Design Process and Progress Report of a 5-stage 32 bits Pipeline RISC-V Processor Based on RV32I

*

1st Yufei Gu
Department of Electrical and Computer Engineering
Rice University, Houston, Texas, USA
yg77@rice.edu

2nd Jie Gao
Department of Electrical and Computer Engineering
Rice University, Houston, Texas, USA
jg122@rice.edu

3rd Jikun Lu
Department of Electrical and Computer Engineering
Rice University, Houston, Texas, USA
jl287@rice.edu

4th Karan Venaik
Department of Electrical and Computer Engineering
Rice University, Houston, Texas, USA
kv25@rice.edu

5th Luxi Zhu
Department of Electrical and Computer Engineering
Rice University, Houston, Texas, USA
lz70@rice.edu

***Abstract*—This paper details the development of a 32-bit processor based on the RISC-V RV32I instruction set architecture. Our core design features a 5-stage pipeline, comprising Instruction Fetch, Instruction Decode, Execute, Memory Access, and Register Write Back stages and supports data forwarding. Regarding the peripheral component. it supports SPI and UART interfaces, integrated through AHB and APB protocols of the AMBA interface with the core. We have completed all coding aspects of the design and conducted preliminary testing. Entire design was independently completed by our team, without using any third-party IP modules, such as FIFO. The detailed process and further optimizations of our design are elaborated in the following sections.**

***Index Terms*—5-Stage pipeline, RISC-V RV32I, Data forwarding**

I. Introduction

Reduced Instruction Set Computing (RISC) architectures, especially Advanced RISC Machines (ARM) Instruction Set Architectures (ISA), dominate mobile and embedded computing due to their high performance-per-watt, affordability, and licensable nature. These energy-efficient systems are increasingly used in both consumer and enterprise computing. However, ARM ISA's cost limits academic research, unlike RISC-V's open, royalty-free ISA which encourages academic innovation. Especially in Computer Vision, Machine Learning, and Artificial Intelligence, where specialized processors with custom Intellectual Property (IP) blocks outperform traditional architectures, the need for computation-heavy tasks is leading to the development of custom cores and IPs. Universities are now designing custom Application Specific Integrated Circuits (ASICs) for high-performance tasks without heavy investments, thanks to RISC-V's open platform.

The presence of a simplified, integer-only subset of the RISC-V ISA along with the proliferation of open-source front-end processor design and verification tools allow graduate students to quickly experiment on, design, implement, and practice fully functioning cores. As such, the authors of this paper envisioned and implemented the front-end, Register-Transfer Level (RTL) design of a canonical, 5-stage pipelined processor that realizes the RV32I ISA. The processor is aimed as a general-purpose embedded application processor and thus requires integrated interface support, such as SPI, UART, etc. for ADC/DAC and other peripheral control. The aim is to provide a synthesizable RTL core with a fully validated module-level/unit-test level design. As a side-effect of this project, the team aims to explore the challenges and complexities of independent ASIC IP development.

II. Methods

A. Design Philosophy

The team aimed to complete the front-end RTL design and module-level testing of an RV32I-compliant processor. In addition, the team wanted to experiment with the implementation of ARM AMBA specification-compliant interfaces, interconnect, and controllers, that facilitate high-performance data transfer between master devices such as compute blocks, and slave devices such as the Instruction-Tightly Coupled Memory (ITCM) and Data-Tightly Coupled Memory (DTCM). Lastly, the team intended to implement peripheral blocks and controllers to further gain real-world experience.

The design direction was top-down, emphasizing system definition and specification, description of block interlinks, and module behavior first. These were converted into behavioral flow, procedural, or finite state descriptions (or a combination). Next, the RTL design of all blocks was conducted in Verilog with an emphasis on adherence to predetermined finite state machines. Lastly, the blocks were tested individually in Verilog using test benches tailored towards each block, either generated by hand or simple scripts.

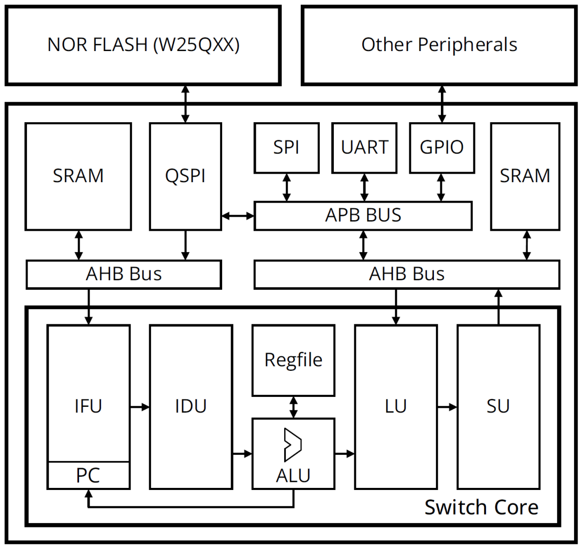


Fig. 1. Overall Framework

B. Overall Design

As shown in Figure 1, the design of this MCU includes two parts: core and peripherals. The core has a 5-stage pipeline whose functions are fetching, decoding, execution, memory access and writeback. Peripherals include simple GPIO as well as SPI and UART, etc.

C. Core Design

The core design contains several parts.

1) chip_top: This module serves as the top-level file for the core, interfacing the core's internals with the data and instruction buses. It encompasses the core's 5-stage pipeline. Within the module, two instances of sram_swc are created. One instance, named itcm_swc_inst, is used as the instruction memory, while the other, named dtcm_swc_inst, serves as the data memory. Both memory blocks have a size of 4096 bytes and are implemented using registers. These memories are connected to the core via the AHB bus, which operates at a frequency of 72MHz, with both address and data widths being 32 bits.

2) ifu_swc: This module is an instruction fetch unit, designed to retrieve instructions from memory based on the address in the PC (Program Counter) register and forward them to the next level of the pipeline. Its clock signal is hclk, and the reset signal is hrstn. The module connects to the memory through the AHB bus and includes pipeline stall control signals to pause this stage of the pipeline when needed. The signal ifu_idle indicates the idle state of the instruction fetch unit. This signal is used to show whether the fetch unit is idle. When it is idle and the cycle_cnt (cycle count) reaches 4, cycle_cnt is reset to 0, allowing the pipeline to proceed to the next cycle. The pc_write and pc_wdata signals control the override of the PC register. When the pc_write signal is active, the PC register is updated with the value from pc_wdata, allowing the execution unit to control the PC register for implementing branch and jump instructions. The inst_out signal is used to output the result of the current fetch to the decoder. The module defines a state machine with four states to control AHB transfers. Among these, the WAIT1 state is designated for waiting for the first 'hready' signal to transfer the address and control signals. Conversely, the WAIT2 state is utilized for waiting for the second 'hready' signal to acquire the required instructions from memory. The transitions between these states are as Figure 2.

