WHIPPET: Wide-swing High-performance Instruction Processing with Power-efficient Energy Tuning

Dev Patel
ECE Department
UIUC
Champaign, USA
devdp2

Jack Tipping
ECE Department
UIUC
Champaign, USA
jacket2

Jason Dhand
ECE Department
UIUC
Champaign, USA
jodhand2

Chirag Maheshwari
ECE Department
UIUC
Champaign, USA
chiragm4

Nazar Kalyniouk
ECE Department
UIUC
Champaign, USA
nazark2

Ryan Liem
ECE Department
UIUC
Champaign, USA
liem2

***Abstract*—This final report outlines W.H.I.P.P.E.T. (Wide-Switch High-performance Instruction Processing with Power-efficient Energy Tuning), a RISC-V microcontroller featuring dynamic voltage and frequency scaling (DVFS) to achieve high energy efficiency. At a high level, our processor can dynamically choose between power or performance, allowing it to reduce energy consumption during non-critical workloads. In order to achieve this, we developed a mixed signal design with custom SDLS standard cells. These standard cells take in two extra biasing voltages to adjust frequency and leakage, which are supplied by the analog blocks. The analog block also contains an Adaptive Clock Generator (ACG) that adjusts the global frequency of the CPU. On the digital side, we implemented a 32-bit RISC-V processor, which includes support for most fixed point vector instructions.**

***Index Terms*—Dynamic Voltage and Frequency Scaling (DVFS), Vector Processing Unit (VPU), Scalable Dynamic Leakage-Suppression (SDLS), Voltage Scaling Controller (VSC), Adaptive Clock Generator (ACG)**

I. Project Summary

A. Main Features and Functionality

1) DVFS Operation: To enable fine-grained performance, the DVFS framework scales frequency using the Voltage Scaling Controller (VSC) and an Adaptive Clock Generator (ACG). Rather than adjusting the supply voltage, the system maintains a constant V_{DD} and adjusts performance through bias-controlled tuning. The VSC generates complementary bias voltages, VCN and VCP, which are fed to custom designed standard cells to trade off leakage and delay. The ACG dynamically adjusts the system clock frequency to track the maximum operating speed based off these voltage biases. With a bank of 8 selectable ring oscillators, and a clock divider up to 128, we can scale the frequency anywhere from as low as 3.69 kHz, all the way up to 1.10 GHz. This wide frequency range enables operation at the target frequencies across all modes: Active (3.5 MHz), Moderate (500 kHz), and Passive (55 kHz). As we switch modes from active to passive mode, the SDLS standard cell's leakage current decreases, allowing the chip to consume less power.

2) RISC-V IMCV CPU: The CPU implements a 32-bit 5-stage pipelined RISC-V core with support for scalar multiply (M), compressed (C), and 64-bit fixed point vector instructions.

B. High-Level Architecture

1) CPU Architecture: For our CPU implementation, we built upon an existing 5-stage pipelined RV32I core as our foundation. Given our area budget and the requirements for integrating vector instruction support, we determined that an in-order microarchitecture was the only feasible approach. We implemented five standard RISC-V pipeline stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB), which were extended to accommodate both scalar and vector operations within a unified datapath.

a) Unified Pipeline Integration: A key architectural decision involved integrating vector unit logic directly into the existing Decode and Execute stages rather than implementing a separate dedicated vector coprocessor. This unified approach yielded significant area savings while maintaining functional correctness. By consolidating hardware resources within the existing execution units, each unit can execute both vector and scalar instructions. Scalar operations are handled as a special case of vector processing: the system configures SEW to 32 bits and zeroes out the upper 32 bits of the 64-bit datapath, allowing scalar instructions to appear as single-element vector operations with remaining vector lanes inactive. However, this design choice required careful construction of control logic and modifications to hazard detection due to vector-specific architectural features such as the `v0` mask register and vector control and status registers (`vtype`, `vl`, `vstart`), which require dedicated forwarding paths not present in the scalar-only pipeline.

b) Variable Vector Granularity: The granularity of vector operations in our design is dynamically configurable through setting the Standard Element Width (SEW), which dictates the size of individual elements within a vector register for an instruction. This variable granularity supports element widths ranging from 8 bits up to 64 bits, with the 8-bit granularity

serving as the fundamental building block for our execution units.

To illustrate this design principle, consider our adder implementation: it comprises eight parallel 8-bit adder units with configurable carry propagation. The carry chain connections between these adders are selectively enabled or disabled based on the current SEW value, allowing the same hardware to efficiently process elements of varying widths:

- **SEW = 8 bits:** All eight adders operate independently with no carry propagation, enabling eight parallel 8-bit additions per cycle.
- **SEW = 16 bits:** Carry chains connect pairs of 8-bit adders, forming four 16-bit adders.
- **SEW = 32 bits:** Carries propagate across four consecutive adders, creating two 32-bit adders.
- **SEW = 64 bits:** All eight adders function as a unified 64-bit computational unit with full carry propagation.

This modular, width-agnostic design methodology extends to all arithmetic and logical execution units, providing flexibility while maximizing hardware reuse

c) Register File Architecture and Data Flow: A critical architectural distinction in our design is the presence of separate register files for vector and scalar operations. The scalar register file contains thirty-two 32-bit general-purpose registers (`x0-x31`), while the vector register file contains thirty-two 64-bit vector registers (`v0-v31`), each capable of holding multiple elements depending on the SEW configuration.

This separation necessitated careful management of data flow throughout the pipeline:

- **Decode Stage:** Instruction type is determined, and appropriate register file read ports are selected via multiplexers based on whether operands reside in the scalar or vector register file.
- **Execute Stage:** Multiplexer select signals route operands from the correct register file to the execution units. Mixed-mode operations (e.g., vector-scalar instructions) require additional logic to properly align and broadcast scalar values across vector lanes.
- **Writeback Stage:** Instruction type tracking ensures results are written to the appropriate register file. Write enable signals are generated separately for each register file, with the active signal determined by the instruction class decoded in earlier stages.

