Rice RISC-V

Juan Garza
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
jag33@rice.edu

Haibo Li
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
hl187@rice.edu

Sean Hamilton
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
smh19@rice.edu

Lindsey Russ
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
ltr1@rice.edu

Chenran Li
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
cl201@rice.edu

Lechuan Sun
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
ls127@rice.edu

Chengyun Tang
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
ct77@rice.edu

***Abstract*—The RISC-V ISA, an open-source specification, is gaining popularity across industries often relying on proprietary IP components. Our team developed a fully open-source RISC-V microcontroller with associated IP blocks addressing this. Current solutions rely on UVM and tools like Cocotb for verifying design, but lack a standardized approach for verifying RISC-V compatibility. Our solution integrates Cocotb, UVM, and FPGA emulation into a single verification test suite. This ensures consistent results across simulation and hardware testing, covering designs like the core, AMBA, and GPIO. Looking forward, our framework streamlines testing for creating a revised RISC V core next semester.**

I. Introduction

RISC-V is rapidly gaining adoption across various devices and industries. Despite its open nature, many RISC-V implementations still depend on proprietary Intellectual Property (IP) for certain design components. To address this, our Rice team has developed a fully open-source RISC-V core and associated IP blocks, which require integration and thorough verification within a unified framework. Current industry practices use UVM (Universal Verification Methodology) and open-source tools like Cocotb for automated testing, as well as FPGA emulation for hardware validation.

Our proposed solution is a standardized verification suite that is compatible with any RISC-V design, from complete Systems on Chip (SoCs) to individual sub-blocks. The suite leverages Cocotb for unit testing, UVM for more complex verification tasks, and FPGA emulation for exhaustive block-level testing.

Using our verification suite we are able to achieve the same results in both simulation (UVM and Cocotb) and FPGA testing across all of our current designs which include the core, AMBA, and GPIO blocks. With the verification suite creating new designs, whether IP blocks or complete SOCs will be efficient for future team usage and our testing methodology will be up to the standard of a chip design company.

II. Methods

A. Simulation Based Verification

1) Verilog: We began verifying the core with simple Verilog based testbenches. The main purpose of these tests was to ensure general functionality of the current modules. We wanted to make sure that we were able to put in inputs and get out coherent outputs. Further, having Verilog testbenches in place makes it easy to test specific errors that we may encounter further down the line.

While we mostly used Verilog to test individual modules, we also used it to test combinations of modules to figure out how they work with each other and if any commands were overwriting each other and causing the core to work in unexpected ways. For example, the regfile was being written to by most modules in the exu as well as the wbu. To address a concern that these values may be overwriting each other, we created a top module containing these modules and tested this with Verilog, with the results being shown in Figure 1. This showed that, while we cannot guarantee from simulation that this is how it would work in hardware, the overwriting seems to not be an issue. The regfile receives a "high impedance" value from all the other modules except for the one intending to write an actual value, and so the regfile takes the one intended value.

While Verilog testbenches are helpful in their simplicity and specificity, it is challenging to test full functionality of a given

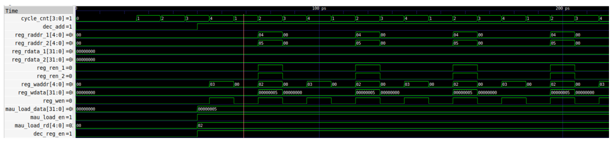


Fig. 1. Waveform for exu wbu regfile test

module with Verilog alone. This is why we chose to use Coco TB to do more thorough simulations of individual modules and of the core as a whole.

2) Coco TB: Coco TB is a Python based verification framework that enables the user to leverage Python systems to create a more flexible and efficient environment for developing testbenches [1]. Unlike traditional hardware development languages Coco TB makes large use of Python libraries and tools, and allows for high level test scenarios which largely make use of coroutines to mimic asynchronous hardware interactions [2]. This largely simplifies the verification process and also supports features such as randomization, assertion-based checks, and functional coverage which ensure rigorous testing.

