HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Integration of 32-bit RISC-V ALU and ARM via Simple Bus with Scan Testability

Masters in Electrical Engineering and Embedded Systems

Authors

Tsedalu Fentaw Mekonen
10841375

Nishanth Lazar Mohan
15541744

Omkar Nimith Muthabyraiah
16841822

Jayant Patil
18041745

Tejaswi Satish
22041551

Under the guidance of

Prof. Dr. Andreas Siggelkow

History Table		
Current Version: 2.0, 2024-01-16 Previous Version: 1.0, 2023-10-30		
Paragraph (in previous version)	Paragraph (in current version)	Changes
-	All Chapters	Chapters created for the first time.

Table 1: History Table

Contents

List of Tables

List of Figures

Chapter 1

Requirements

Requirements	**ID**	**Priority**	**Verification**	**Description**
General				
Peripheral	G1	High	System C/ Simulation	The peripheral shall consist of 32-bit registers.
Read operation	G1	High	System C/ Simulation	ALU must be readable via registers.
Write operation	G1	High	System C/ Simulation	ALU must be writable via registers.
LEDs	G2	High	System C/ Simulation	The LEDs should respond to logic high and low inputs.
Switches	G3	High	System C/ Simulation	The switches should have high and low states, which can be updated by user inputs.
TAP Controller	G4	High	System C/ Simulation	The TAP controller allows the movement of data through operations ”Scan in” and ”Scan out” via TDI and TDO pins respectively. It also allows the performance of a scan test.
Communication				
Protocol: Simple bus	C1	High	System C/ Simulation	The connection between the ARM core and the registers shall be established using the Simple bus.
Software				

System C	S2	High	N/A	The peripheral must be accurately modeled using System C.
Test				
Scan Test	T1	High	System C/ Simulation	Ensure a scan test implementation on registers with a scan chain model.

Chapter 2

Product Overview

2.1 Top Level View

2.1.1 Introduction

The product overview that follows includes a detailed description of a project targeted at improving the capabilities of the ARM core on the Zed Board FPGA. The project entails the addition of a peripheral that includes a 32-bit RISC-V ALU that can be used for both reading and writing tasks. The registers are intended to control the LEDs by capturing switch inputs. The Simple bus facilitates communication between the ARM and the ALU. A TAP-controller (intended) and a scan chain are added into the system to assure the design's testability.

As shown in the Figure 2.1, we drive the LEDs by reading the states of the switches embedded on the ZedBoard. To enable the read and write operations, we develop a write function and a read function within the software component of the project. In addition, we implement 32 bit RISC-V ALU with System C. The code for the peripheral's 32-bit RISC-V ALU is meticulously crafted to facilitate both read and write operations, enabling seamless communication with the ARM core. One of the registers shall drive the LEDs, and the other shall receive inputs from switches on the ZedBoard. The communication between the ARM core and the ALU is established through the utilization of the Simple bus. Finally, to facilitate the testing of the system, it is intended to implement TAP controller, which enables the movement of data through the ”Scan in” and ”Scan out” operations via the TDI and TDO pins, respectively, and it also allows the execution of a scan test.

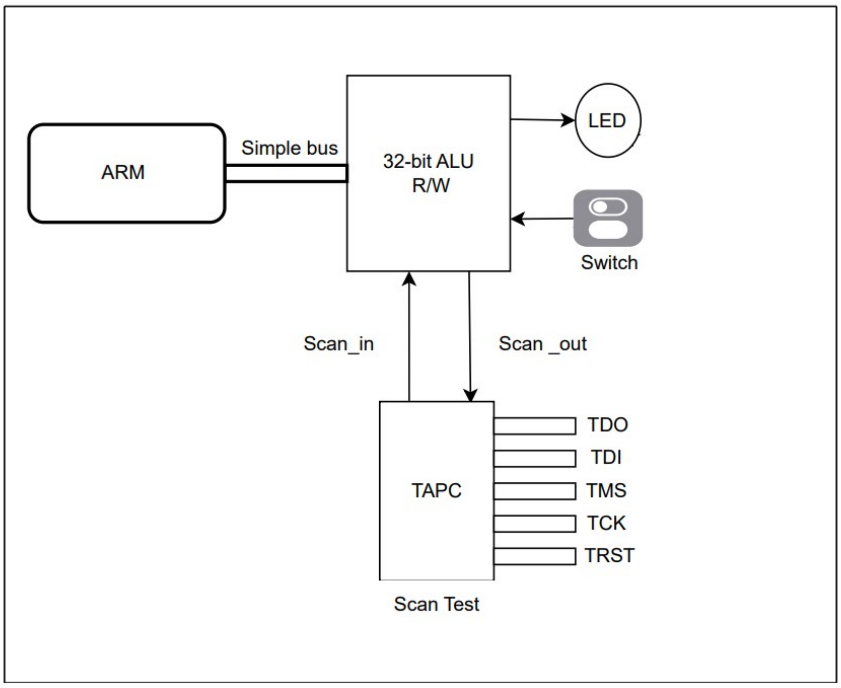


Figure 2.1: Top Level view

2.1.2 Key Features

ARM

The ARM system employs registers to store and manage data. Registers act as fast, accessible storage units directly connected to the ALU. The code indicates that the ARM writes data to specific registers, initiating the processing cycle. This implies a register-based architecture, a common feature in many processor designs.

RISC V 32-bit ALU

- Reduced Instruction Set: The RISC-V ALU would support a reduced set of simple and efficient instructions, which are designed to execute quickly and enable faster processing.
- Single Clock Cycle Execution: Most instructions would be designed to execute in a single clock cycle, ensuring fast and efficient processing.
- Data Processing: The ALU operates on 32-bit data, which means it can perform arithmetic and logical operations on 32-bit binary numbers.

Simple Bus

- Supports read and write data transfers of 32-bit width.

- Supports Addressing and Identification.
- Error Handling and Correction.
- Synchronous bus systems for precise timing of data transfer.

TAP Controller

- IEEE Standard 1149.1
- 16-state Finite State Machine
- TAPC is part of JTAG

Scan Chain

- IEEE Standard 1149.1
- Ability to scan and test every flipflop in an Integrated Circuit (IC)
- Interconnect testing, In-System programming and debugging

2.1.3 Functional Block Diagram

The below depicted model will be actively implemented for the purpose of executing RISC-V Instructions on a Zybo board via ALU with 32-bit registers. These Registers are Readable and Writeable by the Processing System of the Zybo board via Simple bus.

Top Module: The Top module system's components, which include the ARM core, registers, ALU, and scan chain, communicate via a simple bus architecture. This design promotes efficient data sharing. The system stores data in registers, which are directly coupled to the ALU in a register-based design. The ALU conducts arithmetic and logical operations on data from registers, generating results that are returned to registers. Write operations include beginning data writes to certain registers, delivering opcode to decide ALU operations, and carrying out the desired operation. Read operations involve getting results from specific registers and reading flags indicating various circumstances such as zero and carry flags, which provide information about the ALU's status following the operation.

Simple Bus: In a system, a fundamental communication framework is established through a simple bus architecture featuring master and slave components. This bus, functioning as a data highway, consists of parallel wires that convey signals such as data, addresses, and control instructions. At the forefront is the master, typically the central processing unit (CPU), which directs communication by generating control signals and addresses. Masters, typically CPUs, initiate read or write operations with specific slave devices.

Scan Chain: The design includes debugger and 7 registers of 32-bit each. The data transmission considered is Parallel In Parallel Out (PIPO). During the test mode, 1-bit Test Data Input (TDI) is fed to each register to the LSB position. The register being 32-bit, TDI is then left shifted to MSB and Test Data Output (TDO) is fetched from each of the register's MSB position. As TAPC is not part of the implementation, standalone debugger is used to pass the TDI.