Control logic maintains proper synchronization between the two register files and ensures that data hazards are detected correctly regardless of whether instructions operate on scalar or vector registers. This dual register file architecture adds complexity to the control path but provides clear separation of concerns. It also simplifies the implementation of vector-specific features such as element-level masking and vector length management.

2) SDLS Standard Cell Design: In order to reduce leakage current by trading off performance, we use custom scalable dynamic leakage suppression (SDLS) cells, which are based off Truesdell et al. [1]. We can do this by adding two extra nmos cells above the standard VDD rail and two extra pmos cells below the standard VSS rail. These extra cells are connected to two biasing voltages, VCN and VCP, which are inverses of each other.

The schematic modification for each cell is shown in Fig. 1. In the schematic, n1 will act as the new power supply rail to the standard cell, and n2 will act as the new ground rail for the standard cell. Everything else in the standard cell remains unchanged.

In our current schematic configuration, we are able to size the SDLS standard cells to a 1.83x height increase, and a small increase less than 0.2 um in width for small standard cells, shown in Fig. 2. In total, there is roughly a 2x increase in area compared to the ARM standard cell library.

The SDLS mechanism allows us to reduce leakage by reducing VCN and increasing VCP. To simplify things, let us define an extra variable connecting these two:

$$Vc(V), \quad \{0 < V < 0.5\}$$

$$VCN = 0.8 + Vc(V)$$

$$VCP = 0.5 - Vc(V)$$

This bias range is derived from our simulation results, which showed a leakage current reduction of approximately 3x, while still maintaining functionality.

Transient simulation results of a Vc Sweep from 0 to 0.5 can be seen in Fig. 3. The SDLS inverter is chained to another SDLS inverter, and the second inverter is given a capacitor load (this set-up is essentially a buffer). As we reduce VCN and increase VCP, the performance starts to degrade and the output response of both inverters slow down.

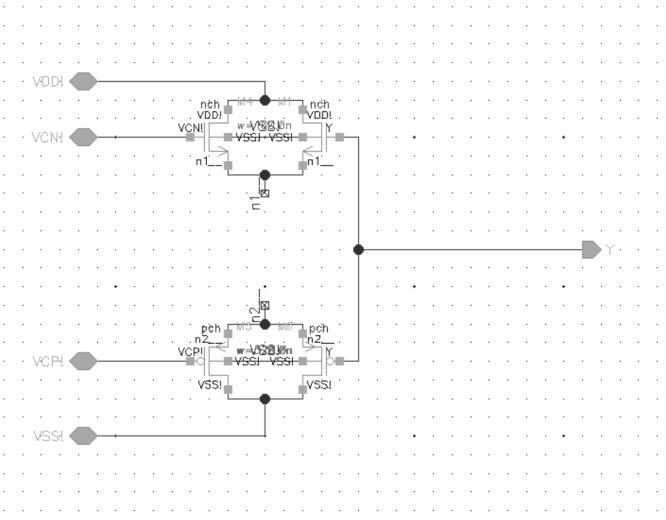

Fig. 1. SDLS Cell Schematic

3) Analog Design:

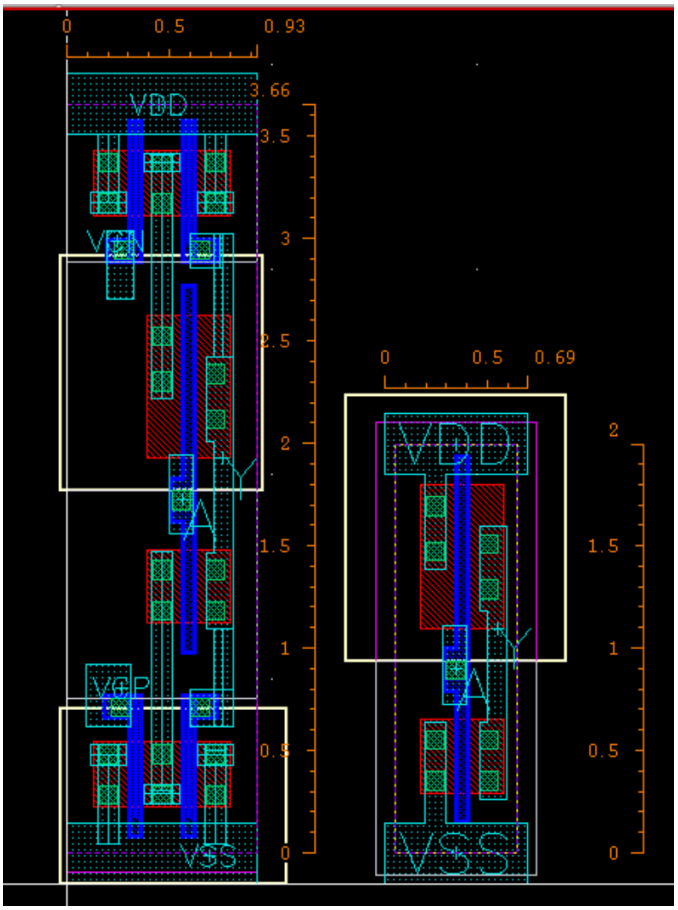

Fig. 2. SDLS Inverter Layout

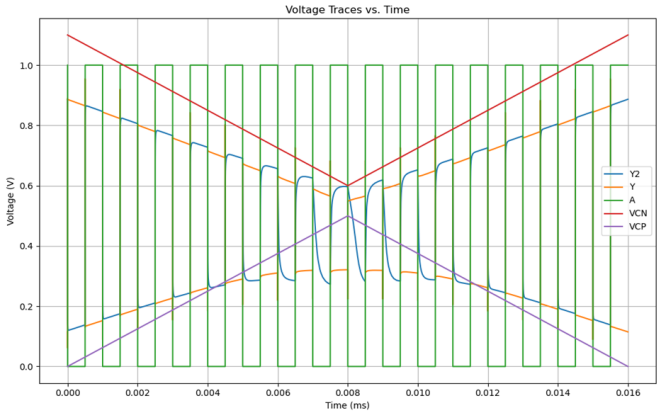

Fig. 3. SDLS Inverter Transient Simulation

a) Adaptive Clock Generator: The Adaptive Clock Generator (ACG) provides a dynamic clock that tracks the delay of the processor's critical path under bias voltages, V_{CN} and V_{CP}, to allow for DVFS operation. This approach allows us to operate by keeping the supply voltage, V_{DD}, constant. The ring oscillators are implemented using SDLS logic, allowing the system to adapt to changes in leakage suppression strength and process variation. The ACG contains a bank of ring oscillators or varying inverter length, coupled with a clock divider, to allow for a wide range of frequency scaling. Achievable frequencies range from 3.69 kHz to 1.10 GHz. The ACG is able to maintain near-maximum performance without violating timing constraints. A block diagram is shown in Fig.4

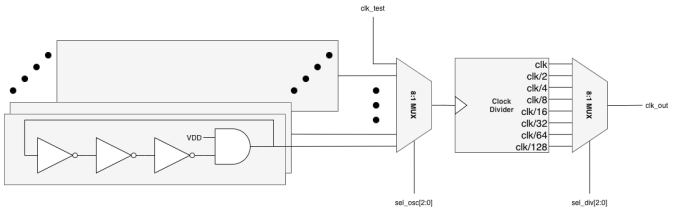

Fig. 4. ACG Schematic

b) Voltage Scaling Controller: The Voltage scaling controller switches between three primary operating modes (Active, Passive, and Moderate). These three operating modes cover the entire rail to rail voltage range, which posed a very unique challenge in our design. However, one aspect of these biasing voltages that made our problem space significantly simpler was the fact that they drive the gates of mosfets, effectively reducing the load current requirements. This simplified the biasing problem space to only focusing on accurate and stable output voltages, which allowed us to use smaller, low-speed, and more noise sensitive Op-Amp designs (since the reference voltage inputs and outputs are relatively constant).

Looking over the schematic in Fig.5, we first provide the reference voltages into a series of rail-to-rail class AB buffers. We chose off chip references to minimize area, and to avoid and distortion from the input impedance of the AB buffers. The buffered voltage values (0, 0.25, 0.5, 0.8, 1.05, 1.3) are then passed into two analog muxes (one for VCN, one for VCP). The DVFS register on the CPU dictates which bias voltage value to select before passing the reference voltage into the final output stage (one stage for each biasing rail).

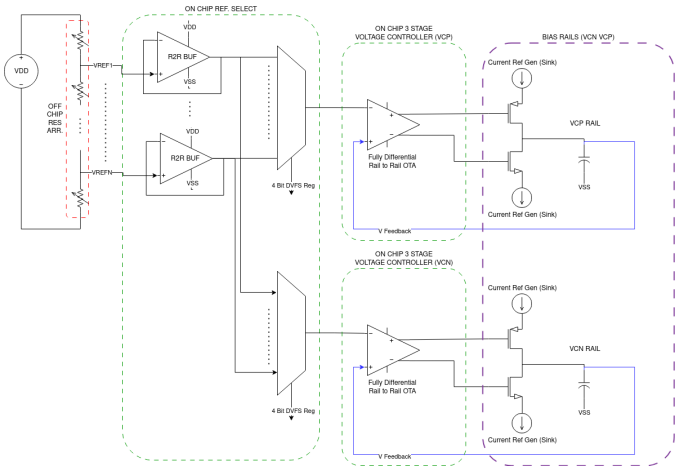

Fig. 5. VSC Schematic

The output stage proved to be the most interesting part of the analog controller. We decided to use a ”bang-bang” controller, which consists of a fully differential rail-to-rail OTA, which then drives a dual mosfet current source/sink. This means that the output voltage constantly oscillates over a small ripple voltage, rather than being held constant. As it averages over time, the voltage appears constant, but a closer analysis reveals a sawtooth like curve. During simulation, we found this noise had a negligible affect on the SDLS Cells. The current references for all the devices in the VSC were provided by a Beta Multiplier based current reference, which is similar to an active differential pair. This Beta Mult has a very small area profile compared to a traditional bandgap based reference, although it is more susceptible to PVT variances.

C. Physical Design

Since the SDLS standard cells consume much more area compared to the provided ARM standard cell library, we were unable to fit the entire design in the constrained area if we were to use them for everything. As such, we only utilize the SDLS cells on the execute block of the CPU, since that is the most utilized stage of the pipeline. Because of this, we had to perform extra PnR steps:

- PnR the execute block with only SDLS cells as a separate block.
- Perform LEF extraction on the PnRed module, and create a functional liberty file.
- Synthesize the entire CPU with the execute block as a ’black box’, such that synth will only generate the netlist for the rest of the CPU with ARM stdcells.
- Place the analog, cache blocks, and the execute block as a ’PnRed module’, and perform PnR on the rest of the CPU with ARM stdcells.
- Integrate the PnR block with the IO bond pads, and manually route the biasing lines from the analog to SDLS execute block.

A final layout of our design can be seen in Fig. 7, and the final PnR result can be seen in Fig. 6. The bottom left of the chip is occupied with the SDLS execute block, the top left is occupied by the analog block, and the caches are distributed above the execute block and across the right side of the chip.