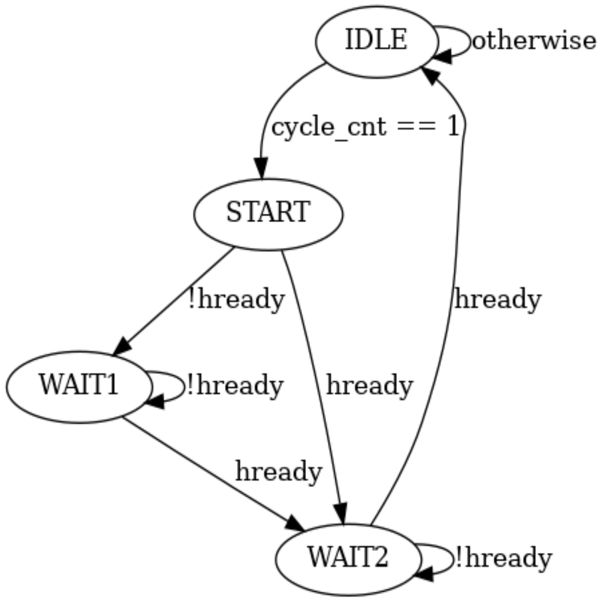


Fig. 2. FSM of IFU

3) dec_swc: This module is the design of the decoder in the pipeline. The signal ifu_dec_stall is an input to this module. When active, this stage of the pipeline will halt operations until the signal is deactivated. Additionally, all signals starting with 'dec', such as dec_lui, dec_auipc, are outputs of this module. These signals, which are only 1 bit, are used to indicate the current decoding instruction. Signals like dec_upper_en and dec_imm_en are used to indicate the type of the current decoding instruction. Signals such as dec_imm_type_i and dec_rd indicate the results of the current decoding for immediate values and registers. This stage of the pipeline employs multi-level pipeline registers to reduce the complexity of combinational logic circuits between registers, thereby decreasing combinational circuit delay and potentially increasing the clock frequency. When cycle_cnt equals 1, the module performs preliminary decoding, decoding the instruction based on the position of bits. When cycle_cnt equals 2, the module performs an intermediate stage of decoding, identifying if the result of the first step matches certain values. When cycle_cnt equals 4, the module outputs the final results based on the outputs of the previous two steps.

4) exu_top_swc: This module is used for the execution of specific instructions. Similar to previous modules, cycle_cnt is a counter used to control specific execution steps, counting from 1 to 4. Ifu_dec_stall is a signal for controlling pipeline stalls, and there are several inputs from the previous level decoder and outputs to the next-level memory access module. The module divides the execution of instructions across multiple sub-modules, grouping similar operations into the same module. For instance, dec_addi and dec_slti, which both read from a register and an immediate value and write to a register, have nearly identical execution steps except for a difference in one of the logical operations in the middle. Therefore, they are implemented in the ex_imm_swc module. Instructions like dec_jal and dec_jalr, which require overriding the program counter and flushing the pipeline, are implemented in the exu_jump_swc module. This module instantiates exu_upper_swc, ex_imm_swc, exu_reg_swc, exu_jump_swc, exu_branch_swc, exu_load_swc, exu_store_swc, and ex_flush_swc. The first seven modules are used for executing the forty-plus instructions in RV32I, while ex_flush_swc is used for controlling the flushing of the pipeline.

5) regfile_swc: This module is registers and its control logics, containing a total of 32 registers, ranging from reg[0] to reg[31], with each register being 32 bits wide. The module provides interfaces for reading from and writing to these registers. It features one write port and two read ports to prevent structural hazards associated with the read ports. It's important to note that the 0th register of this register array is hardwired to ground, meaning that data cannot be written to register 0, and its read value will always be 0.

6) mau_swc: This module is the memory access module, which is the fourth level of the core pipeline. Based on the output control signals of the previous execution unit, it reads or writes to the memory through the AHB protocol and outputs the results to the next writeback module. In each pipeline cycle, this module monitors two signals: exu_load_en and exu_load_store_en. If either of these signals is active, a load or store operation is initiated. These operations are implemented through a state machine. During a load operation, the states READ_WAIT2 and READ_WAIT1 are encountered, which are used for waiting for the transmission address and control signals, and for obtaining data, respectively. During a store operation, the WRITE_WAIT state is needed to wait for the hready signal, which indicates that data can be written to the specified address. The transitions of this state machine are shown in Fig. 3.

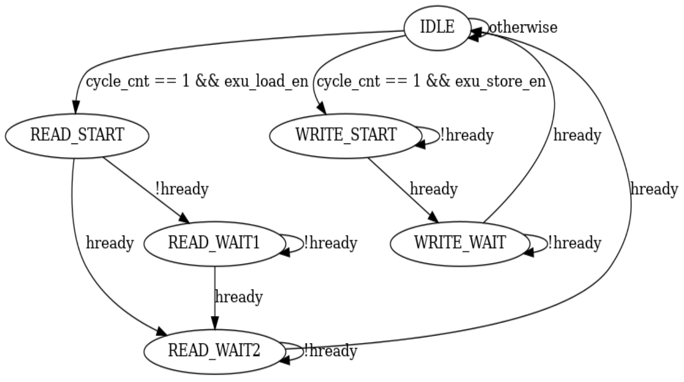


Fig. 3. FSM of MAU

D. AMBA Design

1) Overall Design of AMBA: In our switch MCU's AMBA setup for the peripheral section, the design includes the AHB Master, integrated within the core for handling data transfers on the Advanced High-performance Bus (AHB), and the AHB to APB Bridge, which consists of an AHB Slave and an APB Master for facilitating communication between the AHB and Advanced Peripheral Bus (APB). Additionally, there's an APB Slave used for connecting peripherals such as UART and SPI, acting as a communication medium between these devices and the bus system.our overall design for this part is shown in Fig. 4. The AMBA connection to the SRAM comprises key components, including the AHB Master, which functions similarly to the peripheral section, and the AHB Slave, situated in the SRAM's backup plan version as controller. The AHB Slave is responsible for responding to the AHB Master's requests and handles the reading and writing processes to the SRAM.The process is shown in the Fig. 5.

2) Essential Signals of AMBA in our design: In the AHB protocol, several signals play crucial roles. HADDR[31:0] serves as the address bus, specifying the address for read or write operations, with a 32-bit width for a large address space. HMASTLOCK, the master lock signal, is vital for forming a locked sequence of transfers, especially in read-modify-write operations where uninterrupted access is needed. HPROT[6:0] indicates the protection level of a transfer, providing details about the transaction type, such as its cacheability, bufferability, and privilege level. HSIZE[2:0] specifies the transfer size, ranging from byte to word sizes. HTRANS[1:0] identifies the type of transfer, such as non-sequential, sequential, idle, or burst. HWDATA[31:0] is the write data bus, carrying data for write transactions, while HWRITE signifies whether a transaction is a read or write. Additional signals like WBUFFREAD, RBUFFREAD, RBUFFWRITE, and RBUFFDATA[31:0] are used for