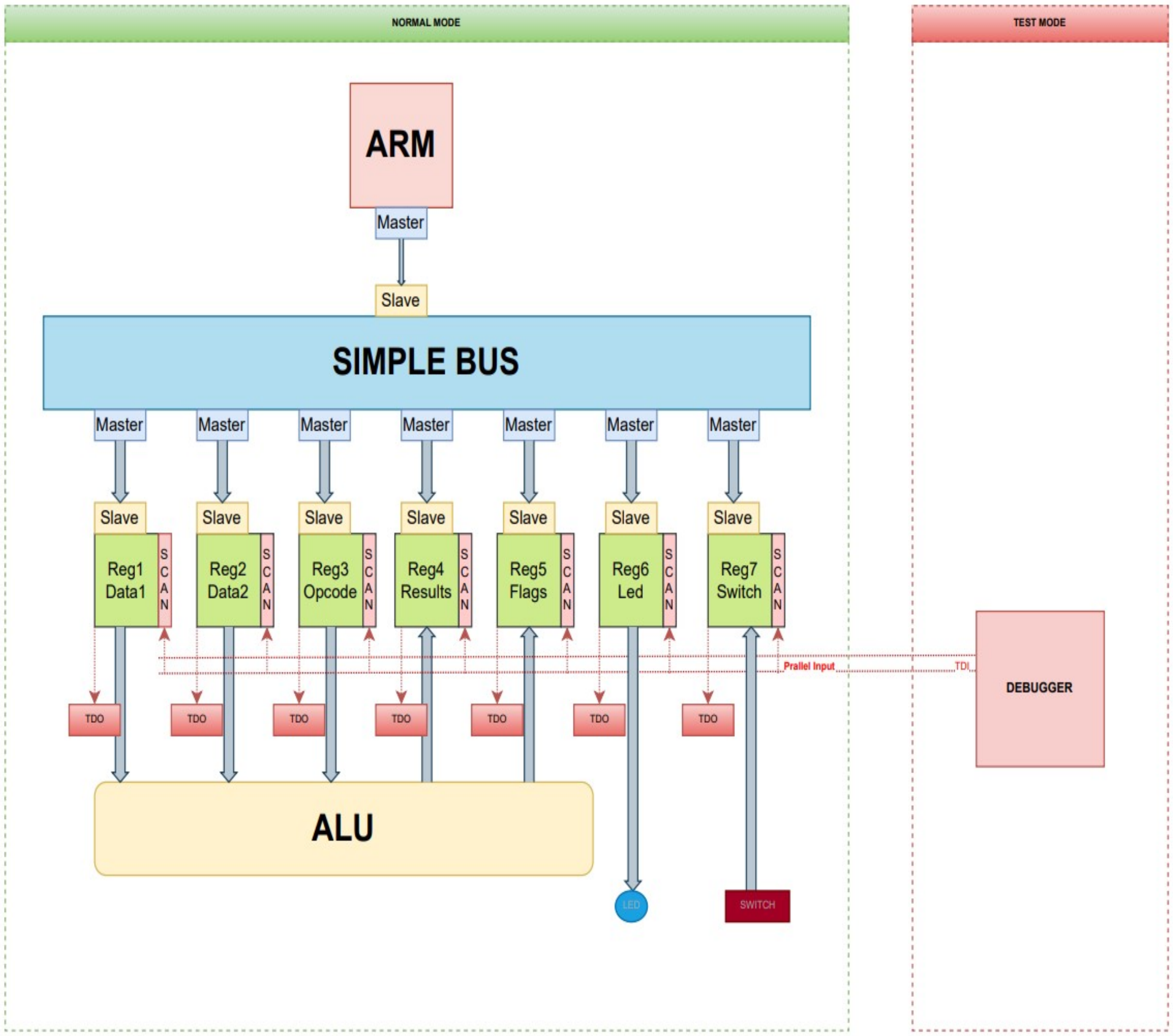


Figure 2.2: Functional Block diagram

Chapter 3

Architecture Concepts

3.1 Bus Concept

In a computing system, the orchestration of data transfers and communications on the bus is facilitated through the bus peripheral interface (BPI). This interface serves as a dedicated module responsible for managing interactions between the central processing unit (CPU), memory, and various peripheral devices connected to the system's bus. The crucial aspect highlighted in the statement is that the handling of bus transactions is not hardwired but is instead programmed. This programming aspect implies that developers write code to instruct the BPI on how to execute specific actions related to the bus, such as read and write operations, data transfers, and adherence to communication protocols. By utilizing programming instructions within the BPI, the system gains flexibility, allowing for customization of bus behavior to meet diverse requirements and adapt to various scenarios.

3.1.1 Master BPI

- `SystemMasterBPI<T>` is a templated class that represents a System Master Bus and Peripheral Interface.
- The constructor takes parameters such as the module name (nm), an ID, and a reference to an event (mBPIDatRdEv).
- Initialization of the constructor includes setting up the module name, ID, event reference, and creating a socket (Master1Socket).
- Two SystemC threads (initThread and cmdThread) are registered in the constructor.

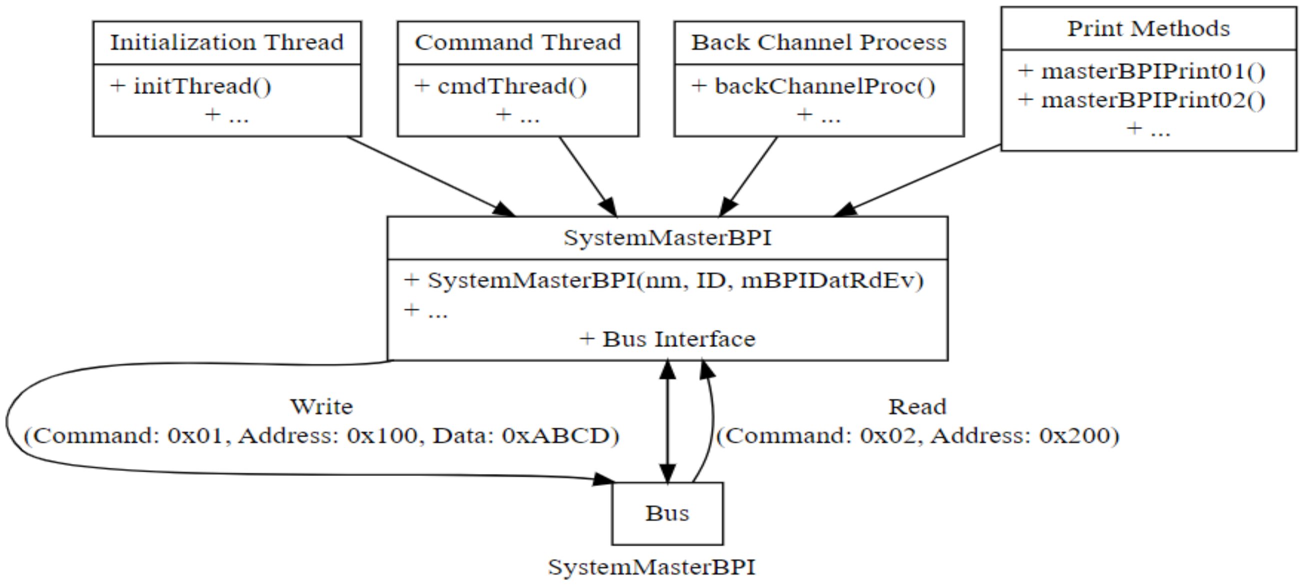


Figure 3.1: Master BPI Block Diagram

- Input exports (`data_i`, `write_i`, `address_i`) are bound to corresponding internal signals (`dataWr_s`, `write_s`, `address_s`).