Using this framework we created more rigorous testing environments for the individual modules especially with regard to randomness and series of instructions in a row. We began first by working with individual modules and generating instruction and input data for each one specifically to test the various instructions with general random inputs. The modules tested in this way were the IFU, decoder, EXU, MAU, and REGFILE. These modules were tested independently as each one had its own specific scenarios that could be triggered, whereas the WBU was only responsible for porting the information from the MAU to the REGFILE, and thus was tested in combination with those two circuits.

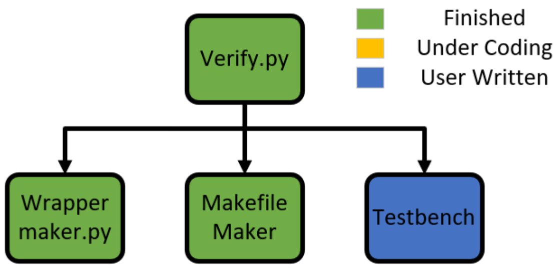


Fig. 2. Single module testing workflow

With regards to the IFU, we needed to confirm that the instruction that would come from our bus would properly propagate to the decoder, and that it would take into account the stalls within the system. One of the quirks of the core that we were working with is a stall system, that unfortunately limits the ability for the core to be pipelined. For the decoder, we checked the output results based on instruction type. To do this we developed a system for generating random instructions within each instruction type using parameters to control a python script that would generate bit strings accordingly. Thus for our testing of the decoder we would run randomized tests for each of the 5 types of instructions, checking the output register addresses and decoded instruction flag. For the EXU each module was individually tested both in Verilog and in Coco TB. To test the execution of every possible command, commands and other inputs were randomly generated in accordance with the module under test, and python models for the functionality of each EXU module were written for comparison. For each module, 50 tests were randomly generated and checked to see if the results matched up with the expected values from the python model to ensure the calculations were in accordance with the RV32I standard [3]. These randomly generated test cases allowed for edge case testing that the basic Verilog testbench could not catch. Several computation errors were found and corrected during this process. For the MAU and REGFILE the individual task of each was much smaller and as a result testing individually was less rigorous and mainly was to confirm that the various loads within the MAU as well as updating of register 1 and register 2 in the REGFILE was being handled correctly.

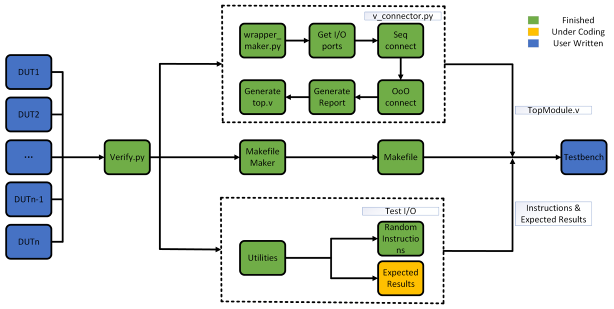
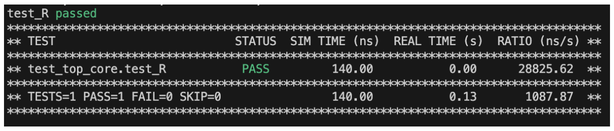


Fig. 3. Multiple module testing workflow

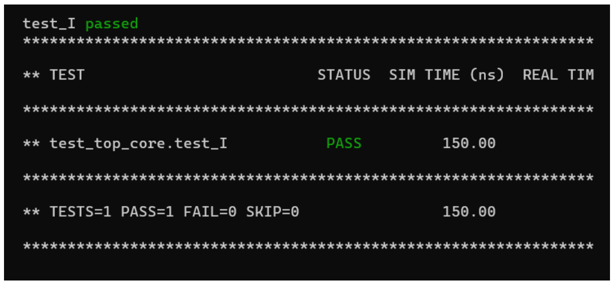
After individual testing of modules, module groups were tested together, mainly the IFU and decoder, as well as the MAU, WBU, and REGFILE. The linking of these modules was automated with the '$v\_connector.py$' script which ensured that correctly named ports would be wired together, with outputs overriding inputs in the top level wrapper. Since Coco TB is unable to view internal registers, or wiring within a module it was important to define each of the outputs from our modules as outputs in the top level wrapper, as otherwise it would not be possible to check their values. For the IFU and decoder testing was done to ensure that the decoder could accept the instructions provided by the IFU, and that the timing would line up correctly to still output valid results. For the MAU, WBU, and REGFILE, it was imperative to determine that the data coming from the MAU would be correctly forwarded to the WBU and that data would be properly stored within the regfile. This was especially important since one of the potential problems with the REGFILE discovered through close reading of the code was the existence of the '$cycle\_cnt$' variable. Multiple output signals from the EXU and WBU were responsible for driving certain inputs of the REGFILE, and by connecting these modules and feeding in the expected outputs from the EXU we were able to confirm the ability of the WBU to override those signals and that the REGFILE could process and update the internal registers.