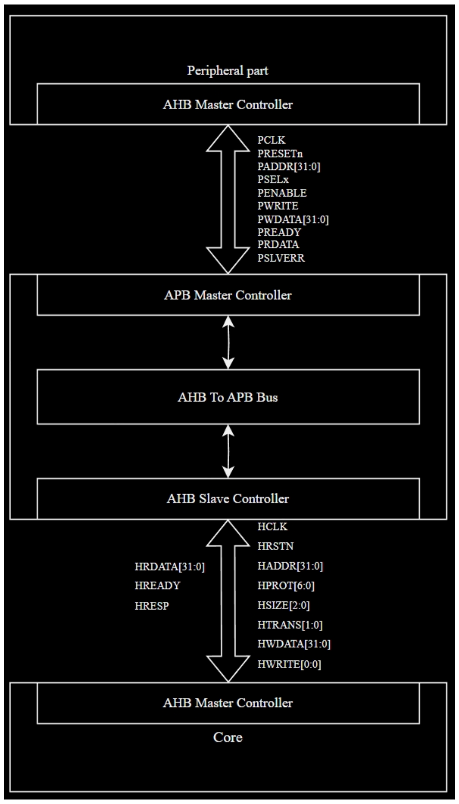


Fig. 4. AMBA Peripheral Overall Design

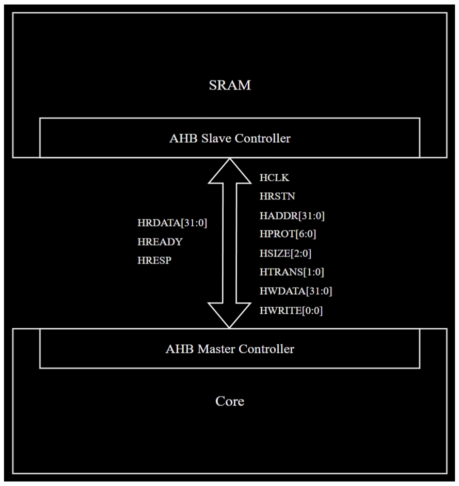


Fig. 5. AMBA SRAM Overall Design

debugging purposes. Lastly, DONE signals the completion of a transaction, and RESP represents the transaction's response or status.For detailed information on the signal refer to AMBA's AHB manual [1]. As for the APB protocol [2]. PCLK, the Peripheral Clock, synchronizes transactions on the APB, aligning them with its rising edge. PRESETn, active low, serves as the Peripheral Reset signal, resetting peripherals on the APB when asserted. PADDR[31:0], a 32-bit Peripheral Address Bus, addresses peripheral registers, allowing unique access to each connected peripheral. PSELx, the Peripheral Select signal, chooses a specific peripheral for communication, with each having an associated PSELx. PENABLE, the Peripheral Enable signal, enables data transfers to and from the selected peripheral, indicating active read or write operations when high. PWRITE signifies the data transfer direction, with high for write operations and low for reads. PWDATA[31:0] is the 32-bit Peripheral Write Data Bus for sending data during write operations. PREADY, used by the peripheral, indicates the completion of data transfers, signifying the validity of data on PRDATA (for reads) or acceptance on PWDATA (for writes). PRDATA, the Peripheral Read Data Bus, carries data read from peripherals. Lastly, PSLVERR, the Peripheral Slave Error signal, indicates transaction errors, with a high state signaling an error during data transfer.

3) Module Design of AMBA: From the preceding description of AMBA's overall design, it's evident that AMBA's module design can be divided into several parts, namely AHB Master, AHB Slave, APB Master, and APB Slave.
For AHB Master,the state machine is as shown in the Fig. 6

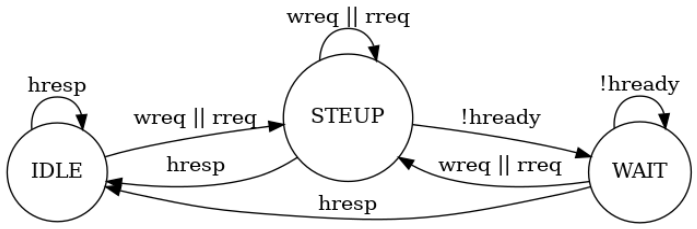


Fig. 6. FSM of diagram of AHB master

When at IDLE state the module is in a passive mode, waiting for a signal to initiate an action. It represents a standby phase, ready to transition based on incoming read or write requests. The STEUP state is where the module prepares for a data transfer, setting up necessary parameters for either a read or write operation. It acts as a brief preparatory phase before actual data handling begins. The WAIT state indicates active engagement in data transfer, either reading from or writing to a target. The module remains in this state until the transfer is complete or another request is received, ensuring continuous data handling.
The above diagram describe the power supply topology.
For AHB Slave,the state machine is as shown in the Fig. 7

For the core, all states beginning with 'R', such as RDONE, RWAIT, RSTART, are associated with read operations. Conversely, states like WDONE, WWAIT, WPAUSE, and WSTART are indicative of write operation states. Within this framework, the states START, PAUSE, WAIT, and DONE represent sequential stages in a data transfer operation. START is the initiation stage, where initial conditions are

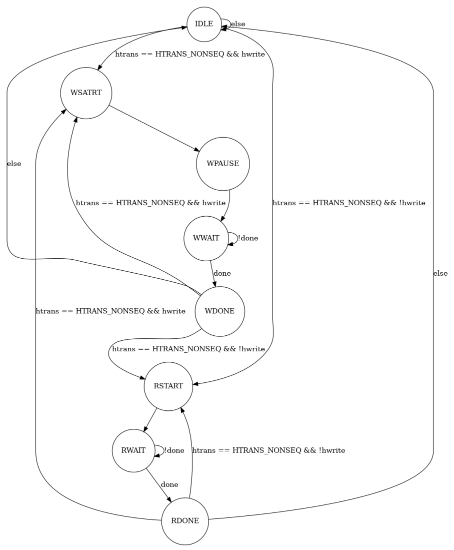

IDLE
else
htrans == HTRANS_NONSEQ && hwrite
WSATRT
WPAUSE
else
htrans == HTRANS_NONSEQ && hwrite
htrans == HTRANS_NONSEQ && !hwrite
WWAIT
!done
done
else
htrans == HTRANS_NONSEQ && hwrite
WDONE
htrans == HTRANS_NONSEQ && !hwrite
RSTART
RWAIT
!done
htrans == HTRANS_NONSEQ && !hwrite
done
RDONE

Fig. 7. FSM of diagram of AHB slave

set up. PAUSE acts as an intermediate stage, potentially for buffering or preparing for the subsequent phase of the operation. WAIT involves the state machine in a holding pattern, awaiting certain conditions or responses before moving forward. DONE signifies the completion of the operation, marking the successful conclusion of either a read or write process. IDLE represents the initial or idle state of the system.
For APB Master,the state machine is as shown in the Fig. 8

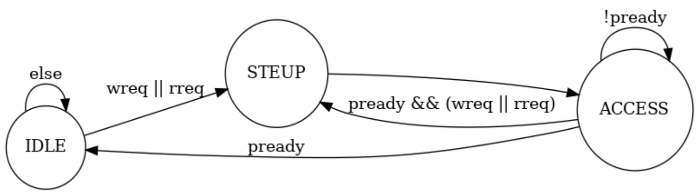


Fig. 8. FSM of diagram of APB master

IDLE is the default state, where it waits for either a read (rreq) or write (wreq) request. Upon receiving a request, it transitions to the STEUP state. STEUP is a preparatory state for APB access, setting up necessary parameters like the address (paddr), operation type (pwrite), and buffer read signals (wbuffread for writes, rbuffread for reads). After these preparations, it advances to the ACCESS state. ACCESS is a state that manages data transfers with the APB. It either sends data (pwdata) for write operations or retrieves data (prdata) for read operations. The module stays in this state until the transfer is complete (pready). After completion, it either cycles back to STEUP or returns to IDLE if there are no more requests.
For APB Slave,the state machine is as shown in the Fig. 9

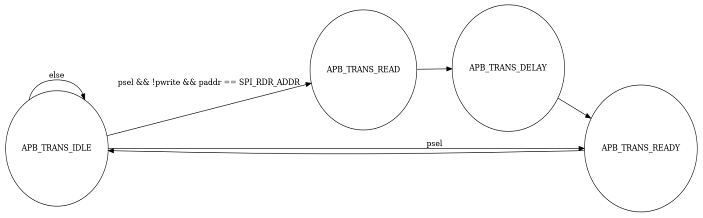


Fig. 9. FSM of diagram of APB slave

APB TRANS IDLE is the default state. When in APB TRANS IDLE, the state machine waits for an APB bus select signal (psel). If a read request for the SPI Read Data Register (SPI RDR ADDR) is detected, it transitions to APB TRANS READ. For any other select signal, it moves to APB TRANS READY.APB TRANS READY is a transitional state indicating that the module is ready to proceed with the next operation. After the necessary conditions are met, the state machine returns to APB TRANS IDLE.APB TRANS READ is a stage that the module prepares to read data from the SPI. It then transitions to APB TRANS DELAY for further processing.APB TRANS DELAY is a state appears to add a delay or buffer phase before transitioning back to APB TRANS READY, is for synchronize the read operation.