Initialization Thread:

- `initThread` is a SystemC thread responsible for controlling the bus transactions. It waits for changes on the bus (`cmd_e` event) and reacts accordingly.
- If a write command is detected, it retrieves data from `dataWr_s` and prepares a payload for the socket. The payload is sent to the socket using blocking transport (`b_transport`).
- If a read command is detected, it processes the read data, writes it to `data_o`, and notifies the associated event (`mBPIDatRdEv_e`).
- Back channel responses are checked for errors using `backChannelProc`.
- Additional printouts are included for debugging purposes.
- There's a wait for a delay at the end.

Back Channel Process:

- `backChannelProc` is responsible for processing back-channel responses from the socket.
- If there is a response error, it reports the error using `SC_REPORT_ERROR`.
- If there is no error, it processes the successful response.
- If debug prints are enabled, it prints relevant information.

Command Thread:

- `cmdThread` is another SystemC thread responsible for monitoring changes in the write signal (`write_s`).
- It waits for the signal to change and then notifies `initThread` that something has happened on the bus.

masterBPI Print Methods

- `masterBPIPrint01` This method prints a message indicating that the Master-BPI module is waiting for some action on the bus.
- `masterBPIPrint02` Prints a message indicating the end of waiting for action on the bus.
- `masterBPIPrint03` Prints information about a bus operation, including the byte address, data, and command (Read or Write).
- `masterBPIPrint04` Prints a message indicating the end of a delay in the bus operation.
- `masterBPIPrint05` Prints information about a successful back-channel response, including the data received.
- `masterBPIPrint06` Prints information about a change in the command signal on the bus.

3.1.2 Slave BPI

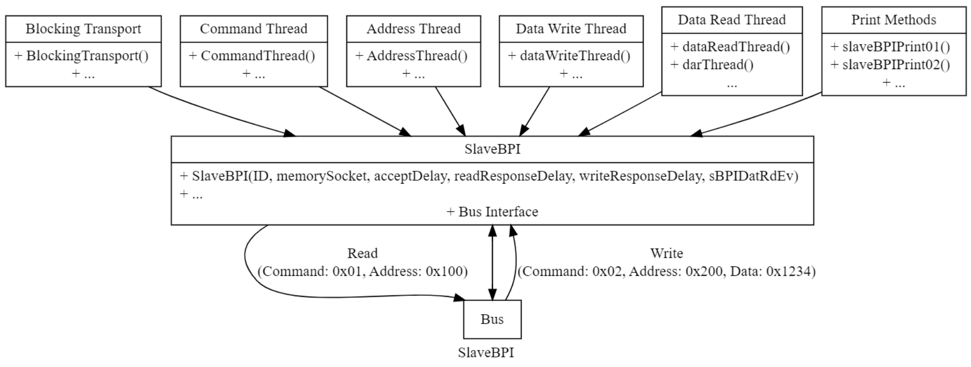


Figure 3.2: Slave BPI Block Diagram

Components:

- `ID`: Target ID for the slave device.
- `memorySocket`: Socket name for communication.
- `acceptDelay`: Accept delay for bus transactions.
- `readResponseDelay`: Delay for read response.
- `writeResponseDelay`: Delay for write response.
- `sBPIDatRdEv`: Reference to an event used for data read .synchronization

Blocking Transport Method:

- Implements the TLM blocking transport method for handling read and write transactions.
- Extracts information from the payload, such as command, address, data pointer, length, etc.
- Checks for errors (address error, byte enable error, burst error).
- Performs read or write operations based on the command.
- Sets the response status and delay accordingly.

Command Thread:

- Delivers ”Read” or ”Write” signals to the `sc_export`.

- Monitors changes in the `write_s` signal and writes the value to the `write_o` `sc_export`.

Address Thread:

- Delivers/writes ”Address” signals to the `sc_export`.
- Monitors changes in the `adr_s` signal and writes the value to the `address_o` `sc_export`.

dawThread:

- Delivers/writes ”Data” signals to the `sc_export`.
- Monitors changes in the `daw_s` signal and writes the value to the `data_o` `sc_export`.

darThread:

- Handles data read events and triggers the bus transport.
- Waits for data coming from the functional block (`sBPIDatRdEv_e`).
- Reads data from the `dataRd_s` signal.
- Notifies the bus transport that data is available (`dataRd_e.notify(SC_ZERO_TIME)`).

SlaveBPI Print Methods:

- `slaveBPIPrint01` - Prints information about a `b_transport` operation, including the word address, maximum memory size, and target ID.
- `slaveBPIPrint02` - Prints information about a read operation, including the word address and data.
- `slaveBPIPrint03` - Prints information about a write operation, including the word address and data.
- `slaveBPIPrint04` - Prints information about an ignore command, including the word address, data, and command.
- `slaveBPIPrint05` - Prints information about a data read event from the functional block.

3.2 Registers

A 32-bit RISC-V register serves as a dedicated storage location within a computer's central processing unit (CPU), designed to accommodate a 32-bit binary value. The term "32-bit" denotes the register's capacity, indicating its ability to store data comprising 32 binary digits (bits). Integral to the RISC-V architecture, these registers form part of the general-purpose register set employed by the CPU for various computational tasks.Key attributes of a 32-bit RISC-V register encompass its size, capable of holding 32 bits, offering a vast range of 2^{32} potential binary combinations. These registers are designated as general-purpose, allowing them to store diverse data types, including integers, addresses, and other fundamental data entities. Embedded within the RISC-V architecture, denoting Reduced Instruction Set Computing - Version 5, these registers play a pivotal role in storing and manipulating data during the execution of instructions.

Beyond their general-purpose nature, 32-bit RISC-V registers contribute significantly to data processing tasks, participating in arithmetic and logical operations. The CPU utilizes these registers to perform computations on the stored data, underscoring their indispensability in executing a wide array of tasks and programs. The addressing of RISC-V registers involves distinct register names, such as `x0, x1, ..., x31`, providing a systematic means to identify and manage different registers within the processor.Programmers and compilers leverage the capabilities of these registers to optimize code execution, capitalizing on the simplicity and efficiency inherent in the RISC-V architecture. The preference for 32-bit registers in modern computer architectures reflects a strategic balance between computational capabilities and memory efficiency, affirming their central role in contemporary computing systems.

3.3 ARM System Architecture Overview

The ARM system relies on a simple bus architecture for communication between different components. The bus facilitates the transfer of data between the ARM core, registers, ALU, and memory. This straightforward communication scheme simplifies the overall design and ensures efficient data exchange.

3.3.1 Register Interaction:

The ARM system employs registers to store and manage data. Registers act as fast, accessible storage units directly connected to the ALU. The code indicates that the ARM writes data to specific registers, initiating the processing cycle. This implies a register-based architecture, a common feature in many processor designs.

3.3.2 ALU Operation:

The Arithmetic Logic Unit (ALU) is a critical component of the ARM system responsible for performing arithmetic and logical operations on data. The ALU takes input data from registers, executes the specified operation (such as addition, subtraction, etc.), and produces a result. The result is then stored back into registers for further processing or retrieval.

3.3.3 Write Operations

1. **Operand Data Write (Write Section)**: The ARM system initiates write operations by providing operand data to specific registers. This is done through the simple bus, where data is written to designated registers with unique identifiers (Slave IDs).

2. **Opcode Write**: The ARM system also writes opcode to a specific register. This opcode determines the type of operation the ALU should perform. The control signal is transmitted over the bus and stored in the corresponding register.

3. **ALU Processing**: Once the necessary data and opcode are in place, the ALU executes the specified operation. This involves reading data from the registers, performing the operation, and generating a result.

3.3.4 Read Operations

1. **Result Read (Read Section)**: After the ALU completes the operation, the result is stored in a designated register. The ARM system then reads this result from the register, providing visibility into the outcome of the computation.
2. **Flag Read**: Additionally, the system reads flags that indicate various conditions such as zero flags, carry flags, etc. These flags provide information about the state of the ALU after the operation.