Finally we had to test the top module. Unfortunately for us, '*v_connector.py*' did not work for this, as due the the naming convention provided to us, too many modules used the same names for things such as enable signals, despite different origin points. This made it difficult for '*v_connector.py*' to handle the port connections, and so it was used preliminarily and then edited and fixed manually. In addition to this the most surprisingly difficult part of the analysis was generating our expected results. This is because of the way that Python handles strings of binary and hexadecimal numbers. When python interacts with a binary or hexadecimal number, it will interpret it as a literal 'string' which means that math cannot be performed with it. This necessitates the use of conversion to an integer to perform operations with, thus taking away the ability to perform tasks such as sign extensions, which are a core part of many RV32I instructions. While this was not the end of the world it did require us to do more manual checking than we would have liked to do considering how much automation our testbench did have, with automatic timing recognition and random instruction generation. While this was a slight setback we were still able to confirm the validity of each instruction within the core and its outputs, some of which were handled automatically and others which were handled manually.



```
test_R passed
******************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
******************************************************************************
** test_top_core.test_R           PASS          140.00           0.00      28825.62  **
******************************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                 140.00           0.13       1087.87  **
******************************************************************************
```

Fig. 4. R type instruction test on the top level core wrapper



```
test_I passed
*****************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIM
*****************************************************************
** test_top_core.test_I           PASS          150.00
*****************************************************************
** TESTS=1 PASS=1 FAIL=0 SKIP=0                 150.00
*****************************************************************
```

Fig. 5. I type instruction test on the top level core wrapper

3) Address map and C Compiler: To test real code on our hardware, we aimed to understand the 32I RISC-V specification and utilized the GNU toolchain to generate RISC-V assembly instructions for the 32-bit Integer ISA, based on the 39 base instructions (pure bare-metal C, without system calls). An address map was created as input to the compiler, along with C code, which was converted into assembly instructions. Scripts automated the compilation process, generating stimulus files for Verilog simulations or direct FPGA programming.

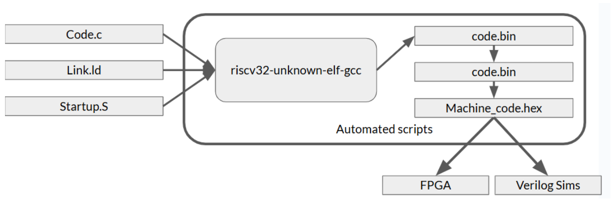
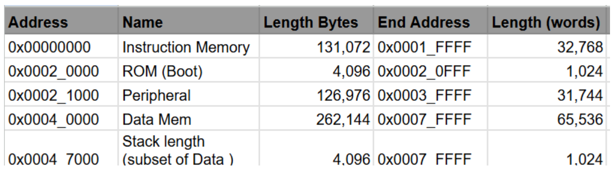


Fig. 6. C code compiler flow

The generated assembly adheres to RISC-V coding conventions for register usage, enabling the compilation and execution of complex C code, including open-source applications, on the designed core. A flow diagram illustrating this process is shown in Figure 6.

B. Emulation Based Verification

We began by exploring the FPGA's capabilities and programming the interaction between the Processing System (PS) and Programmable Logic (PL). BRAM was accessed via Python Pynq API by specifying an address and data (32-64 bits). Two BRAMs of different sizes could be used, with a total FPGA allocation of ½ MB. Initially, BRAM supported unit-level DUT testing: stimulus was written to one BRAM, GPIO signals triggered the test, and results were stored in a second BRAM. However, this was limited to 64K cycles with 32-bit I/O per cycle. To overcome this, BRAM was later used for system-level testing, emulating a memory map adhering to the defined address space as shown in Figure 8.



