be seen in the figure below.

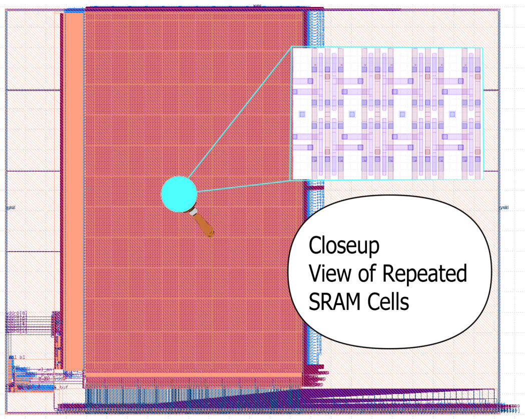
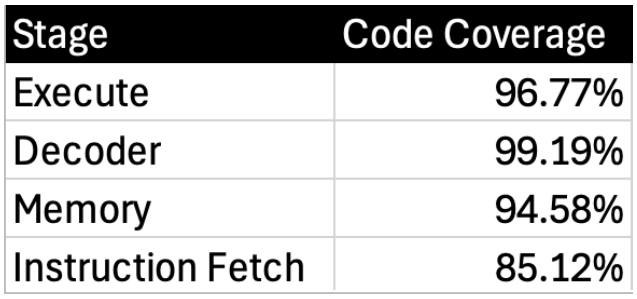


Fig. 5. SRAM GDS File.

VI. RESULTS AND ANALYSIS



Stage	Code Coverage
Execute	96.77%
Decoder	99.19%
Memory	94.58%
Instruction Fetch	85.12%

Fig. 6. XCelium Code coverage from every stage.

The table above demonstrate that code coverage exceeded the 85% threshold required by our engineering specifications. This result was achieved through the effective use of randomized input stimuli applied to various design stages.

The coverage reports confirm that all critical signals were exercised, and every functional block within the design was successfully activated during simulation, validating the thoroughness of our verification strategy.

A. *Individual Testbenches Validated*

As part of our modular testing strategy, each pipeline stage of the RAPID-X RV32I core was individually validated using structured testbenches. These testbenches ensured that each stage functioned correctly in isolation. For example, the Instruction Fetch (IF) unit's finite state machine (FSM) transitioned correctly through all operational states, as confirmed by waveform results that validated proper instruction delivery and program counter sequencing. The IF module also successfully handled memory stalls using our NOOP injection strategy and correctly responded to asynchronous control signals such as mid-execution program counter updates. Cache and memory outputs demonstrated aligned timing and consistency with expected instruction flows. Through a combination of waveform analysis and log inspection, it was confirmed that the FSM-driven behavior met the intended functionality.

Additional results from the Instruction Decode, Execute, and Memory stages confirmed correct data path behavior and control signal interactions across modules. The Decode unit correctly identified opcodes and activated the appropriate control lines. The Execute stage validated the ALU functionality across arithmetic, logical, and shift operations using a variety of test cases. The Memory stage was validated for accurate address calculations and data forwarding under controlled load/store scenarios. These testbenches also verified inter-stage communication—for instance, by feeding decoded instructions directly into the Execute stage to validate pipeline control propagation, or linking the Execute and Memory stages for realistic memory operation testing. Collectively, these individual testbenches validated each module in isolation prior to transitioning to a full-system UVM environment.

B. *UVM Code Coverage and Validation*

As the team transitioned from directed testbenches to a more standardized functional verification approach using Universal Verification Methodology (UVM), the focus on modular validation remained. UVM environments were first built for the Execute, Decode, and Memory FSM modules. After establishing confidence in their correctness, we scaled up to system-level UVM testing that included the Instruction Fetch stage and cache interactions.

Persistent testing efforts yielded strong code coverage results, with most modules exceeding the 85% threshold specified by our verification requirements. UVM tests used both randomized and constrained-random input sequences to stimulate a wide range of behaviors and configurations. Functional correctness was assessed through waveform inspection and UVM-generated logs (e.g., `UVM_INFO`, `UVM_ERROR`). These logs facilitated targeted debugging and resolution of unexpected behavior. As a result, UVM helped uncover latent edge cases and further solidified our design's functional robustness.