3.4 Arithmetic Logic Unit (ALU)

The brains of the computer are the arithmetic logic unit (ALU), which can execute logical operations like AND and OR as well as set less than instructions. It can also perform arithmetic operations like addition and subtraction. In order to demonstrate how combinational logic functions, this section builds an ALU using four hardware building blocks: AND and OR gates, inverters, and multiplexers. We'll see how more ingenious designs can speed up addition and subtraction. We require a 32-bit wide ALU since the RISC-V registers are 32 bits wide. The Arithmetic Logic Unit (ALU) plays a crucial role in the execution of operations within a computer's central processing unit (CPU). Once the essential data and control signals are properly configured, the ALU swings into action, carrying out the specified operation. This process encompasses the retrieval of necessary data from registers, the execution of the operation itself, and the subsequent generation of a resulting value.

Opcode in decimal	Opcode in Binary	ALU Operation
0	0000	AND
1	0001	OR
2	0010	Adder
3	0011	Subtract
4	0100	Set less than
5	0101	NOR

Table 3.1: The values of the three ALU control lines, Ainvert, Bnegate, and Operation, and the corresponding ALU operations

As the ALU concludes its computation, the resulting value is securely stored in a designated register. This register acts as a temporary holding space for the outcome of the operation. Following this, the ARM system proceeds to read or fetch this result from the register. This step is pivotal for gaining visibility into the computed outcome and is fundamental to the overall functionality of the ALU.

Beyond obtaining the numerical result, the system also seeks information about the state of the ALU through the retrieval of flags. Flags serve as indicators that convey specific conditions met during the operation. These conditions may include whether the result is zero, if a carry occurred, or other relevant status updates. By reading these flags, the system gains valuable insights into the nuances of the computation, allowing for further analysis and decision-making in subsequent stages of processing. This aspect enhances the overall efficiency and adaptability of the ALU in responding to diverse computational scenarios.

3.5 Scan Chain

An illustration depicts the distinctions between a scannable and a standard flip-flop. The scannable flip-flop only interacts with the TDO pin, whereas the regular flip-flop connects to both the TDI and TDO pins. These flip-flops must be part of a scan chain in order to be tested effectively. The scan chain connects all flip-flops in a series loop; the TDO pin of one flip-flop is immediately connected to the TDI pin of the next flip-flop. To load test vectors, scan chain clocks are pulsed after the TDI pins have been set to the required value. This process ensures that the test vector is systematically shifted along the scan chain before being loaded into the appropriate flip-flops. To unload a test vector, pulse the scan chain clock while setting the TDI pin to a high impedance condition. This smart action causes the test vector to go through the flip-flops and emerge on the TDO pin, completing the circuit's testing cycle.

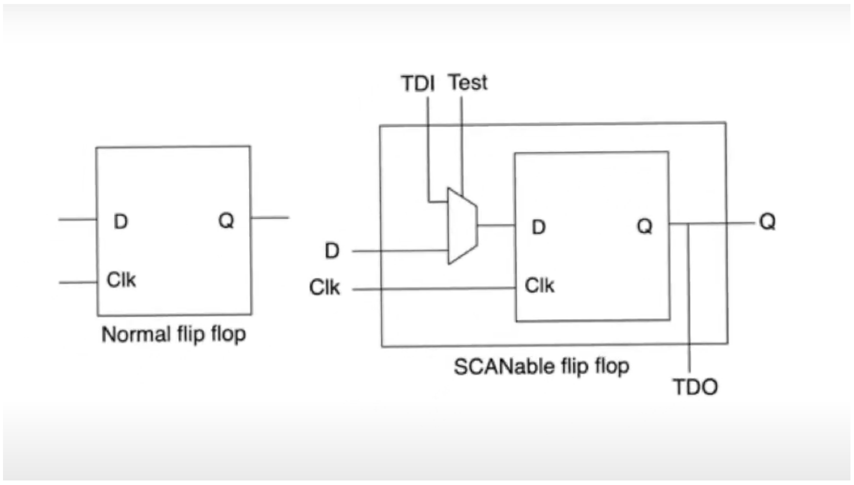


Figure 3.3: Scan Flipflop

Chapter 4

Description of Design Elements

4.1 ARM

1. **Constructor:** - The constructor initializes the ARMFunc module and sets up its SC_THREADs. - It also binds the data_i port to the dataIn_s export.

2. **getDatThread:** - This thread waits for an event (mFuncDatRdEv_e) signaling that read data is available. - Upon receiving the signal, it reads data from the dataIn_s port and prints information if specified.

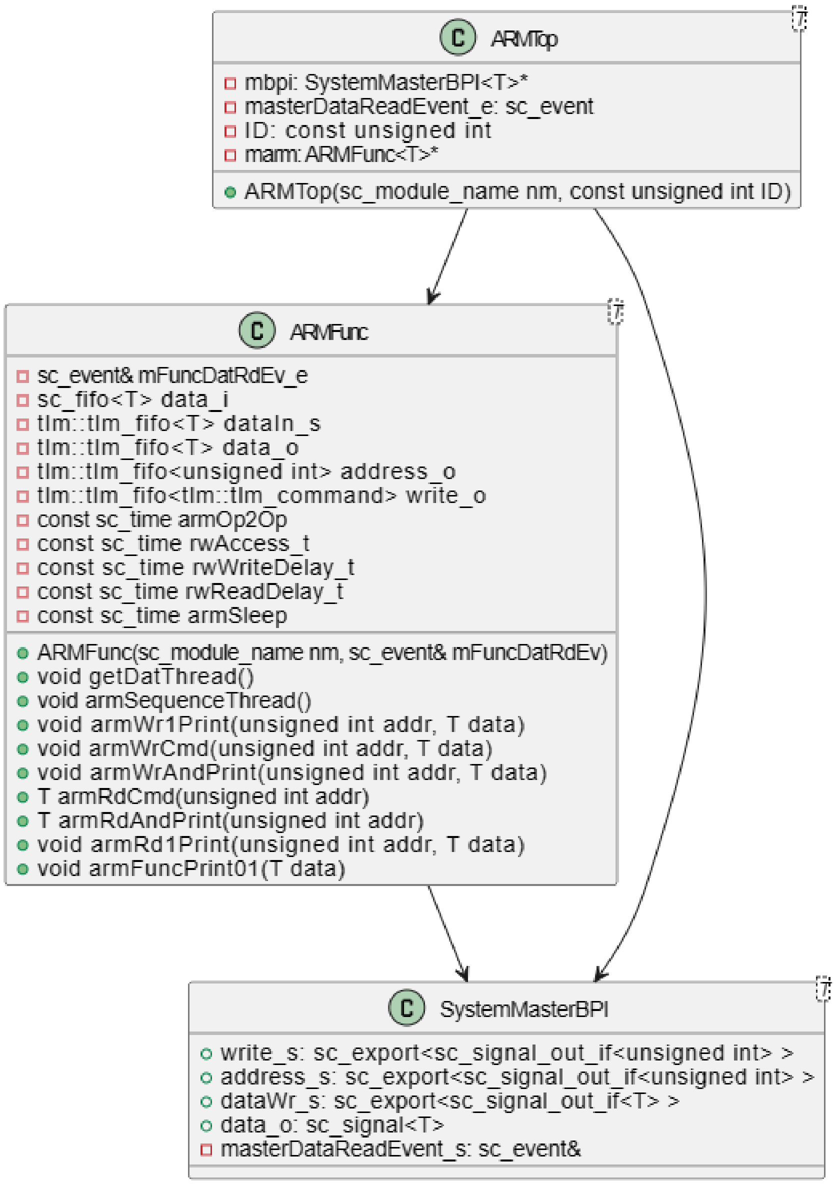
3. **armSequenceThread:** - This is the main algorithm of the ARMFunc module, executed in a SystemC SC_THREAD. - It performs a series of write and read operations to simulate the behavior of an ARM processor. - The operations include writing data to operands, writing an opcode (control signal), and reading results and flags. - The printed information includes details about the operations, addresses, data values, and timing.