Address	Name	Length Bytes	End Address	Length (words)
0x00000000	Instruction Memory	131,072	0x0001_FFFF	32,768
0x0002_0000	ROM (Boot)	4,096	0x0002_0FFF	1,024
0x0002_1000	Peripheral	126,976	0x0003_FFFF	31,744
0x0004_0000	Data Mem	262,144	0x0007_FFFF	65,536
0x0004_7000	Stack length (subset of Data)	4,096	0x0007_FFFF	1,024

Fig. 7. Memory Map

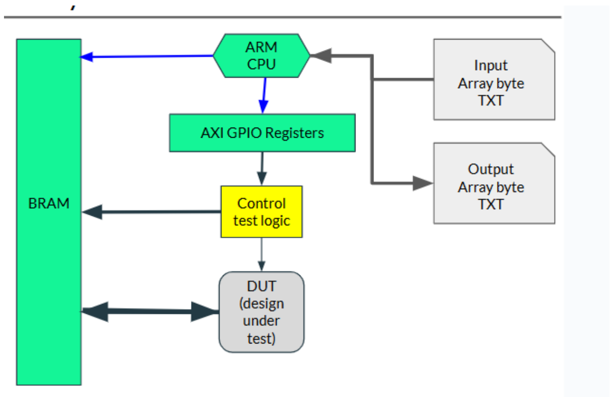


Fig. 8. BRAM use in testing

A script transfers data from the PS to the PL, including the address map contents (instructions and data memory). A GPIO register controlled by PS activates the testbench, and after a set number of cycles, BRAM contents are read to check memory state. Initial tests involved loading data from PS to BRAM , then reading from the PL, full system testing of the core is not yet ready. Unit-level testing was prioritized to enable exhaustive verification. PS-to-PL communication leveraged AXI Stream for high-throughput transfers, supporting up to 64 million cycles with 64-bit I/O signals. Data is sent to DRAM (up to 512 MB), transferred via DMA with AXI Stream to the DUT, which outputs data to a FIFO for the PS to read. A unified implementation diagram is shown in Figure 9.

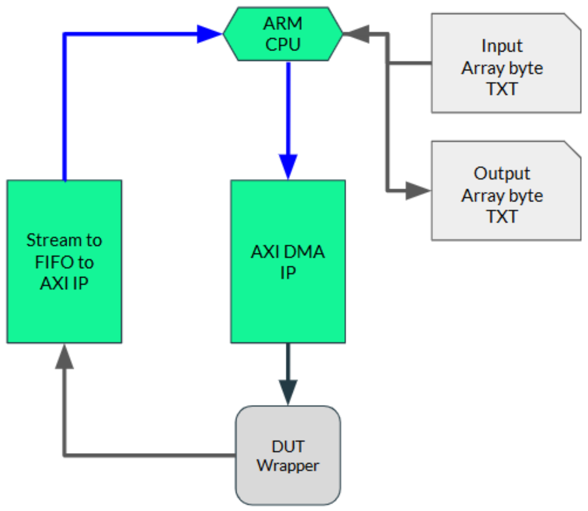


Fig. 9. Implementation Diagram

Logic was added to the DUT to interface with AXI Stream signals and compute one cycle at a time as data is transferred to the PS via FIFO. To ensure the DUT completes a full cycle within one AXI Stream cycle, DUT registers update on the falling edge of the FPGA clock. This restricts FPGA testing to synchronous signals for testbenches, more is shown in Figure 10.

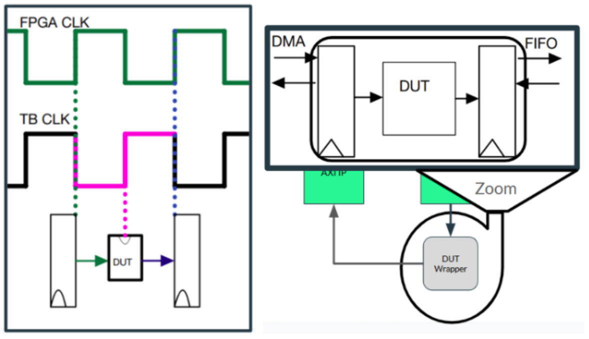


Fig. 10. FPGA clock and interface

Unit-level testing is limited to 64-bit input and output per clock cycle. For modules with larger I/O sizes, data must either be constrained to 64 bits or mapped to a larger size, adding logic overhead. Additionally, significant development time is required to define the 64-bit I/O mapping for each module. Making Unit level testing more of a final step of verifying.