E. Peripheral Design

1) Overview of peripheral we have support: Our design objective is to support UART and SPI for our peripherals, which are used to connect devices like ADCs and other external peripherals. We've made some simplifications compared to the original port definitions. For instance, in the SPI module, we have eliminated the SS (Slave Select) signal, which is typically used for device selection. Our overall design draws inspiration from the STM32's STM32F102 and involves defining corresponding registers. For detailed functionalities, one can refer to the Switch_Microcontroller_Spec_Preview.md file in our repository, which includes supported features, and definitions for SPI control and status registers.
Similarly, our UART design also references the STM32F102 manual but is somewhat simplified. The STM32 uses USART, which includes clock synchronization - essentially an augmentation to UART. Our design is intended to support standard UART communication. Unlike the STM32F107, it omits clock synchronization and some aspects of verification. The specific manual definitions can also be found in the Switch_Microcontroller_Spec_Preview.md file.
For official references on SPI [3] and UART [4], can be found in the reference selection.

2) Design of the peripheral: Firstly, our design, whether it's for UART or SPI, is essentially a modification of the APB (Advanced Peripheral Bus) Slave component, which is utilized as the controller. The design of the SPI controller is illustrated in the Fig. 10. The design for 'Format Convert' is a FIFO (First In, First Out) buffer. We opted for a synchronous FIFO in our design, rather than an asynchronous FIFO, and incorporated a counter design to facilitate the alteration of the clock rate during transmission. RDR (Read Data Register) and WDR (Write Data Register) serve as our read and write registers. The shifting of data to MOSI (Master Out Slave In) and MISO (Master In Slave Out) is achieved using a state machine. The details and workings of the SPI state machine will be explained later.

For our design the spictrl swc module of the Verilog code

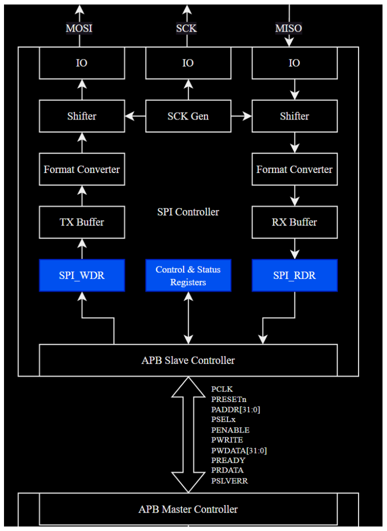


Fig. 10. SPI overall digram

manages the SPI (Serial Peripheral Interface) communication process and the FSM diagram for the SPI communication,the diagram is shown in Fig. 11. In the SPI TRAN IDLE state, the FSM waits for the Write Data Register (WDR) to be non-empty and the SPI Control Register (SPI CR) to enable SPI communication. If these conditions are met, it transitions to SPI TRANS READ, where it starts the read process from the WDR. Next, it moves to SPI TRANS DELAY, which seems to serve as a buffer or delay stage before proceeding. Following this, the SPI TRANS BUFFER state buffers the data for SPI transmission. The SPI TRANS COUNT state involves a counter to manage the timing of the SPI communication. If the counter reaches a maximum value (maxcnt), the state transitions to SPI TRANS WRITE, where the buffered data is written to the SPI bus. If the WDR is not empty after writing, it goes back to SPI TRANS READ to continue processing data; otherwise, it returns to SPI TRANS IDLE. This FSM effectively manages the data flow for SPI communication, ensuring data is read from the WDR, buffered, and then written to the SPI bus in a controlled, sequential manner. The data in the WDR (Write Data Register) comes from a FIFO (First In, First Out) buffer. The FIFO facilitates the transfer of data from the APB (Advanced Peripheral Bus) Slave module to the WDR. The design of the entire SPI (Serial Peripheral Interface) controller is based on the code of the APB Slave. The results will be elaborated in detail in the Test section.

As for the overall design of the UART module, it is similar to

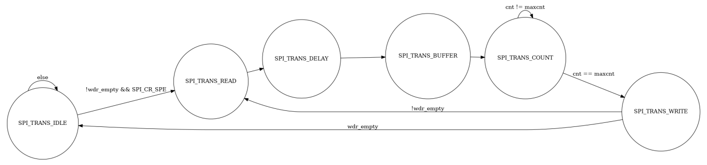


Fig. 11. Diagram for SPI FSM

SPI, with roughly the same components. However, there are subtle differences, mainly reflected in the design of the state machine and the use of a counter divider for differentiation. The design of our UART's state machine is as shown in the following diagram. The design of the state machine differs significantly from that of the SPI section.

The UART state machine Fig. 12within the uart_swc module functions to manage the reception of data through the UART protocol. In the UART_RX_IDLE state, the system idles, awaiting a negative edge on the RX line and checking if the UART control register 1 (UART_CR1_UE) is enabled, indicating that the UART is active. Upon detecting these conditions, it transitions to the UART_RX_COUNT state. In this state, the machine counts the received bits and checks if the total number of bits received (including start, data, parity, and stop bits) matches the expected number based on the UART configuration. If the count is complete, it moves to the UART_RX_WRITE state, where the received byte is written to the UART Receive Data Register (UART_RDR). In case the count is not complete, it remains in the UART_RX_COUNT state to continue receiving the rest of the bits. After writing the data to the register in the UART_RX_WRITE state, it transitions back to the UART_RX_IDLE state, ready to receive the next byte. This cyclical process ensures continuous and ordered reception of data bytes according to the UART communication protocol.

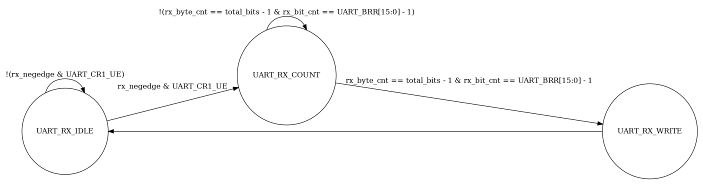

!(rx_byte_cnt == total_bits - 1 & rx_bit_cnt == UART_BRR[15:0] - 1)
!(rx_negedge & UART_CR1_UE)
rx_negedge & UART_CR1_UE
UART_RX_IDLE
UART_RX_COUNT
rx_byte_cnt == total_bits - 1 & rx_bit_cnt == UART_BRR[15:0] - 1
UART_RX_WRITE

Fig. 12. Diagram for UART FSM

F. Design of the SRAM

1) Overview of data sram: Our SRAM module ahbs_dsram_swc, serves as an interface to AHB, designed to facilitate load and store operations. The module employs a state machine structure comprising states IDLE, READ, and WRITE controlled by AHB signals (hresp, hrstn, htrans, hwrite). Upon initialization or reset (hrstn), the state machine defaults to the IDLE state. State transitions are determined by combinations of these signals, directing the module to read or write data from/to the SRAM storage array, data_sram, based on the provided address (haddr) and data (hwdata). During the READ state, the module fetches data from data_sram indexed by the address (haddr), storing it in hrdata. Conversely, in the WRITE state, it stores the incoming hwdata into data_sram at the specified address (haddr). Additionally, the design includes logic to handle different address sizes and protocol signal combinations, defaulting to the IDLE state for unhandled cases. The methodology involved simulation using Verilator, testbench creation, and stimuli generation to validate correct operation, ensuring proper state transitions, address range checks, and error handling. This design approach, delineated in detailed comments within the code, offers a replicable method for implementing an SRAM module interfacing with AHB, ensuring functionality.