In order to increase usable area, we removed the right side of the IO pads. The rest of the IO pads are used for off-chip communication, biasing voltage inputs, failover clock, debug pin, and debug pins.

The Total Utilization of our design is 98.2%, with a density of 57.8% before fillers. An observation can be made that this high utilization is in the congested part of the design between the execute block and caches. The utilization percentage could be lowered by changing the SDLS execute block to be a non-rectangular shape, which would allow the routing to expand outwards without congesting the design. However, we decided not to do this since the current PnR was already working.

The design’s frequency is dependent on the current operation mode. We have three different modes: Active (3.5 MHz), Moderate (500 kHz), and Passive (55 kHz). Generally, each

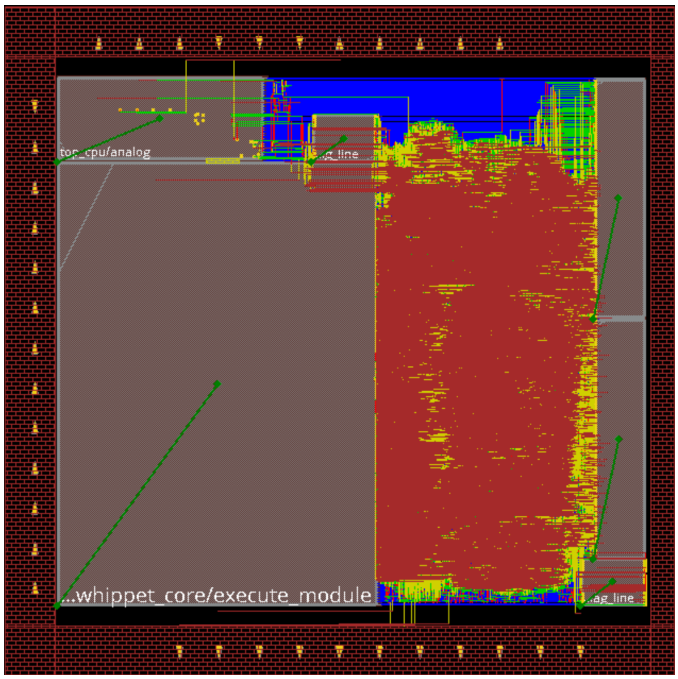

Fig. 6. Final Place and Route

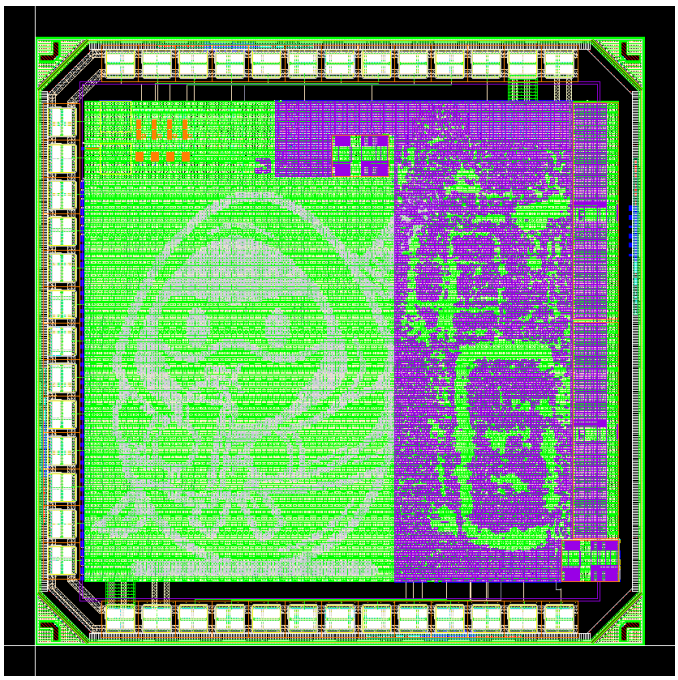

Fig. 7. Final Chip Layout

mode's listed frequency is an upper bound, and we are able to scale down the frequency based on the ACG's supported range of 3.69 kHz to 1.10 GHz.

In terms of power consumption, initial synthesis results show 2.8 mW of power consumption. This number should be the lower bound of the chip's power consumption, since it doesn't take into account the SDLS execute block. Modeling power on the SDLS execute block proved to be too difficult, since there are 3 different operating modes, and acquiring the timing numbers already took a significant amount of time. As such, we will perform the power measurements in the bringup process.

D. Design Choices and Justification

- We decided to only utilize the custom standard cells for the execute module. This decision was because the custom cells consumed too much area to apply to the entire design. Synthesis estimates us requiring around 2-3 mm^2. As such, we chose to SDLS the execute module since it was the most utilized part of the design.
- Our major breakthrough on saving area came from removing one entire side of the IO pads. In order to do this, we reduced our off chip communication pins from 16 to 8. This does slow down our operation, but we couldn't fit the design otherwise. As we removed one side of the IO pads, nearly all of our area concerns were solved.
- For the ACG, we decided to implement a seven-stage ring oscillator combined with a clock divider, programmable up to 128, enables a wide range of achievable operating frequencies. Additionally, a failover clock was added to support debugging. This was done to expand the available frequency ranges, and increase robustness of the design.
- The VSC utilizes a smaller Op-Amp design, since they are only driving mosfet gates, reducing the load current requirements. This is done to minimize area while keeping functionality.
- We chose not to implement certain subsets of the vector extension, namely permutations, widening, and multiply/add instructions. Most notably, we did not add support for LMUL > 1, though we do support fractional LMULs. This limitation means that vector operations are restricted to at most (64 / SEW) elements per instruction. We made this decision primarily to reduce area overhead. Furthermore, certain vector instructions with LMUL $>$ 1 cannot be decomposed into multiple LMUL = 1 operations. For example, reduction instructions (such as `vredsum`) that accumulate across entire register groups require maintaining dependencies and intermediate state across all registers in the group, which would have required substantial additional control logic and significantly increased design complexity.