C. UVM Based Verification

1) Verification Algorithm: Verification is used to find bugs in the DUT. This process is usually implemented by putting the DUT into a verification platform. A verification platform should implement the following basic functions:

The verification platform should simulate various real usage situations of the DUT, which means applying various stimuli to the DUT, including normal stimuli and abnormal stimuli. The driver achieves the generation of stimuli.

Additionally, the verification platform should be able to judge whether the behavior of the DUT is consistent with expectations based on the output of the DUT. This part is done by the scoreboard.The verification platform should collect the output of the DUT and pass it to the scoreboard. Here, the monitor completes this function.At the same time, the verification platform should be able to give the expected results. The judgment criteria are mentioned in the scoreboard, and the judgment criteria are usually expectations. In the verification platform, the reference model will compare both the result from the DUT and the scoreboard and give the comparison result.

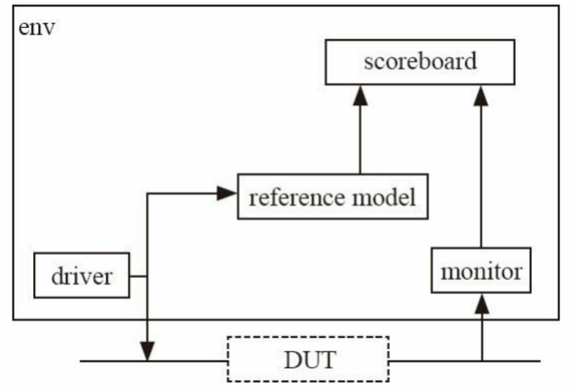


Fig. 11. UVM structure

2) Module Verification: The verification effort focused on applying UVM-based methodologies to ensure protocol compliance and functional correctness of an AHB-APB bridge and its connected peripherals, such as a UART module. Starting from a refined AMBA-based code aligned with standardized ARM specifications, the verification team developed a UVM environment that thoroughly exercises the bridge's functionality. The bridge is responsible for translating high-speed Advanced High-performance Bus (AHB) transactions into lower-speed Advanced Peripheral Bus (APB) operations, enabling communication to devices like UARTs. Verification scenarios included both single and burst read/write patterns to stress the interface under a variety of conditions. By closely examining complex state transitions—from the idle state (ST_IDLE) through read/write states $(ST\_READ/WRITE)$ and into enable states $(ST\_WENABLE)$—the team validated the proper sequencing and timing of control signals, ensuring that parameters such as HTRANS, HWRITE, and Valid bits followed the specified AMBA protocol. Detailed state diagrams guided this process, providing a visual representation of critical paths and decision points in the bridge logic. Ultimately, the UVM-based verification campaign delivered greater confidence that the AHB-APB bridge and associated UART modules meet both the functional and protocol-level requirements set forth by the ARM specifications.

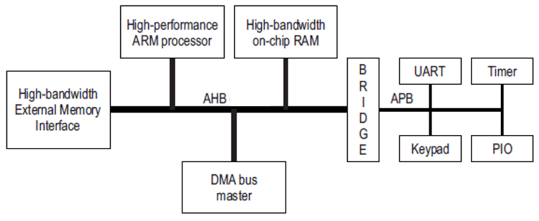

High-performance ARM processor
High-bandwidth on-chip RAM
High-bandwidth External Memory Interface
AHB
BRIDGE
UART
Timer
APB
Keypad
PIO
DMA bus master