4. **Flag Handling:** The code includes flag handling, where different flags (such as zero, carryin, lessthan, carryoverflow, negative) are checked based on ALU operation results. Flags play a crucial role in controlling program flow and providing status information.

5. **armWrCmd and armWrAndPrint:** - armWrCmd writes data to the BPI (Bus-Peripheral Interface) by sending a write command. - armWrAndPrint additionally prints information about the write operation.

6. **armRdCmd and armRdAndPrint:** - armRdCmd reads data from the BPI by sending a read command. - armRdAndPrint additionally prints information about the read operation.

7. **armRd1Print, armWr1Print, armFuncPrint01:** - These functions are responsible for printing various information related to read and write operations. The code contains various print statements that provide detailed information about the execution of the ARM functional block, including read and write operations, addresses, data values, and timing information.



C ARMTop
mbpi: SystemMasterBPI<T>*
masterDataReadEvent_e: sc_event
ID: const unsigned int
marm: ARMFunc<T>*
ARMTop(sc_module_name nm, const unsigned int ID)
C ARMFunc
sc_event& mFuncDatRdEv_e
sc_fifo<T> data_i
tlm::tlm_fifo<T> dataIn_s
tlm::tlm_fifo<T> data_o
tlm::tlm_fifo<unsigned int> address_o
tlm::tlm_fifo<tlm::tlm_command> write_o
const sc_time armOp2Op
const sc_time rwAccess_t
const sc_time rwWriteDelay_t
const sc_time rwReadDelay_t
const sc_time armSleep
ARMFunc(sc_module_name nm, sc_event& mFuncDatRdEv)
void getDatThread()
void armSequenceThread()
void armWr1Print(unsigned int addr, T data)
void armWrCmd(unsigned int addr, T data)
void armWrAndPrint(unsigned int addr, T data)
T armRdCmd(unsigned int addr)
T armRdAndPrint(unsigned int addr)
void armRd1Print(unsigned int addr, T data)
void armFuncPrint01(T data)
C SystemMasterBPI
write_s: sc_export<sc_signal_out_if<unsigned int> >
address_s: sc_export<sc_signal_out_if<unsigned int> >
dataWr_s: sc_export<sc_signal_out_if<T> >
data_o: sc_signal<T>
masterDataReadEvent_s: sc_event&
T

Figure 4.1: ARM Class Diagram

4.2 Registers

4.2.1 Operation

In this system architecture, the coordination of signals, crucial for the interaction between the master (simple bus), registers, and the 32-bit RISC-V ALU, is typically implemented through code. Signal binding in the code ensures that the correct signals are generated, transmitted, and interpreted by the respective components.

For instance, in a read operation, the code governing the master generates the appropriate read signal and addresses the specific 32-bit register on the bus. The code in the register then interprets this signal, fetching the requested data from the 32-bit ALU and placing it onto the bus for the master to read. Similarly, during a write operation, the code generates the write signal, transmits both the 32-bit data and the target register's address, and instructs the 32-bit ALU to store the incoming information in the designated register.

This code-driven signal binding ensures a synchronized and organized communication flow within the system. It aligns the actions of the master, registers, and the 32-bit RISC-V ALU, ensuring that data is transferred accurately and operations are executed as intended. The coded logic governing signal interactions forms a fundamental aspect of the system's functionality, contributing to its efficiency in data handling and computation.

4.3 Simple Bus

In a computer system, a simple bus architecture with master and slave components establishes a fundamental communication framework. The bus, akin to a data highway, comprises parallel wires conveying signals like data, addresses, and control instructions. At the helm is the master, typically the central processing unit (CPU), orchestrating communication by generating control signals and addresses. Masters initiate read or write operations with specific slave devices.

Slaves, encompassing peripherals, memory modules, or other subsystems, await commands from the master. Each slave is assigned a unique address. When the master desires data from a slave, it transmits the target address for a read operation, prompting the slave to respond with the requested data. Conversely, for a write operation, the master transmits both the target address and the data to be written, and the addressed slave stores the information in the appropriate location.

This interaction encapsulates the essence of a master-slave relationship on the bus, where the master takes charge of communication initiation, and slaves respond to these requests. Bus arbitration mechanisms ensure coordinated access, crucial in systems with multiple masters. Ultimately, this simple bus architecture facilitates the seamless exchange of data and instructions, forming the backbone of communication within a computer system.