E. Verification and Test Strategy

1) Digital Verification: Our verification methodology employed a multi-faceted approach to validate both scalar and vector instruction execution, with particular emphasis on ensuring correctness of the newly integrated vector extensions. The verification strategy combined reference model validation, directed testing, and coverage-driven random instruction generation to achieve comprehensive functional verification.

a) Reference Model Integration: To establish a golden reference for instruction behavior, we integrated vector extension support into the Spike RISC-V ISA simulator. Spike served as our primary reference model for validating instruction-level correctness, allowing us to compare our CPU's execution results against a known-correct implementation. This approach provided high confidence in the semantic correctness of both scalar RV32IM instructions and the Zve64x vector extension instructions.

b) Verification Infrastructure Challenges: Initial exploration of using the RISC-V Formal Interface (RVFI) for trace-based verification revealed significant limitations for our use case. The existing RVFI infrastructure was designed primarily for scalar register architectures and lacked native support for vector register files and vector-specific control state. Extending RVFI to accommodate the full vector architecture—including vector registers (`v0- v31`), vector length (`vl`), vector type (`vtype`), and element-level masking—would have required substantial modifications to the specification and reference implementations. Given project time constraints and the substantial modifications required, we developed an alternative verification approach.

c) Python-Based Test Generation Framework: Rather than employing traditional UVM testbenches, we developed a lightweight Python-based test generation framework that produces assembly programs with controlled instruction distributions and coverage targeting. This script-based approach offered several advantages: rapid test development, deterministic coverage tracking, and seamless integration with Spike for reference execution.

The Python framework generates randomized assembly programs that exercise specific functional units and instruction classes based on configurable parameters. The test generator employs structured randomization to create realistic code patterns including loops, conditional branches, and straight-line code sequences, ensuring that generated programs stress realistic execution scenarios rather than isolated instruction sequences. Generated assembly files are executed in both our RTL simulator and Spike, enabling automated comparison of architectural state after execution.

d) Coverage Methodology: Functional coverage was tracked using a bin-based approach derived from instruction opcodes and function fields. The verification framework maintains coverage bins for each instruction type, operand combination class, and architectural feature (including different SEW configurations, vector masking modes, and register hazard scenarios). Coverage reports are generated automatically after each simulation run, identifying gaps in the verification space and guiding subsequent test generation to target uncovered scenarios.

This coverage-driven methodology allowed us to systematically verify the complete instruction set while maintaining visibility into verification completeness. By monitoring opcode coverage, operand pattern coverage, and architectural state coverage, we achieved comprehensive verification of the integrated scalar-vector pipeline without the overhead of a full UVM infrastructure.

2) Analog Verification: Generally, verification for analog components consisted of performing ADE simulations of the circuits. The Voltage Scaling Controller and Adaptive Clcok Generator were both simulated separately, and then a final test-suite was performed, combining all of the analog components together. A similar methodology was done for verifying functionality of the standard cells.

a) Voltage Scaling Controller: The VSC has debug pins that carry out the voltage readings. These can be verified in test with an oscilloscope. For simulation, we used mixed signal xcelium scripts to combine verilog test coverage for the digital register control with parasitic aware layout simulations, allowing us to verify all test cases and integration with the ACG.

b) Adaptive Clock Generator: The clk_out pin will be used as a test output to measure the generated clock at different operating points. In addition, a dedicated failover clock will be used as a DFT feature to facilitate debugging and bring-up of the ACG by providing a known and stable clock source. Together, we will use these methods to enable validation of functional correctness and timing behavior across operating modes.

c) Standard Cells: Functionality of standard cells was verified with custom ADE scripts. At first, we performed individual simulations with several standard cells to determine valid biasing voltage ranges, discussed in the SDLS Standard Cell Design portion of the report. In order to verify functionality of multiple standard cells, we constructed and simulated a full 8-bit adder, since running a simulation on the entire execute block would take too long.

F. Comparison with Initial Proposal

1) Adaptive Clock Generator (ACG): Compared to the initial ACG proposal inspired by Truesdell et al. [1], the final design underwent multiple changes, while preserving overall functionality. The original design included an adjustable duty cycle, which was removed in favor of a fixed 50% duty cycle to have a design with more balanced timing margins. Additionally, a clock divider was incorporated to allow for wider frequency scaling from the ring oscillators while reducing overall area overhead. The decoder previously used to selectively power individual ring oscillators was eliminated, and a glitch-less multiplexer was introduced. This modification prevents unexpected transitions in the ACG output that were previously observed during transitions between clock domains.

2) SDLS Cell Utilization: Initially, we proposed to utilize the custom standard cell for the entire CPU. However, due to area constraints, we were unable to do this, and decided to only utilize the SDLS cells on the execute module. This is further discussed in the Design Choices and Justification section of this report.

3) Vector Extension: Our initial proposal wanted to support the entire RISC-V vector extension. However, we decided not to implement specific subsets of the vector extension. This includes extension, permutations, widening, and multiply/add instructions. Furthermore, we do not support LMUL $>$ 1. The main reason for this is due to area constraints. This is extensively discussed in the Design Choices and Justification section of this report.