Fig. 12. AHB to APB Bridge connections

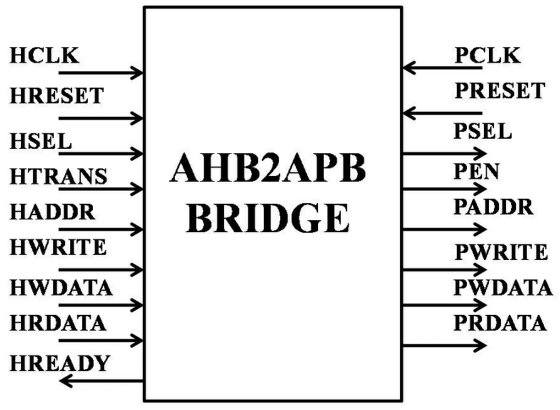


Fig. 13. AHB to APB Bridge inputs and outputs

By building upon the existing AMBA infrastructure, the integration of a UART into the APB domain broadens the communication capabilities of the system, bridging high-speed and lower-power peripheral interfaces while maintaining a consistent protocol environment. Following last semester's initial groundwork, the team leveraged SystemVerilog and UVM testbenches to rigorously verify UART functionality over the APB interface. Key verification objectives included ensuring that UART transmissions and receptions adhered strictly to timing and protocol specifications, validating that the APB read/write transfers mapped correctly to UART registers, and confirming that control signals were properly synchronized across clock domains. Moreover, the verification environment was designed to scale and evolve, allowing the introduction of additional peripherals and communication modules with minimal reconfiguration. This comprehensive approach, spanning from the AHB-to-APB bridge testing to the end-point UART device, established a robust verification framework that could readily be extended to other RISC-V subsystem components in future development cycles.

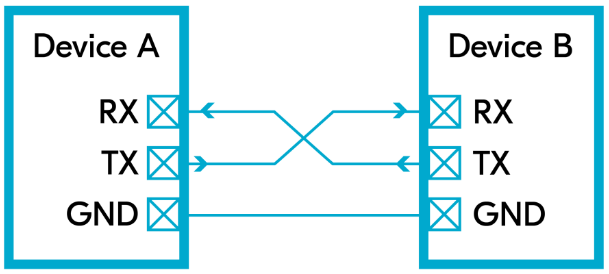


Fig. 14. Bridge interactions

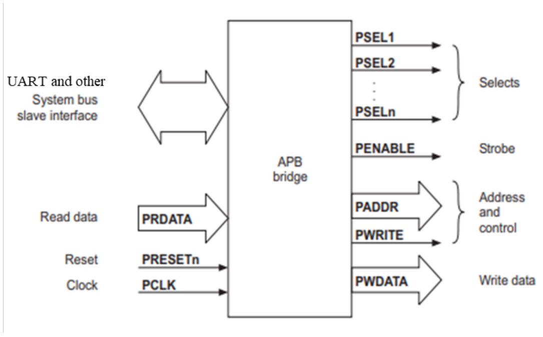


Fig. 15. APB Bridge inputs and outputs

III. Results

From simulation, we were able to identify some problems with the current core. These problems can be placed into three categories: optimization, coding convention, and functionality.

A. Optimization

One feature of the core that we found to be an issue was the deeply ingrained cycle count. To ensure that different modules do not send overlapping commands and interfere with one another, each action is tied to its own "cycle count" that it can act on. While this does help prevent interference and ensure successful timing, it makes pipelining significantly delayed. This issue was not one that we could fix in this semester, as removing it alters the functionality of the entire system. Thus this was a large part of the inspiration behind our next steps: we want to create a more efficient core that would allow for faster operations by removing the cycle count, as it often creates large delays and doesn't allow for much pipelining. This can easily be seen with the IFU which is unable to load new instructions until the EXU is finished and gives it the go ahead. In addition to this, since loads and stores go through the MAU and WBU the solution for this in the core is to flush any other data that is within the system while this is happening. That means that if a new instruction has begun loading into the system, when the load comes through, that instruction will be completely lost with no recovery.

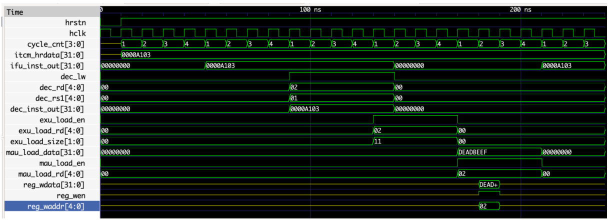


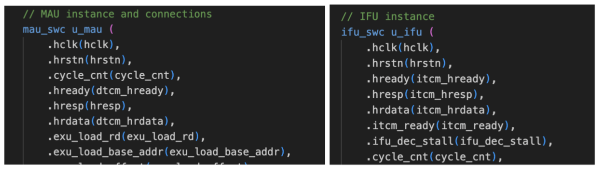
Fig. 16. Cycle Count Delay Test

B. Functionality

Thanks to our randomized testing suite, by running through many iterations we were able to identify edge case errors in the functionality of the previous core. We found data value size errors, incorrect sign extensions, and calculations that were not adhering to the RV32I standards. We were able to both find and correct these issues, resulting in a core better adhering to RISC-V.