2) Overview of instruction sram: Our module ahbs_isram_swc, functions as an interface within the AHB, specifically tailored for managing instruction memory operations. The module aims to facilitate the fetching of instructions from an instruction SRAM (instruction_sram) of adjustable size (INSTRUCTION_MEMORY_SIZE). This design integrates various input signals, including hclk, hrstn, haddr, hmastlock, hprot, hsize, htrans, hwdata, and hwrite, emulating the AHB interface. The core of our design is a state machine structure encompassing two distinct states: IDLE and FETCH. During initialization or upon reset (hrstn), the state machine defaults to the IDLE state. Transitions between states are orchestrated based on specific combinations of AHB signals (hresp, hrstn, htrans, hwrite). In the FETCH state, triggered by specific signal conditions, for example (5'b0_1_10_0), the module extracts instruction data from the instruction_sram array indexed by the provided address (haddr) and outputs the fetched instruction as hrdata. Crucially, the design includes default handling to revert to the IDLE state for unhandled signal combinations.

III. Results

1) Brief Description of Test Environment Setup: We utilized GTKWAVE and Icarus Verilog (iverilog) for setting up our environment, along with tools like Verilator. For details about the environments we used, please refer to the readme.md document on our GitHub. However, the most fundamental tools are GTKWAVE and iverilog. GTKWAVE is a waveform viewer (by the way our team member Yufei developed a similar software), and iverilog is used to generate waveform files. To run iverilog files, our run_simṡh script is required. This script locates all .v files in the directory and executes them. It's important to modify the filename of the top module, otherwise, the script will not run. Detailed information can be found in the AHB APB code documentation.md file.

2) Tests and results for Core: For testing our core components, we need to connect ifu, decoder, mau, regfiles, wbu, and others through core_top_swc for testing. Additionally, our design incorporates pipeline functionality, and we also need to test for Control Hazards associated with the pipeline.

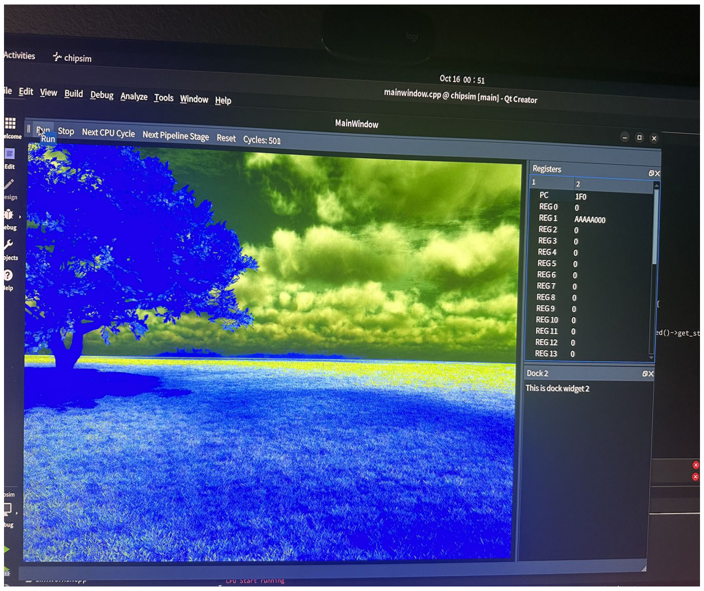
To this end, we have created a brand-new testbench for testing the input of several consecutive instructions and observing the output results. We have also written the connection part with the AMBA bus, transferring data from Regfiles to peripherals and SRAM. Due to development progress, we prepared a simplified (or as we say in Chinese, "Monkey Version") backup version of SRAM for testing data transfer.

After testing, we found that a series of instructions, including J type, I type, S type, R type, U type, B type, showed no issues under the condition of not exceeding ten instructions. Data Hazards and Control Hazards have been resolved and meet requirements. The related test files can be found in the chip folder.

However, our design supports a 5-level pipeline, which is quite complex. Moreover, if compiling high-level languages, a large number of assembly instructions will be input, and the compiler's translation of high-level languages into assembly is completely random. Therefore, we need to conduct extensive testing to see if it meets the requirements.

For the testing of the Core, we have built a graphical test platform using QT, allowing real-time simulation of the core with graphical visualization. It directly reads Regs and also calls related libraries (detailed in our project's readme.md) to view the expected results. The graphical test platform is shown in the following figure. Since it must strictly correspond to the signals of the test module, we added some other signals in the final version. Currently, it needs modifications to run, and this part needs to be changed. Due to the proximity of the report deadline, we only have photos of the previous version without clear screenshots, so the photo is placed here as shown in the 13.

Additionally, we designed a part to judge whether it supports

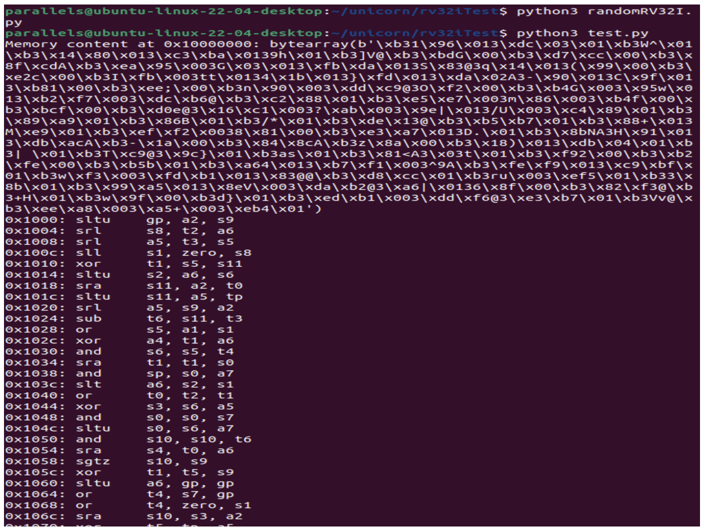

Activities
chipsim
Oct 16 00 : 51
File Edit View Build Debug Analyze Tools Window Help
mainwindow.cpp @ chipsim [main] - Qt Creator
MainWindow
Run Stop Next CPU Cycle Next Pipeline Stage Reset Cycles: 501
Registers
Dock 2
This is dock widget 2

1	2
PC	1F0
REG 0	0
REG 1	AAAAA000
REG 2	0
REG 3	0
REG 4	0
REG 5	0
REG 6	0
REG 7	0
REG 8	0
REG 9	0
REG 10	0
REG 11	0
REG 12	0
REG 13	0

Fig. 13. Photo of the QT core graphical test platform

high-level languages by translating C language into RV32I.