C. *Full Core Verification Benchmarks Matched RIPES.me*

At the core level, our full-system verification strategy included custom benchmark programs executed on the RAPID-X processor and compared against outputs from the RIPES.me visual simulator. These comparisons validated correctness in register values, program counter sequencing, and memory state updates. Benchmarks included cumulative addition, XOR loops, arithmetic progressions, and bit manipulation routines. Every instruction executed by the RAPID-X pipeline matched the outputs produced by RIPES, confirming accurate integration of all pipeline stages.

Control-flow instructions, such as branch if not equal instruction (`bne`), were tested to ensure correct path resolution and proper hazard detection. Particularly, the nested loop addition benchmark heavily stressed the pipeline's interlocking and control logic. The test successfully verified inner and outer loop transitions, loop counter handling, and consistent summation results. Waveform analysis confirmed clean transitions between loop iterations, without deadlocks or unintended stalls. Across all benchmark programs, RAPID-X preserved correctness under control and data hazards, validating the integrity of the entire architecture implementation.

D. *Timing Performance*

Compared to its predecessor, RAPID, the RAPID-X core exhibited substantial performance improvements. Under ideal benchmarking conditions, such as cumulative additions and XOR operations without data hazards, the RAPID-X core maintained a cycles-per-instruction (CPI) near 1.0, compared to approximately 3.0 in the original RAPID design. This performance gain was attributed to a tighter pipeline initiation interval and improved control path management.

Instruction throughput peaked at around 100 MIPS when simulated with a 100 MHz clock. Comparative benchmarks were performed against both the original RAPID implementation and the RIPES simulator. Results demonstrated consistent IPC uplift and reduced latency across all tested programs. For more complex workloads involving hazards and control-flow shifts, RAPID-X still outperformed RAPID. For example, during the nested loop benchmark, which introduced deliberate data dependencies and pipeline interlocks, the core recovered from hazards effectively while maintaining throughput.

In memory-intensive programs like bit rotation tests, the design's instruction cache and NOOP injection logic reduced latency and improved resilience. These results show that RAPID-X performs well not only under optimal conditions but also in more demanding scenarios, affirming its suitability as a high-performance microarchitecture for integer-based workloads.

The Innovus timing reports also showed that there was no negative slack, although the margins were pretty tight.

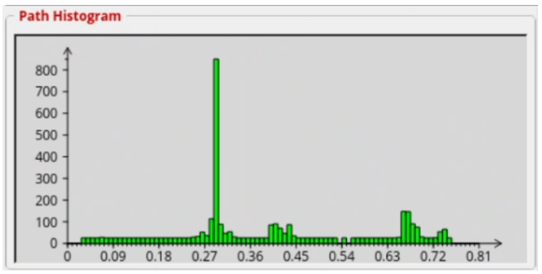


Fig. 7. Path Histogram: Slack (ns) and number of paths.

VII. CONCLUSION

Throughout the duration of the project, the team gained relevant experience in modern IC design flow. Using industry-standard tools and practices, the design processes progressed from defining the engineering specifications parting from the engineering requirements, to implementing the design in RTL and performing behavioral verification, to physical layout and signoff. The final design successfully met all engineering requirements approved by the sponsor.

Particularly important, the team became familiar with the Cadence EDA suite and utilized it to bring the design from specification to tapeout within a satisfactory timeframe. The end result is the layout of an ASIC processor implementing the RV32I standard of the RISC-V specification.

While the team and sponsor are satisfied with the final result, we acknowledge that there are other avenues in the IC design flow that can be explored to further increase the resiliency of the design. In particularly, future work can explore integration of design-for-test (DFT) features. Due to limitations of the target technology, the team was not able to implement functionality to aid in the testability of the design such as built-in-self-tests or scan chains. We're confident that further work in such areas to improve the existing design will lead to a chip capable of being fabricated.

REFERENCES

[1] K. Asanović, "Instruction Sets Should Be Free: The Case For RISC-V," *Electrical Engineering and Computer Sciences University of California at Berkeley*, 2014.

[2] RISC-V International, "The RISC-V Instruction Set Manual Volume I," 2024.

[3] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–6.