C. Coding Convention

We also discovered a lot of inconsistencies in the coding convention, which made themselves particularly apparent when trying to write automated verification code for our testing suite. Modules and signals were named in ways that are misleading and inconsistent, making it difficult to interpret the code. This is something that the following team should keep in mind moving forward.



```
// MAU instance and connections
mau_swc u_mau (
    .hclk(hclk),
    .hrstn(hrstn),
    .cycle_cnt(cycle_cnt),
    .hready(dtcm_hready),
    .hresp(hresp),
    .hrdata(dtcm_hrdata),
    .exu_load_rd(exu_load_rd),
    .exu_load_base_addr(exu_load_base_addr),
```

```
// IFU instance
ifu_swc u_ifu (
    .hclk(hclk),
    .hrstn(hrstn),
    .hready(itcm_hready),
    .hresp(itcm_hresp),
    .hrdata(itcm_hrdata),
    .itcm_ready(itcm_ready),
    .ifu_dec_stall(ifu_dec_stall),
    .cycle_cnt(cycle_cnt),
```

Fig. 17. Naming Convention Inconsistencies

D. SystemVerilog CRT Simulation

This is the UART simulation result. The captured waveform shows that the UART transmitter $(uart\_tx)$ and receiver $(uart\_rx)$ both operate correctly under a 50 MHz clock and 19,200 baud rate. After the transmitter receives a start signal and valid tx_data_in, it sends a start bit (logic low), the proper sequence of 8 data bits, and then a stop bit (logic high). Throughout this process, the tx_active and $done\_tx$ signals behave as expected, confirming accurate timing and proper completion of data transfer.

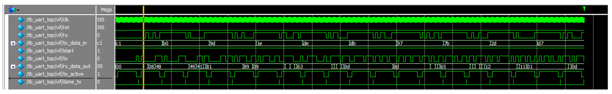


Fig. 18. SystemVerilog simulation results

On the receiver side, the rx line's start bit is detected, each data bit is sampled at the configured baud rate intervals, and the final byte is asserted on rx_data_out once the stop bit is encountered. The observed waveform data aligns with the transmitted message, indicating that the receiver's state machine successfully synchronized to the incoming data stream, reconstructed the 8-bit value, and provided it to the testbench variables.

1) UVM exhausting test: The UVM environment is set up to perform exhaustive testing to ensure the AHB to APB bridge is thoroughly tested. First, we test the single write transfer and read transfer, as shown in Figure 19.

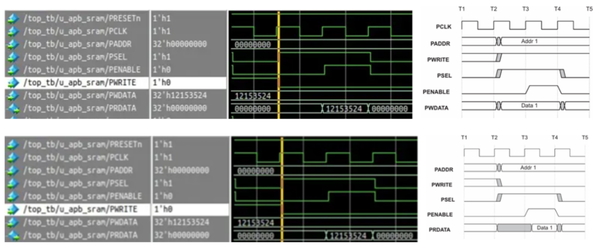
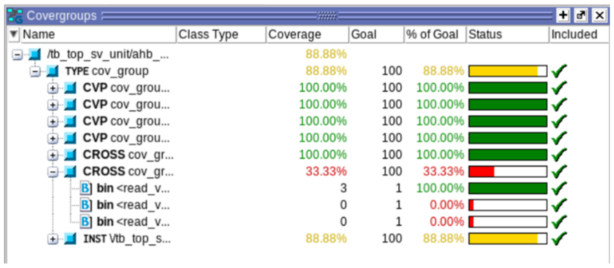