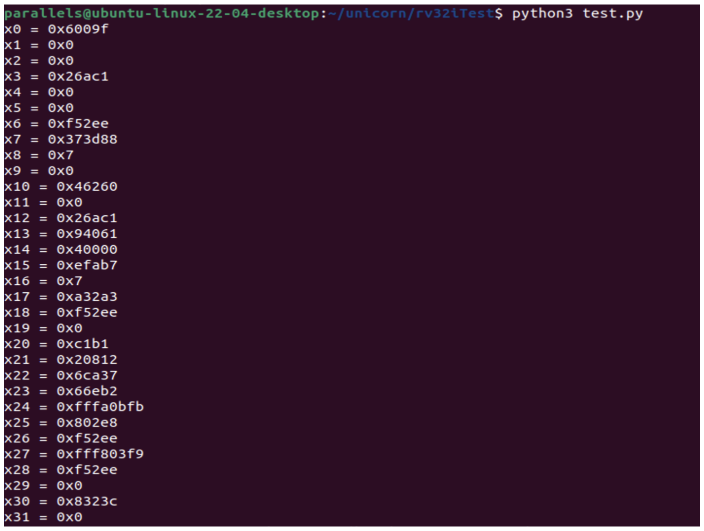
This part is also detailed in the readme document. We have completely run a while loop in the QT visualization platform, and everything worked normally. We wanted to further test the ”Hello world” statement using Printf, but since print requires UART, which was under development at the time, we did not conduct further testing on it.
The results of compiling high-level languages are not sufficient to cover all test results. To further refine our design, we have some additional ideas. We use the Unicorn Engine to read and simulate random instructions generated from python scripts. Then, we put random instructions into SRAM, and then use the existing Capstone library to generate the expected results. Afterward, we will export the results from SRAM and compare them with the expected results to determine if there are any issues with the core.
For this part, data needs to be stored in SRAM. Currently, we only have a backup version of SRAM for testing the AMBA part that is usable when we do the core design. Further modifications are required. Therefore, at this stage, we can use the Unicorn Engine to generate random RV32I instructions. The screenshot of generating instructions is shown in the Fig. 14 and the reg result from the core is shown in the Fig. 15.



```
parallels@ubuntu-linux-22-04-desktop:~/unicorn/rv32iTest$ python3 randomRV32I.
py
parallels@ubuntu-linux-22-04-desktop:~/unicorn/rv32iTest$ python3 test.py
Memory content at 0x10000000: bytearray(b'\xb31\x96\x013\xdc\x03\x01\xb3W^\x01
\xb3\x14\x80\x013\xc3\xba\x0139h\x01\xb3]V@\xb3\xbdG\x00\xb3\xd7\xcc\x00\xb3\x
8f\xcdA\xb3\xea\x95\x003G\x03\x013\xfb\xda\x013S\x83@3q\x14\x013(\x99\x00\xb3\
xe2c\x00\xb3I\xfb\x003tt\x0134\x1b\x013}\xfd\x013\xda\x02A3-\x90\x013C\x9f\x01
3\xb81\x00\xb3\xee;\x00\xb3n\x90\x003\xdd\xc9@3O\xf2\x00\xb3\xb4G\x003\x95w\x0
13\xb2\xf7\x003\xdc\xb6@\xb3\xc2\x88\x01\xb3\xe5\xe7\x003m\x86\x003\xb4f\x00\x
b3\xbcf\x00\xb3\xd0e@3\x16\xc1\x003?\xab\x003\x9e|\x013/U\x003\xc4\x89\x01\xb3
\x89\xa9\x01\xb3\x86B\x01\xb3/*\x01\xb3\xde\x13@\xb3\xb5\xb7\x01\xb3\x88+\x013
M\xe9\x01\xb3\xef\xf2\x0038\x81\x00\xb3\xe3\xa7\x013D.\x01\xb3\x8bNA3H\x91\x01
3\xdb\xacA\xb3-\x1a\x00\xb3\x84\x8cA\xb3z\x8a\x00\xb3\x18)\x013\xdb\x04\x01\xb
3| \x01\xb3T\xc9@3\x9c}\x01\xb3as\x01\xb3\x81<A3\x03t\x01\xb3\xf92\x00\xb3\xb2
\xfe\x00\xb3\xb5b\x01\xb3\xa64\x013\xb7\xf1\x003^9A\xb3\xfe\xf9\x013\xc9\xbf\x
01\xb3w\xf3\x003\xfd\xb1\x013\x83@@\xb3\xd8\xcc\x01\xb3ru\x003\xef5\x01\xb33\x
8b\x01\xb3\x99\xa5\x013\x8eV\x003\xda\xb2@3\xa6|\x0136\x8f\x00\xb3\x82\xf3@\xb
3+H\x01\xb3w\x9f\x00\xb3d}\x01\xb3\xed\xb1\x003\xdd\xf6@3\xe3\xb7\x01\xb3Vv@\x
b3\xee\xa8\x003\xa5+\x003\xeb4\x01')
0x1000: sltu    gp, a2, s9
0x1004: srl     s8, t2, a6
0x1008: srl     a5, t3, s5
0x100c: sll     s1, zero, s8
0x1010: xor     t1, s5, s11
0x1014: sltu    s2, a6, s6
0x1018: sra     s11, a2, t0
0x101c: sltu    s11, a5, tp
0x1020: srl     a5, s9, a2
0x1024: sub     t6, s11, t3
0x1028: or      s5, a1, s1
0x102c: xor     a4, t1, a6
0x1030: and     s6, s5, t4
0x1034: sra     t1, t1, s0
0x1038: and     sp, s0, a7
0x103c: slt     a6, s2, s1
0x1040: or      t0, t2, t1
0x1044: xor     s3, s6, a5
0x1048: and     s0, s0, s7
0x104c: sltu    s0, s6, a7
0x1050: and     s10, s10, t6
0x1054: sra     s4, t0, a6
0x1058: sgtz    s10, s9
0x105c: xor     t1, t5, s9
0x1060: sltu    a6, gp, gp
0x1064: or      t4, s7, gp
0x1068: or      t4, zero, s1
0x106c: sra     s10, s3, a2
```

Fig. 14. Generated RV32I instructions by Unicorn Engine

3) Tests and results for AMBA: For our testing of the AMBA part, we individually tested all modules for both write and read operations, as well as scenarios where write operations are immediately followed by read operations. Additionally, we tested a series of multi-device control signals such as PSEL, referring to the official AMBA manual for guidance and comparison. We compared our actual waveform charts with the expected waveform charts to determine if the results were correct.
Since our design has not undergone synthesis, our simulation is only a behavioral simulation and does not include Static Timing Analysis (STA). The analysis of timing will be a



```
parallels@ubuntu-linux-22-04-desktop:~/unicorn/rv32iTest$ python3 test.py
x0 = 0x6009f
x1 = 0x0
x2 = 0x0
x3 = 0x26ac1
x4 = 0x0
x5 = 0x0
x6 = 0xf52ee
x7 = 0x373d88
x8 = 0x7
x9 = 0x0
x10 = 0x46260
x11 = 0x0
x12 = 0x26ac1
x13 = 0x94061
x14 = 0x40000
x15 = 0xefab7
x16 = 0x7
x17 = 0xa32a3
x18 = 0xf52ee
x19 = 0x0
x20 = 0xc1b1
x21 = 0x20812
x22 = 0x6ca37
x23 = 0x66eb2
x24 = 0xfffa0bfb
x25 = 0x802e8
x26 = 0xf52ee
x27 = 0xfff803f9
x28 = 0xf52ee
x29 = 0x0
x30 = 0x8323c
x31 = 0x0
```

Fig. 15. Data from regfiles

future task. We are aiming for the APB's PCLK to reach an operating frequency of 36MHz, and for the AHB's clock frequency to reach 72MHz.
Taking APB_master and APB_slave as examples it shows the tests we conducted. In the Fig. 16, which includes both reading and writing, our testing showed that the data in wbufferdata could be placed into pwdata, and rbuffer could be placed into prdata. Additionally, the corresponding address was entered into paddr. After testing, we found that this part worked correctly. The pselx signal also met the requirements, selecting the appropriate module. This part of the result is consistent with the APB module signals in the AMBA protocol.
The Fig. 17 shows separate read and write operations. The

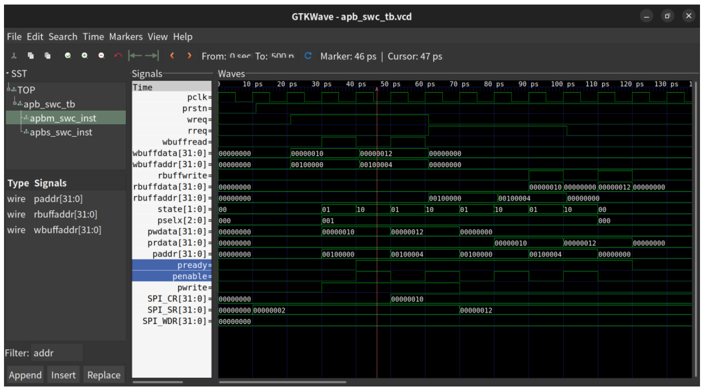


Fig. 16. Test case for APB Slave write right after read

results of this part are the same as Fig. 16 and have also been verified to be correct through testing.
Our AHB_Master, AHB_Slave, APB_Master, and APB_Slave have all undergone such testing. The behavioral tests meet the requirements. We also conducted tests with the SPI peripheral components, and they too showed no issues. Regarding