II. Post-Silicon Validation Plan

Our post-silicon validation and bring-up plan includes checking chip functionality and power consumption across different operating modes. The validation process requires custom hardware interfacing, FPGA-based testing infrastructure, and modified software toolchain support. This process will be performed by Jason, Nazar, and Ryan.

a) PCB and FPGA Interface: We will design a custom PCB to interface the fabricated chip with an FPGA test platform. The PCB supplies power to the chip, provides necessary biasing voltages, and routes signals between the chip and FPGA. It also includes a failover clock input for debugging scenarios where the on-chip clock generator fails. The FPGA provides instruction and data memory and implements an custom off-chip communication protocol supported by our CPU.

b) Compiler Modifications: Since our design implements only a subset of the RISC-V vector extension, we must modify an existing RISC-V compiler to generate code using our supported instructions. Unimplemented vector operations will be expressed as sequences of supported instructions or emulated in software where necessary.

c) Benchmark Testing: Once the hardware and compiler are functional, we will run programs that emulate common edge AI workloads to measure power consumption and execution speed across different operational modes. This will validate our design's performance targets for intensive applications.

d) Debug Capabilities: If the chip does not function as expected, we have several debugging mechanisms: a failover clock input allows operation even if the clock generation fails, and dedicated debug pins provide visibility into internal chip state. These features should enable at least partial functionality testing and help diagnose any silicon issues.

III. Individual Contributions

A. Jason

Worked primarily on RTL design and verification. Responsible for development of vector execution units and register files, vector control register integration, and development of a Python-based testing and coverage report generation. Assisted in post synthesis verification and integrated decode stage.

B. Dev

Worked primarily on design, simulation, and layout of the VSC and Current reference generator (with help from Nazar). Helped Nazar with ACG RTL design and verification. Also helped Ryan with transient simulation of SDLS cells. Lead integration effort between Analog subcomponents (ACG, SDLS, VSC).

C. Ryan

Worked primarily on design, layout, and simulation of all 29 SDLS standard cells. Obtained timing values for .lib files and developed custom flows to extract numbers and generate correct file format. Led physical design from synthesis to PnR, including creation of custom scripts and required files. Helped with SRAM setup, post-synthesis and post-PnR simulations, and also modified RTL to integrate analog and execute properly for synthesis/PnR.

D. Nazar

Worked on the design, layout, and simulation of the Adaptive Clock Generator (ACG), including both physical design and RTL implementation (with help from Dev). Assisted with the design of the Voltage Scaling Controller (VSC) and current reference generator, and worked on the integration of analog components.

E. Jack

Worked primarily on RTL design and verification. Responsible for integration of on-chip SRAM, implementation of compressed instructions, off-chip communication, debug operation and vector memory instructions. Performed post-synthesis and post-PnR verification.

F. Chirag

Worked on adding vector control registers integration, Python-based random instruction generation and RTL verification.

IV. Major Challenges and Lessons Learned

One of the primary challenges encountered during digital design development stemmed from ambiguities in interpreting the Zve64x specification. The vector extension specification, while comprehensive, contains numerous edge cases and implementation-defined behaviors that proved difficult to reconcile during initial hardware development. This led to considerable uncertainty regarding the expected functionality of certain vector instructions, particularly for masked operations, element width transitions, and vector length handling.

Throughout the early hardware development phase, our primary debugging methodology relied on manual waveform inspection, which proved increasingly inefficient as design complexity grew. The iterative process of hypothesizing expected behavior, implementing it in RTL, and validating through waveform analysis was both time-consuming and error-prone, often requiring multiple design iterations to converge on correct functionality.

The integration of Spike as a reference model represented a turning point in our verification strategy. Once Spike was incorporated into our workflow, ambiguities regarding instruction semantics were immediately resolved through direct comparison with the reference implementation. This dramatically accelerated development velocity and reduced debug time, as discrepancies could be traced to specific architectural state differences rather than requiring extensive manual waveform analysis.

In retrospect, integrating Spike at an earlier stage of RTL design would have yielded significant time savings and reduced early-stage design churn.

A second major challenge arose from stringent area constraints that necessitated difficult trade-offs in hardware resource allocation. Initial synthesis results revealed that our design exceeded available silicon area, requiring careful evaluation of which features to retain and which to sacrifice. Most notably, we removed a dedicated 64-bit multiplier hardware, opting instead to implement multiplication through software emulation or reduced-width operations where necessary. This decision reflected a pragmatic assessment that full 64-bit multiplication throughput was not critical for our target application domain.

Similarly, area pressure forced us to prune the vector instruction set, removing operations deemed either redundant or less essential for our anticipated workloads. Instructions that could be reasonably emulated through short sequences of other vector operations were prioritized for removal, as were specialized variants that provided marginal performance benefits over more general alternatives. This selective implementation required careful analysis of instruction usage patterns and performance impact to ensure that the reduced instruction set remained functionally complete for our application requirements.

The design process for our standard cells also posed a significant challenge. While schematic and layout creation was fairly straightforward, acquiring timing numbers for these cells proved to be significantly challenging.

The provided Cadence Liberate tool was supposed to generate the timing numbers, but it failed and didn't generate any timing numbers. Other alternative tools such as Synopsys Silicon Smart also proved to be faulty. However, the tool did generate simulation scripts, which eventually allowed us to run the simulation ourselves and extract the timing numbers from the simulation results.

Many of the analog challenges regarded area, but a persistent issue was our approach to MOSFET Sizing. Initially we were very disorganized with our GM/ID scripts and sizing, and this lead to bottlenecks and delays during simulation and layout. After utilizing data analysis tools and python scrips to help process the GM/ID PDK values, we saw better correlation between design, simulation and layout.