Fig. 19. UVM test waveforms

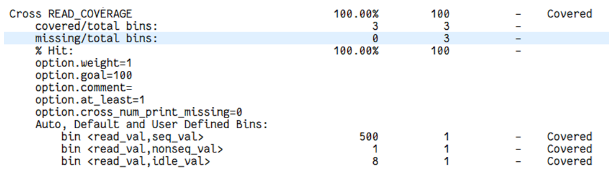
Once the single read and write transfers have been correctly tested, we move on to burst transfer, which involves a single address followed by multiple data transfers. We transfer 500 data items at a time, generate predictions based on the instructions, and compare them with the results. Then, the covergroup is used to collect all the results and check the coverage. We received 100% coverage, indicating that the burst transfer has been fully tested, as shown in Figures 20 and 21.



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/tb_top_sv_unit/ahb_...		88.88%				
TYPE cov_group		88.88%	100	88.88%		✓
CVP cov_grou...		100.00%	100	100.00%		✓
CVP cov_grou...		100.00%	100	100.00%		✓
CVP cov_grou...		100.00%	100	100.00%		✓
CVP cov_grou...		100.00%	100	100.00%		✓
CROSS cov_gr...		100.00%	100	100.00%		✓
CROSS cov_gr...		33.33%	100	33.33%		✓
bin <read_v...		3	1	100.00%		✓
bin <read_v...		0	1	0.00%		✓
bin <read_v...		0	1	0.00%		✓
INST Vtb_top_s...		88.88%	100	88.88%		✓

Fig. 20. UVM Covergroups



```
Cross READ_COVERAGE                                   100.00%        100           -    Covered
    covered/total bins:                                     3          3           -
    missing/total bins:                                     0          3           -
    % Hit:                                            100.00%        100           -
    option.weight=1
    option.goal=100
    option.comment=
    option.at_least=1
    option.cross_num_print_missing=0
    Auto, Default and User Defined Bins:
        bin <read_val,seq_val>                            500          1           -    Covered
        bin <read_val,nonseq_val>                           1          1           -    Covered
        bin <read_val,idle_val>                             8          1           -    Covered
```

Fig. 21. UVM read coverage

IV. Discussion

With regards to the future and results of the project, we have determined that our next step is to either adjust or rebuild the core, with a design focus in mind. While we are unsure what we want this focus to be exactly, we believe that the core should have a specialty as opposed to being a basic implementation. This includes adding pipelining capability as well as the capacity to handle CSRs and interrupts. This would enable the handling of the base 47 instructions that are part of the privileged ISA, while our core currently only handles the base set of 40 instructions from the unprivileged ISA.. We would also like to do away with things such as the '$cycle\_cnt'$ variable as it only provides easy timing and does not benefit the core design otherwise.

The creation of the testing and verification suite, will enable this testing process to be much smoother however, and in the long run will save us time from writing testbenches in Verilog. Connecting the top level modules together and randomly generated instructions will help us to identify which instructions work, and which do not which will help us adjust our core to be, so that it follows RV32I compliance. Ultimately we have developed a tool that helps improve and simplify the design process, streamlining it to improve efficiency of testing. This will not only benefit our team in the future, but also any other group at Rice, or elsewhere who is seeking to have a simple and reliable method for verification of RISC-V compliance with regards to the RV32I ISA.

Acknowledgment

We would like to express our gratitude towards Professor Ooi and Professor Young for their support of our project, as well as the previous team who created the core that we used when developing our testing platform. Their hard work and the support from our professors made this project possible in the first place.

References

[1] Ankitha, A., & Aradhya, H. V. R. (2021). A Python based Design Verification Methodology. Journal of University of Shanghai for Science and Technology, 23(06), 901–911. https://doi.org/10.51201/jusst/21/05358

[2] "Library Reference — cocotb 1.1 documentation," Cocotb.org, 2014. https://docs.cocotb.org/en/v1.2.0/library_reference.html#interacting-with-the-simulator (accessed Oct. 13, 2024).

[3] The RISC-V Instruction Set Manual Volume IbUnprivileged Architecture, Version 20240411, Google Docs. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

[4] Microchip Technology Inc., PIC32 Family Reference Manual, Section 21: UART, Document No. DS60001107H, 2007–2017 https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/ReferenceManuals/60001107H.pdf.[Accessed: Dec.06,2024].