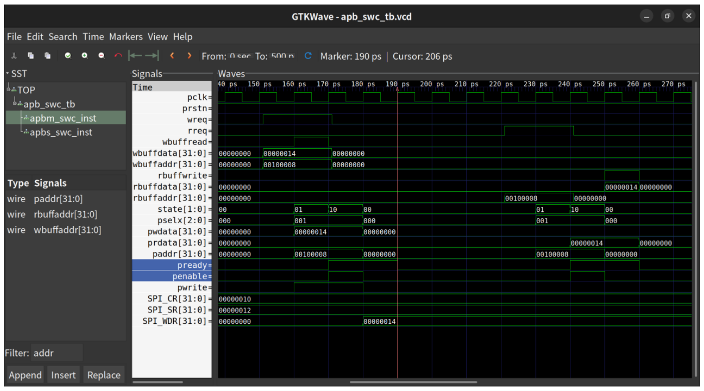
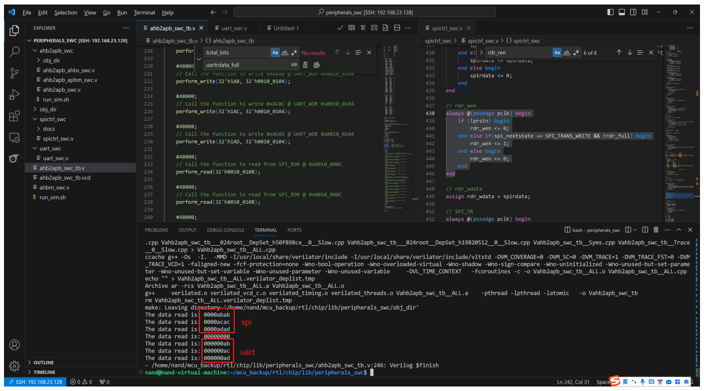


Fig. 17. Test case for APB Slave write seperate with read

SRAM, we tested our backup version of SRAM, and it was able to communicate normally with SRAM. The behavioral simulation of the AMBA part meets the requirements, and all functionalities are working correctly.

4) Tests and results for peripherals: For the SPI part, as it needs to be controlled by the Core, data is input from the Core to MOSI and then the data received from MISO is transmitted back to the Core. At the same time, the status of the SPI port, represented by SPI_SR, also needs to be transmitted to the Core. Moreover, the SPI_CR requires the data to control through the Core. Therefore, for testing SPI and UART, we connect the APB to AHB bridge with our peripheral module modified based on APB_Slave for testing. The testbench is modified based on the ahb2apb.tb file. Regarding the testing of SPI, our idea is to test all functionalities of SPI_CR, including control of the prescaler, switching of CPOL and CPHA, switching of the Data Frame Format (16-bit or 8-bit), as well as enabling SPI and switching between LSB and MSB. Through our testing, all functionalities of SPI_CR have been implemented. The reference waveforms for these functionality tests case can be found in the STM32F107 manual, and the related vcd files can be viewed in the SPI peripheral folder in the github. The testing of SPI_SR has indicated the current state of SPI, such as the Busy flag, Overrun flag, TX enable, and RX enable. These parts represent the read and write FIFO status signals in SPI, and testing has confirmed that these have also been implemented. For data testing, we have set MISO equal to MOSI in the testbench. This way, checking whether the incoming and outgoing data are the same can meet the requirements. Through our testing, the data sent out by the AHB master through the AHB to APB bridge, and then sent out by the SPI controller, is connected back from MOSI to MISO and sent back to the core through the same path. Our design in this part has been verified to be correct.
We conducted similar tests for the UART as well, such as verifying the accuracy of read and write operations. The tests for the divider section also differ from those for SPI. UART supports different input and output Baud rate, and the change in baud rate is not an even multiple, which cannot be resolved through shifting , and we have conducted separate tests for this aspect. Our testing methods and results are placed in the same way as for the SPI section. Through our testing, all functions of UART_CR are operable. We also performed a Loopback test by connecting RX to TX, with data widths of 9 and 8, odd and even parity, LSB, MSB. Tests for different baud rates were also conducted and proven to be functional. Data can be normally transmitted from AHB_Master through the AHB to APB bridge, to the UART, and then sent back along the same path to AHB_Master, with all results being normal.
The test result is shown in the Fig. 18



```
    perfor
    #48000
    // Call the function to write 0xABAB @ UART_WDR 0x0010_0104
    perform_write(32'h1AB, 32'h0010_0104);

    #48000;
    // Call the function to write 0xACAC @ UART_WDR 0x0010_0104
    perform_write(32'h1AC, 32'h0010_0104);

    #48000;
    // Call the function to write 0xADAD @ UART_WDR 0x0010_0104
    perform_write(32'h1AD, 32'h0010_0104);

    #48000;
    // Call the function to read from SPI_RDR @ 0x0010_000C
    perform_read(32'h0010_0108);

    #48000;
    // Call the function to read from SPI_RDR @ 0x0010_000C
    perform_read(32'h0010_0108);

    #48000;
```

```
    end el
        spirdata <= spirdata;
    end else begin
        spirdata <= 0;
    end
end

// rdr_wen
always @(posedge pclk) begin
    if (!prstn) begin
        rdr_wen <= 0;
    end else if(spi_nextstate == SPI_TRANS_WRITE && !rdr_full) begin
        rdr_wen <= 1;
    end else begin
        rdr_wen <= 0;
    end
end

// rdr_wdata
assign rdr_wdata = spirdata;

// SPI_SR
always @(posedge pclk) begin
```

```
.cpp Vahb2apb_swc_tb___024root__DepSet_h50f898ce__0__Slow.cpp Vahb2apb_swc_tb___024root__DepSet_h19820512__0__Slow.cpp Vahb2apb_swc_tb__Syms.cpp Vahb2apb_swc_tb__Trace
__0__Slow.cpp > Vahb2apb_swc_tb__ALL.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include -I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM
_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-overloaded-virtual -Wno-shadow -Wno-sign-compare -Wno-uninitialized -Wno-unused-but-set-parame
ter -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable     -DVL_TIME_CONTEXT   -fcoroutines -c -o Vahb2apb_swc_tb__ALL.o Vahb2apb_swc_tb__ALL.cpp
echo "" > Vahb2apb_swc_tb__ALL.verilator_deplist.tmp
Archive ar -rcs Vahb2apb_swc_tb__ALL.a Vahb2apb_swc_tb__ALL.o
g++     verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o Vahb2apb_swc_tb__ALL.a    -pthread -lpthread -latomic   -o Vahb2apb_swc_tb
rm Vahb2apb_swc_tb__ALL.verilator_deplist.tmp
make: Leaving directory '/home/nand/mcu_backup/rtl/chip/lib/peripherals_swc/obj_dir'
The data read is: 0000abab
The data read is: 0000acac
The data read is: 0000adad
The data read is: 00000000
The data read is: 000000ab
The data read is: 000000ac
The data read is: 000000ad
- /home/nand/mcu_backup/rtl/chip/lib/peripherals_swc/ahb2apb_swc_tb.v:246: Verilog $finish
nand@nand-virtual-machine:~/mcu_backup/rtl/chip/lib/peripherals_swc$
```