4.3.1 Simple Bus operation with Registers

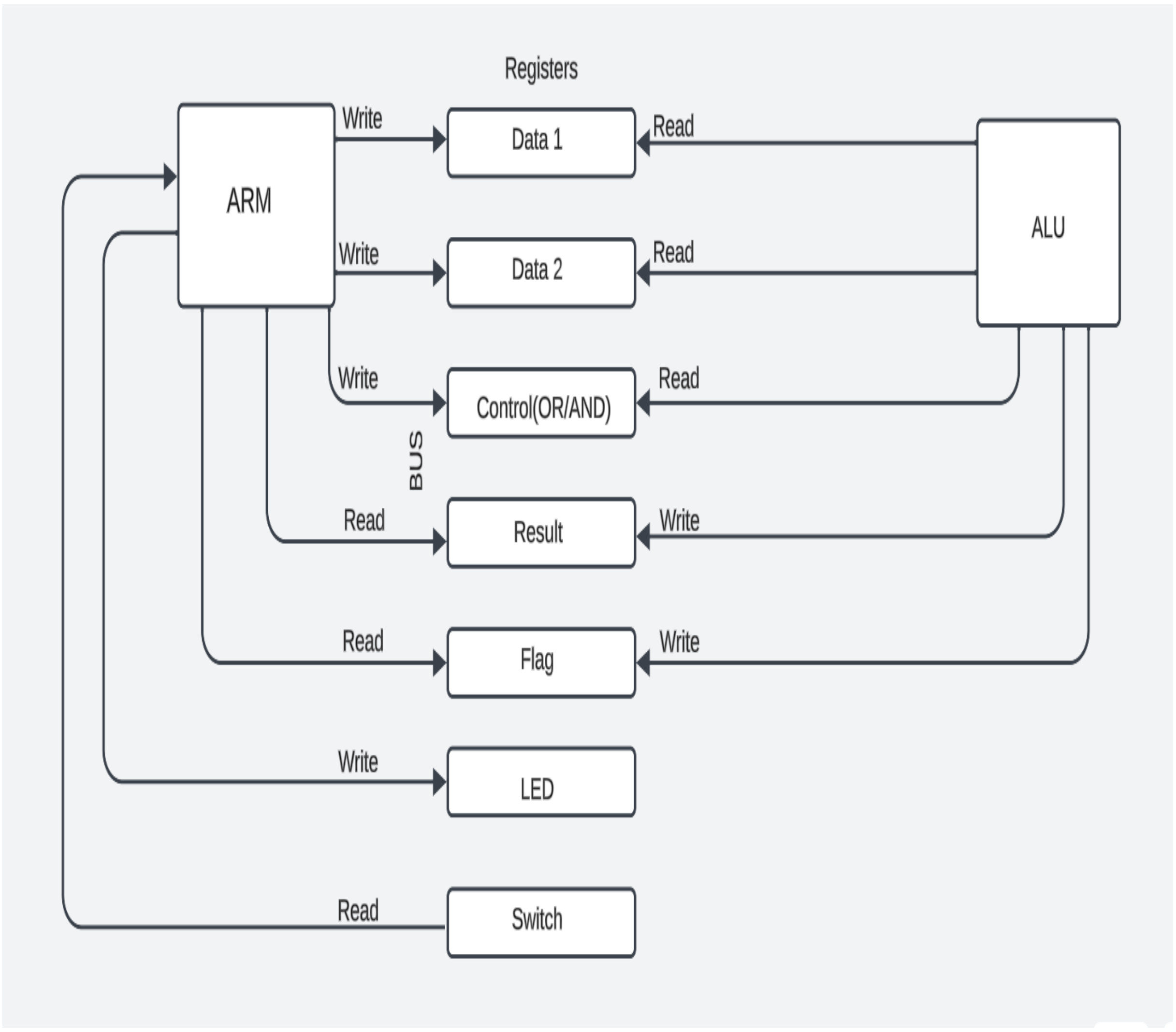
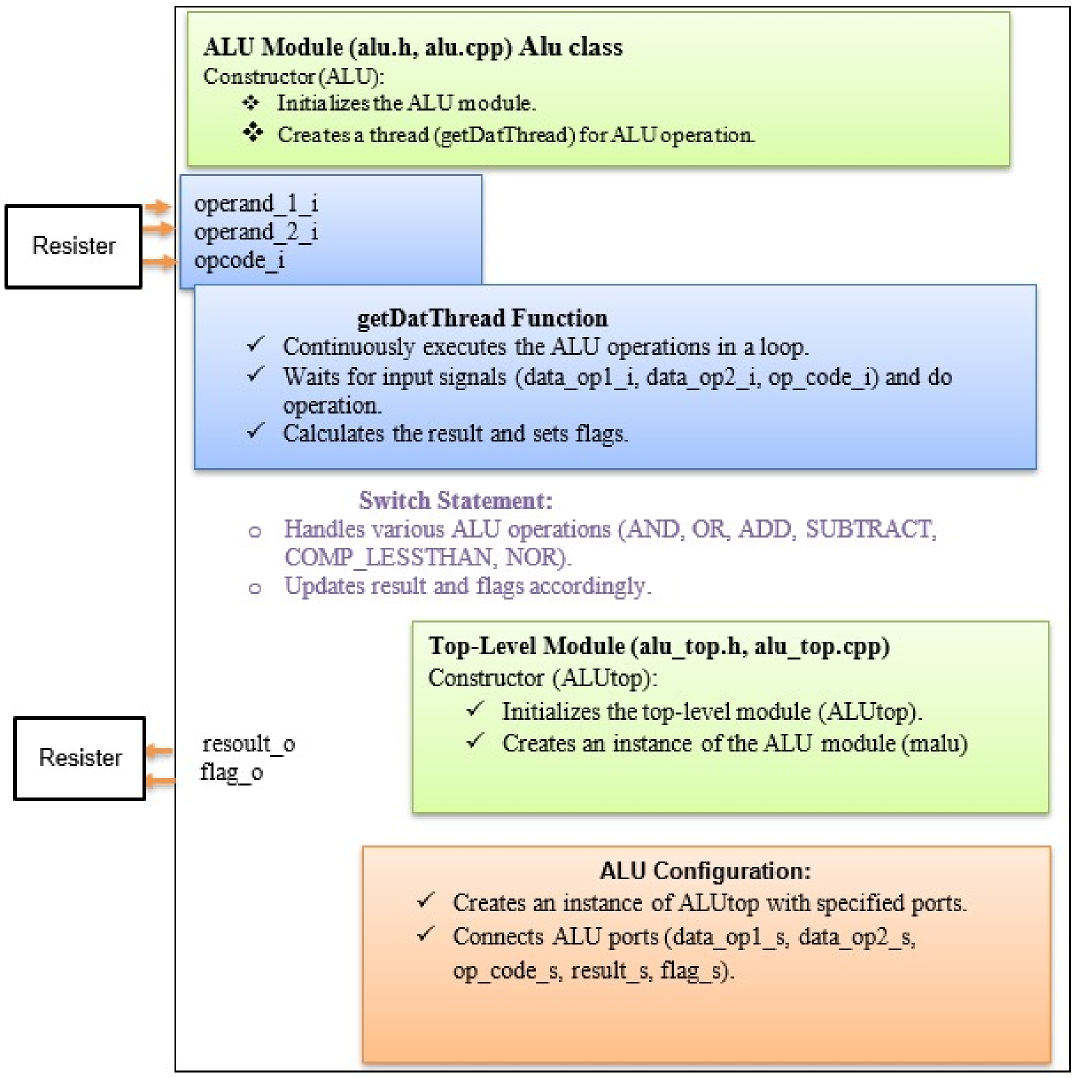


Figure 4.2: Simple Bus Block Diagram

In a system featuring a simple bus as the master connected to 7 registers functioning as slaves, a communication network is established to facilitate data transfer and control within the system. The simple bus, acting as the master, initiates and coordinates data transactions by sending control signals and addresses to communicate with the connected registers. Each of the 7 registers, serving as slaves, is assigned a unique address on the bus, enabling the master to target and interact with individual registers. Building upon the initial configuration of a simple bus connected to 7 registers, where registers 1 and 2 write data, register 3 writes control, register 4 reads results, register 5 reads flag conditions, and registers 6 and 7 are designated for an LED and a switch respectively, the system extends its functionality by further connecting these registers to ARM with Arithmetic Logic Unit (ALU).

The ALU, a digital circuit capable of performing arithmetic and logical operations, becomes an integral part of the system's computational capabilities. During operations involving data written to registers 1 and 2, the simple bus coordinates the transfer of this data to the ALU for computation. Register 3, responsible for control signals, communicates instructions to the ALU, directing it on the type of operation to be performed. After computations are executed within the ALU, the results are stored in register 4, which serves as a data reader. The flag conditions, read from register 5, can influence the ALU's operations or be used by the ARM to make decisions based on the system's state. The interconnected setup ensures a seamless flow of data and control between the registers and the ALU. Additionally, the integration of registers 6 and 7, associated with an LED and a switch respectively, allows the system to interact with external components. For instance, the switch state (register 7) might influence the ALU's behavior, and the results or status could be reflected through the LED (register 6). This comprehensive communication network, with registers playing specific roles and the ALU contributing computational capabilities, enhances the system's versatility and efficiency. The simple bus facilitates these interactions, creating a framework for data exchange, control, and computation within the system.

4.4 ALU



ALU Module (alu.h, alu.cpp) Alu class
Constructor (ALU):
- Initializes the ALU module.
- Creates a thread (getDatThread) for ALU operation.

getDatThread Function
- ✓ Continuously executes the ALU operations in a loop.
- ✓ Waits for input signals (data_op1_i, data_op2_i, op_code_i) and do operation.
- ✓ Calculates the result and sets flags.

Switch Statement:
- Handles various ALU operations (AND, OR, ADD, SUBTRACT, COMP_LESSTHAN, NOR).
- Updates result and flags accordingly.

Top-Level Module (alu_top.h, alu_top.cpp)
Constructor (ALUtop):
- ✓ Initializes the top-level module (ALUtop).
- ✓ Creates an instance of the ALU module (malu)

ALU Configuration:
- ✓ Creates an instance of ALUtop with specified ports.
- ✓ Connects ALU ports (data_op1_s, data_op2_s, op_code_s, result_s, flag_s).

Figure 4.3: ALU Class Diagram.

4.4.1 A 1 Bit ALU AND and OR

The 1-bit logical unit for AND and OR is shown in Figure 4.4. Next, the multiplexer on the right takes the inputs from the ARM and performs the operation of operand_1 AND operand_2 or operand_1 OR operand_2, depending on whether the value of operation is 0 or 1. See how the control and output lines of the multiplexer have been renamed to reflect the function of the ALU more accurately.