We also took some time to figure out the integration process of the different blocks. Since we essentially had 3 different parts of the design (SDLS Block, Analog Block, and normal stdcell block), we had to understand how to properly integrate

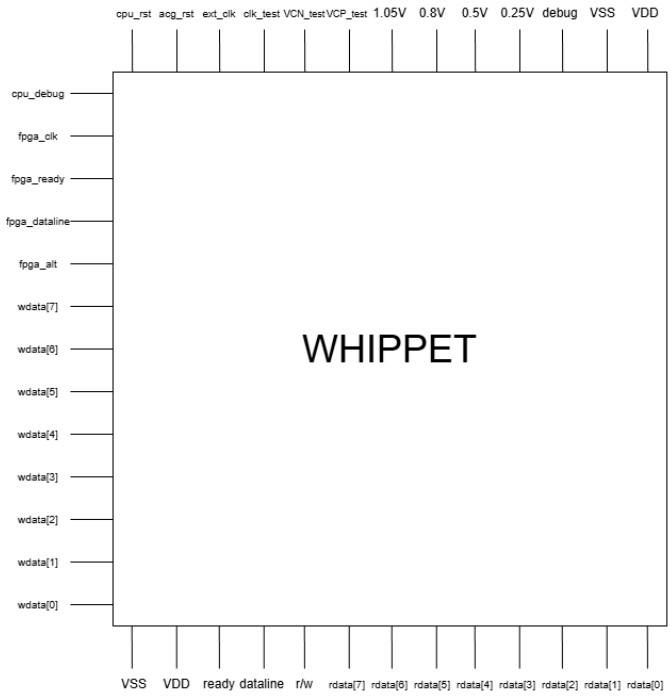

Fig. 8. Final Chip Pin Assignments

these blocks into the PnR process. Eventually, we realized that it was possible to embed these blocks into the RTL itself, and essentially have the PnR tool perform the integration.

Overall, the design process for our mixed-signal cpu proved to be a significant challenge, from the digital side to the analog side. Many of our challenges stemmed from being initially disorganized, but as the semester went on, we developed our methods to stay on track. A big lesson to be learned is the importance of organization, since our team would have definitely performed better had we been more diligent and organized from the start.

V. Documentation

A pinout diagram can be seen in Fig.8. For the analog part of the design, we have multiple input pins for biasing lines, labaled according to their respective voltage values. For off-chip communication, we have a write line and a read line, each with their own data and ready pins. Since the FPGA's clock and the CPU's clock isn't synchronized, we utilize an async FIFO, which is why we feed in the fpga clock to the chip. Additionally. we have an external clock input in case the ACG fails, and a debug pin to monitor the current value of the registers.

At the highest level, we can divide our design into two different halves, digital and analog. The full block diagram for the digital side is provided in Fig. 9, while the full block diagram for the analog side is provided in Fig. 10. The main components of our design include:

a) RISC-V IMCV CPU: The main basis of our digital design is a RISC-V IMCV 5-stage pipelined CPU, which supports scalar multiply (M), compressed (C), and 64-bit fixed point vector instructions. The design is heavily discussed in the High-Level Architecture portion of this report.

b) SDLS Standard Cells: Scalable Dynamic Leakage Suppression (SDLS) cells take in two extra biasing voltages VCN and VCP in order to trade off power and delay, allowing us to effectively reduce power consumption. The design is discussed in the High-Level Architecture portion of this report.

c) Adaptive Clock Generator: The Adaptive Clock Generate Generator (ACG) generates a dynamic clock and allows us to change the frequency of the entire CPU. A block diagram is provided in Fig. 4. The design is discussed in the High-Level Architecture portion of this report.

d) Voltage Scaling Controller: The Voltage Scaling Controller (VSC) is required to alternate between biasing voltages, and switch between the three primary operating modes. A block diagram is provided in Fig. 5. The design is discussed in the High-Level Architecture portion of this report.