Fig. 18. Loop back result for UART and SPI

5) Tests and results for SRAM: To test the ahbs_dsram_swc SRAM module, the main objective was to thoroughly validate the functionality of the SRAM design under different stimulus scenarios, encompassing both load and store operations within the AHB framework. The test includes a sequence of operations, including write and subsequent read cycles, while monitoring various signals such as hclk, hrstn, haddr, hmastlock, hprot, hsize, htrans, hwdata, hwrite, hready, hrdata, and hresp. Throughout the simulation, our focus lay on ensuring accurate state transitions, proper data handling, and signal responsiveness within the SRAM module. The store/write operation commenced by asserting the write signal (hwrite) and providing specific data such as 32'h12345678 at a designated address (32'h10). Subsequently, the load/read operation followed, confirming the correct retrieval of previously written data. The result meticulously verified these operations by simulating clock cycles, observing changes in signals, and confirming the expected behavior of the SRAM module under these controlled conditions. The primary outcome revealed successful read-and-write functionalities, showcasing the SRAM's ability to appropriately handle data storage and retrieval as per the AHB specifications. Moreover, the verification of state transitions and signal responses under different stimulus conditions validated the robustness and reliability of our SRAM design. Overall, the test results shown in Fig. 19 confirmed the expected behavior and validated the efficacy of our SRAM module in managing data transactions.
To test the ahbs_isram_swc module, the focal objective was to scrutinize the instruction fetching capability within the AHB

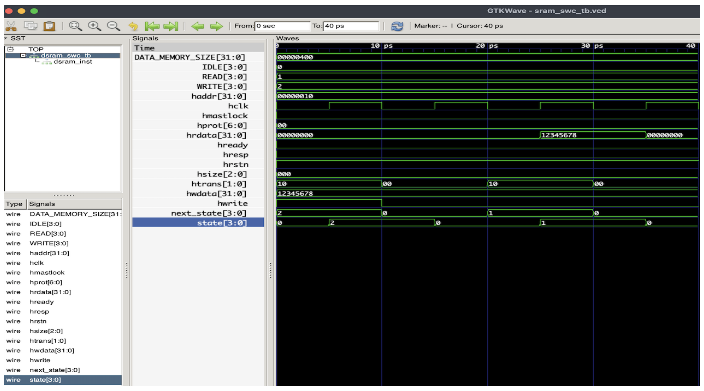


Fig. 19. Test output for data sram performing load/store operations

context. The test is encapsulated by the instantiation of the SRAM module and governed by a clock generation scheme toggling the hclk signal every 5-time units, setting a controlled simulation scenario. The simulation commenced by initializing inputs, including hclk, hrstn, haddr, hmastlock, hprot, hsize, htrans, hwdata, and hwrite, configuring the SRAM module under an idle state. Upon releasing the reset signal (hrstn), our simulation strategically triggered an instruction fetch operation, such as at address 32'h10 by setting the transaction type (htrans) to FETCH. Our primary focus lies in assessing the SRAM's responsiveness and adherence to the AHB specifications for instruction retrieval. The test concluded by resetting the transaction type back to IDLE (htrans = 2'b00), signifying the completion of the fetch operation. The test result shown in Fig. 20 affirmed the SRAM module's capability to execute instruction fetch operations as intended within the AHB, showcasing its aptitude for accurate and controlled instruction retrieval under the specified simulation conditions.

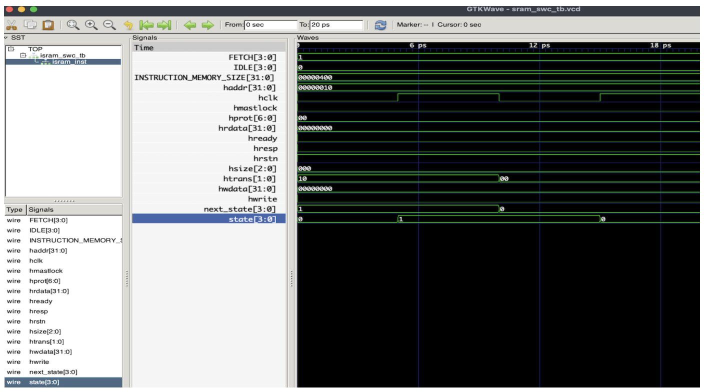


Fig. 20. Test output for instruction sram

IV. Discussion

In developing our switch MCU, we drew inspiration from SiFive's FE310 [5] used in our previous Athena Project but made specific modifications to suit our requirements. Similarly, our datasheet manual, SPI, AMBA, and UART module designs refer to the STM32F107 [6], yet simplified to align with our project goals. For instance, we simplified the USART section, originally supporting smart card mode for the STM32F107 which has the SCK signals, to a basic UART configuration, reflecting our focused design approach. Moreover, we streamlined the AMBA and SPI protocols in accordance with our plan to support only a single ADC, leading to the omission of the CS signal. This strategic simplification reflects our project's unique needs and demonstrates our ability to adapt existing product functionalities to conceive a tailored switch MCU product. Our approach showcases the balance between leveraging established designs and innovating to meet specific project requirements.

V. Conclusion

Due to we have not done the synthesis in our project, our immediate goal is to conduct synthesis and Static Timing Analysis (STA) on each module to ensure they meet timing requirements. The Core and AHB operate at 72MHz, and the APB at 36MHz. If issues arise, further optimization of the design is necessary.

Regarding the Core's design, despite successful tests of various instruction types (J, I, S, R, U, B) using our test platform to run a while loop in C code, and testing Data forwarding, the potential for conflicts with multiple random inputs remains an area for further investigation.

We generated random RV32I instructions using the Unicorn Engine. However, since our back-up SRAM is designated for AMBA communication testing, these instructions need transferring to the official SRAM for comprehensive testing. Once the official SRAM is integrating it into our design, and testing for hazards with the generated RV32I instructions.

For the AMBA part, adding pipeline support to the AHB and incorporating additional signals like HPROT, if needed, as per the AHB protocol, is recommended. For the APB, confirming compliance with STA for peripherals is sufficient.

The SPI part, after thorough testing of the SPI_SR and SPI_CR registers, seems ready for use without timing conflicts. Adding a CS signal for multi-device selection is advisable.

UART basic data communication is achievable, and our testing platform is set. Having just completed testing, future work includes the test of UART_SR.

In summary, our efforts demonstrate that a team of five, in a half-semester, can substantially complete the IP design of a Risc-V RV32I processor, with preliminary verification of modules and simple behavioral tests. This proves that a small team, with limited resources, can feasibly develop a Risc-V processor, at least its frontend, reinforcing the potential of low-cost, small-scale processor development.

VI. Acknowledgements

We would like to express our profound gratitude to Prof. Young for his invaluable support of our project. Without his assistance, we would not have had the opportunity to embark

on this endeavor. This project was initiated based on Yufei's personal idea, and we are immensely thankful to Prof. Young for providing us with the opportunity to continue our research in the form of a Capstone project.
We also wish to extend our heartfelt thanks to Jikun's two cats, Roly and Athena. Their emotional support was a great comfort to us throughout the duration of this project.

REFERENCES

[1] ARM. "AMBA AHB Protocol Specification" ARM Developer Documentation. [Online]. Available: https://developer.arm.com/documentation/ihi0033/latest/. [Accessed: Dec. 1, 2023].

[2] ARM. "AMBA AHB Protocol Specification" ARM Developer Documentation. [Online]. Available: https://developer.arm.com/documentation/ihi0024/latest/. [Accessed: Dec. 1, 2023].

[3] Author(s). "Introduction-to-spi-interface"Introduction-to-spi-interface.[Online]. Available: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf. [Accessed: Dec. 1, 2023].

[4] Texas Instruments, "Title of the Document," [Online]. Available: https://www.ti.com/lit/ug/sprugp1/sprugp1.pdf. [Accessed: Dec. 1, 2023].

[5] StarFive Technology, "FE310-G002 Datasheet V1P2," [Online]. Available: https://starfivetech.com/uploads/fe310-g002-datasheet-v1p2.pdf. [Accessed: Dec. 1, 2023].

[6] STMicroelectronics, "STM32F107VC - Arm Cortex-M3 Microcontrollers," [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f107vc.html. [Accessed: Dec. 1, 2023].