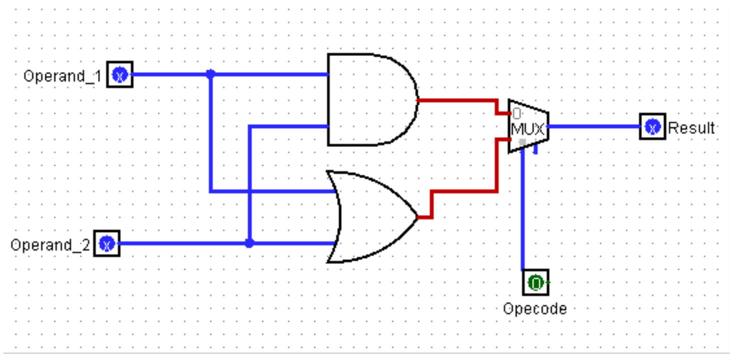


Figure 4.4: The 1-bit logical unit for AND and OR.

4.4.2 Addition and Subtraction

Addition and subtraction are the next function to be included. Two operand inputs and a single-bit output for the sum and difference are required for an adder and subtractor. To pass on the carry, a second output known as CarryOut is required. We require a third input because the CarryOut from the neighbor adder needs to be included as an input. CarryIn is the name of this input.

Adders carry out subtraction in the same way as they would add the negative form of an operand. Remember that we can quickly negate a two's complement number by adding 1 and inverting each bit, which is also referred to as the one's complement. The inputs and outputs of a 1-bit adder and subtractor are displayed in Figure 4.5.

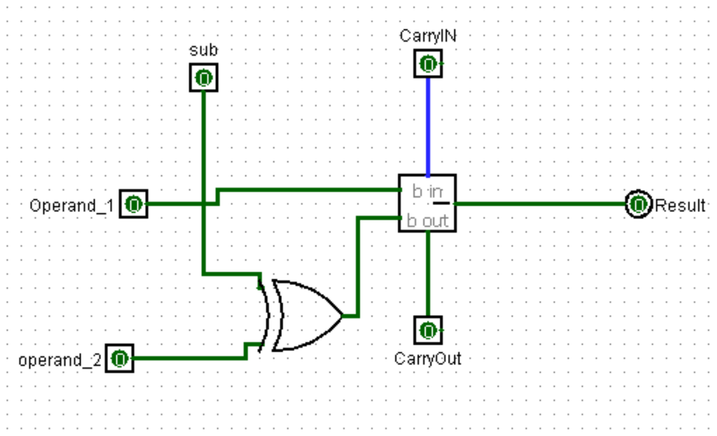


Figure 4.5: The 1-bit Addition and subtraction

4.4.3 1-Bit ALU

Figure4.6 [ref 1, page1187] below illustrates how we compiled all of the individual AND, OR, and addition operations into a single 1-bit ALU operation after completing each one separately. Furthermore, to these three operations—add, AND and OR we need set less than instruction of RISC-V ALU for comparing of the two operands that we have taken from the ARM. so, the set less than instruction (slt) Recall that the operation produces 1 if rs1 less than rs2, and 0 otherwise.

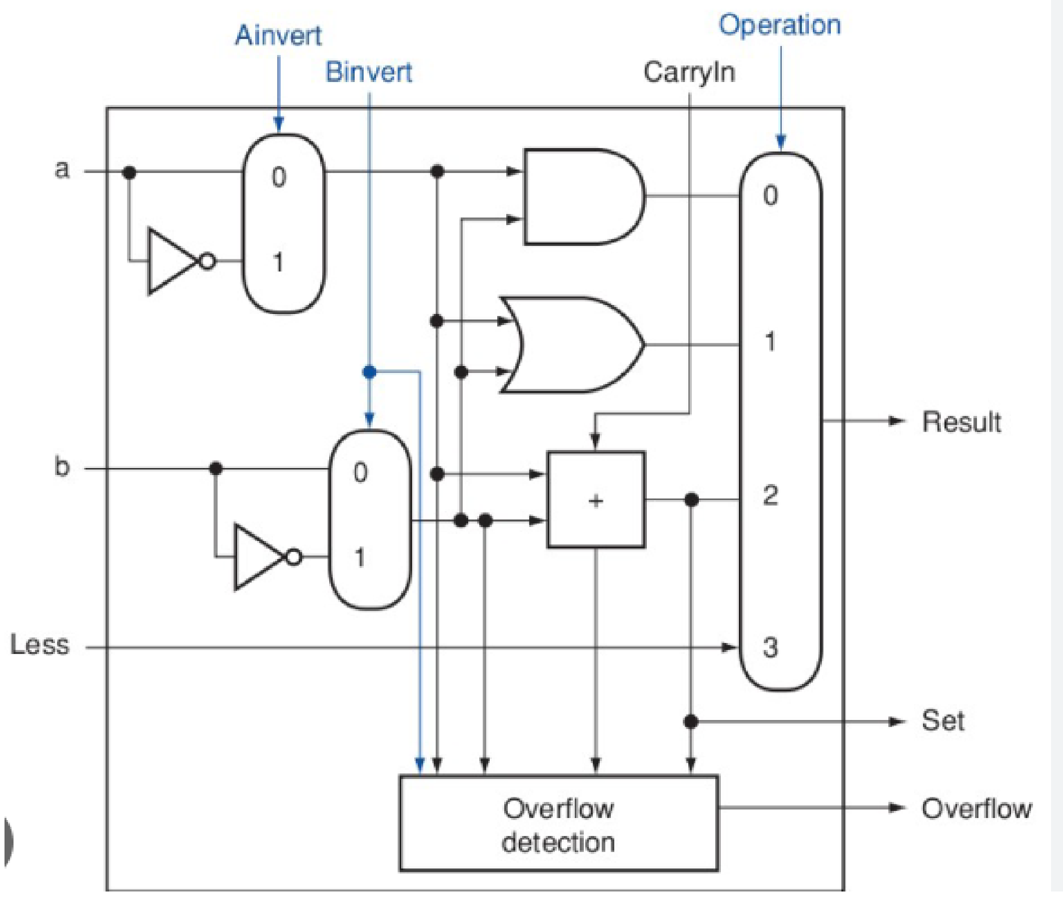


Figure 4.6: A 1-bit ALU

4.4.4 A 32 Bit ALU Operation

Once the 1-bit ALU is complete, we connect the adjacent ”boxes” to generate the complete 32-bit ALU, which can be used for addition, subtraction, AND, OR, and set less than input. After performing the subtraction operation as follows, a–b, CarryIn and Binvert are now both set to 1. Both control lines ought to be zero in the case of additions or logical operations. Therefore, as illustrated in Figure 4.7[ref 1, page1191], we can simplify the control of the ALU by combining the CarryIn and Binvert into a single control line known as Bnegate.