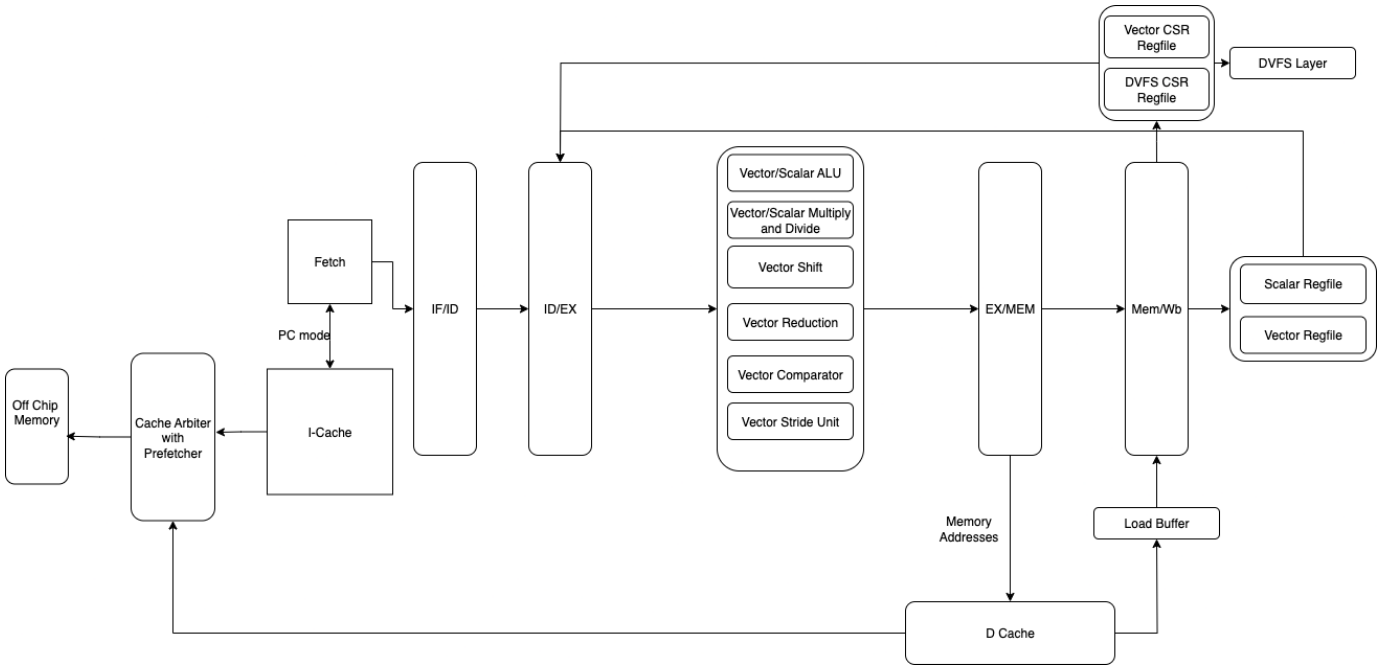

Fig. 9. RTL Design Block Diagram

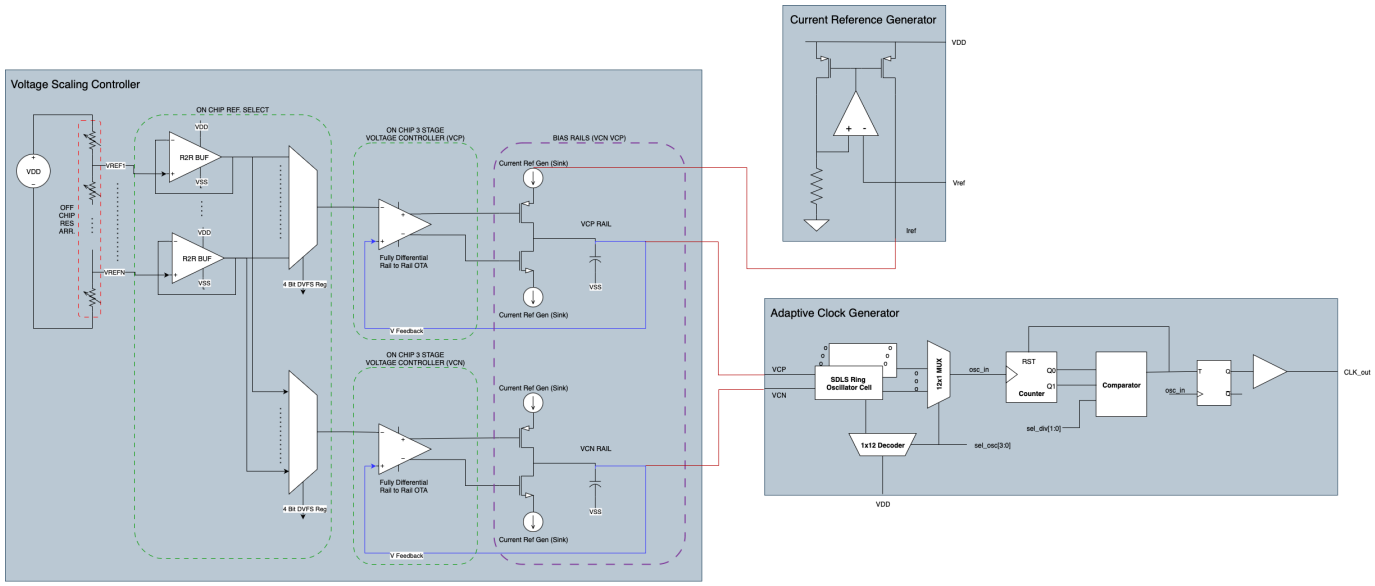

Fig. 10. Full Analog Block Diagram

VI. Testing Instructions

We divide the Testing Instructions into multiple parts. If a TA has any trouble with accessing or verifying these, please don't hesitate to reach out to any of us through message or email, we will be happy to assist.

A. RTL

Currently, the most up to date RTL and verification scripts are located in /home/jacket2. However, it would be slightly complicated to get the scripts up and running from there, so instead here is our github repository link: https://github.com/JasonDhand0508/ECE427_RTL. We think it would be easier to set it up by cloning the repo instead.

The github repo contains all RTL and Verification files, and the current up to date branch is labeled offchipmem_int_tiger. If a TA wants access to verify our work, please message or email Jason Dhand at jodhand2@illinois.edu.

B. Analog

All test benches and schematics are available in the respective user directories (devdp2/nazark2) within the WHIPPET folder.

C. Standard Cells

All standard cells are located in whippet_stdcells_nowtap library. If you want to check all the schematics and layout, run virtuoso on /home/liem2/WHIPPET/ and check that library.

Standard cell timing flows are all run through Cadence Liberate, which does not work. However, it does generate the simulations, so there are scripts that run the simulations and extract the numbers. These scripts are located in /home/liem2/LIBERATE. The master script is runlib.sh. However, since the scripts are tied to the paths, it would be difficult to verify the scripts if we move the directories.

All the .lib file outputs are located in /home/liem2/PNR_FILES and in /home/liem2/example/ip, along with files required for PNR (LEF, GDS.

D. Synthesis / PNR

The modified synthesis/PNR scripts are located in /home/liem2/example, in subfolders syn and pnr respectively. Again, since the scripts are tied to the paths, it would be difficult to move it to another directory. However, you can see the synth outputs and PnR outputs. To see the final PnR run, run make restore on /home/liem2/example/pnr.

References

[1] D. Truesdell, J. Breiholz, S. Kamineni, N. Liu, A. Magyar, and B. H. Calhoun, "A 6–140 nW 11 Hz–8.2 kHz DVFS RISC-V microprocessor using scalable dynamic leakage-suppression logic," *IEEE Solid-State Circuits Letters*, vol. 2, pp. 1–1, 2019, doi: 10.1109/LSSC.2019.2938897.

[2] VLSI Tutorials, "Glitch free clock mux," *VLSI Tutorials*. [Online]. Available: https://vlsitutorials.com/glitch-free-clock-mux/