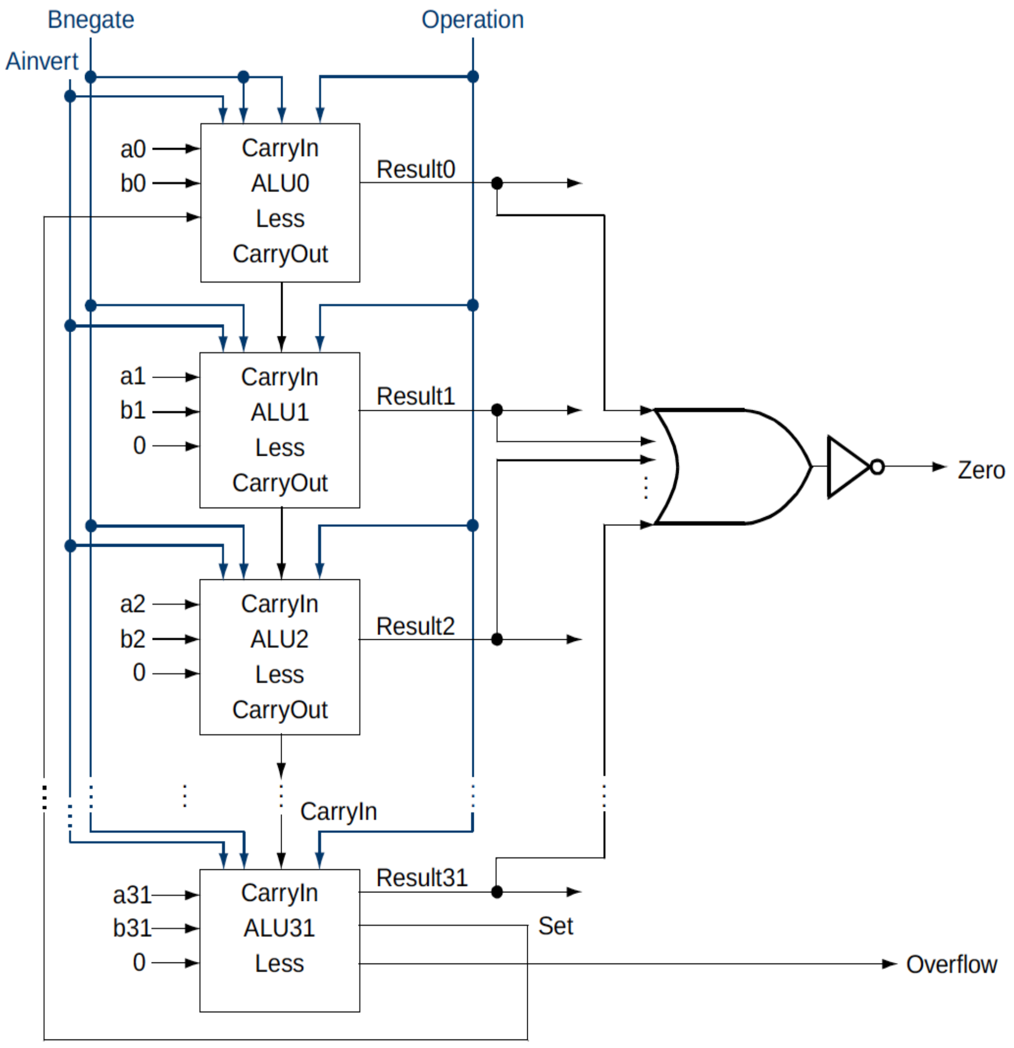


Figure 4.7: The final 32 bit ALU

4.5 Scan Chain

TLM (Transaction Level Modeling) based system is used that simulates a master (scanchain-Func) interacting with a System Master BPI (Bus and Peripheral Interface) module (System-

MasterBPI) and a stimulus module (Stim).

1. scanchainFunc:
    - A templated class representing the functionality of a master in a scan chain system.
    - The getDatThread thread waits for the read data event (mFuncDatRdEve) and then notifies the algorithm about the availability of data.
    - The scanchainSequenceThread thread executes a scan chain sequence involving writing data to multiple registers and printing the results.
    - The sequence involves writing data to multiple registers (Reg1, Reg2, Reg3, Reg4, Reg5) with slave ID's corresponding to each register and printing the results.
    - It has methods for printing, writing, and reading data from the System Master BPI.
2. SystemMasterBPI (ProcPack.h):
    - A template class representing the System Master BPI.
    - It includes signals and exports for communication with the scanchainFunc.
    - It handles read and write commands, notifying the master about data availability.
3. scanchainTop:
    - A top-level module that instantiates the System Master BPI, the scanchainFunc, and a Stim module.
    - It connects the ports of these modules to enable communication.
4. Stim:
    - The Stim module generates a stimulus by feeding the TDI signal.
    - The scanchainFunc module executes a scan chain sequence based on the rising edge of the clock.
    - The System Master BPI handles read and write commands and notifies the scanchainFunc about data availability.
    - The simulation stops after a certain time.

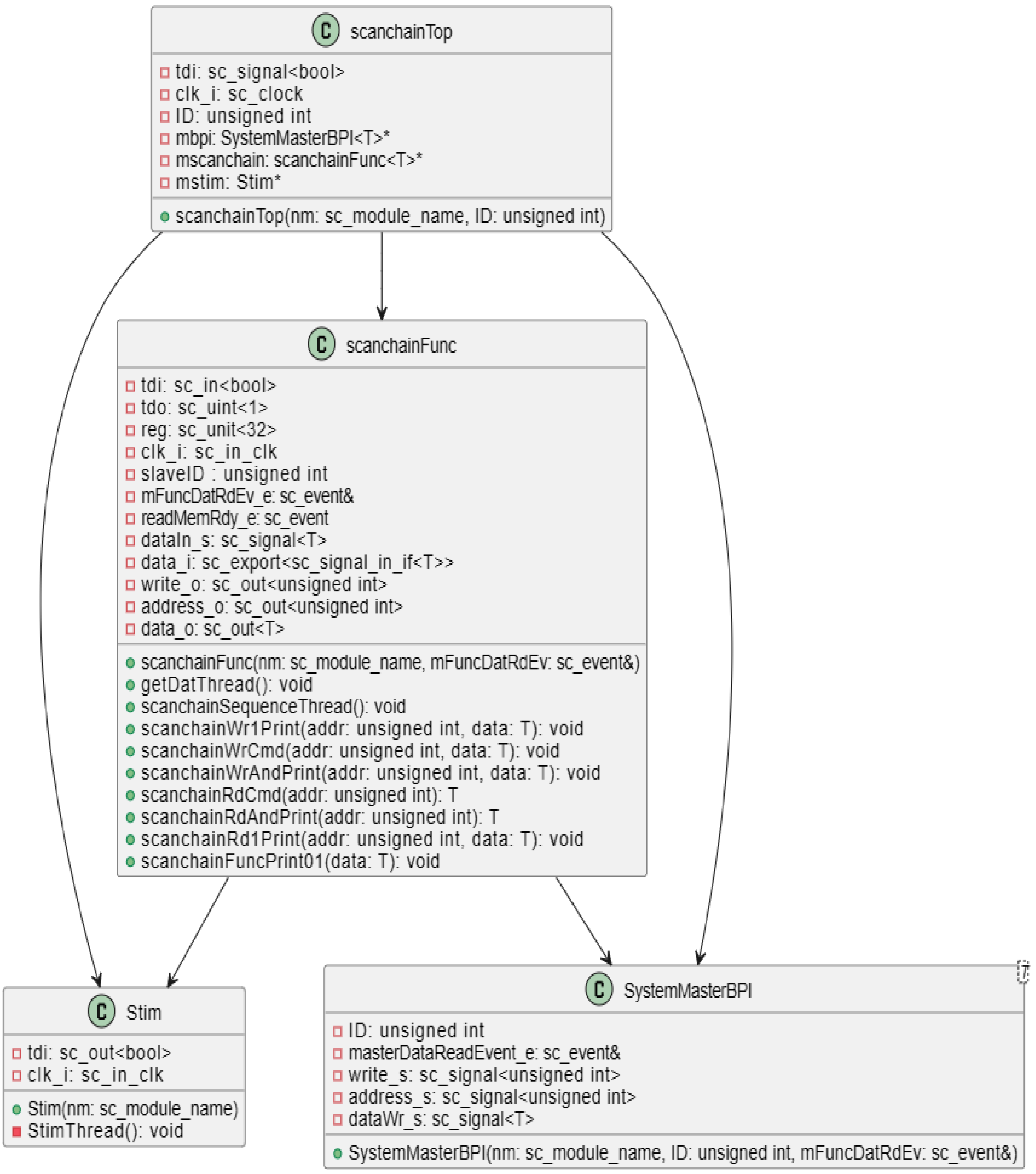
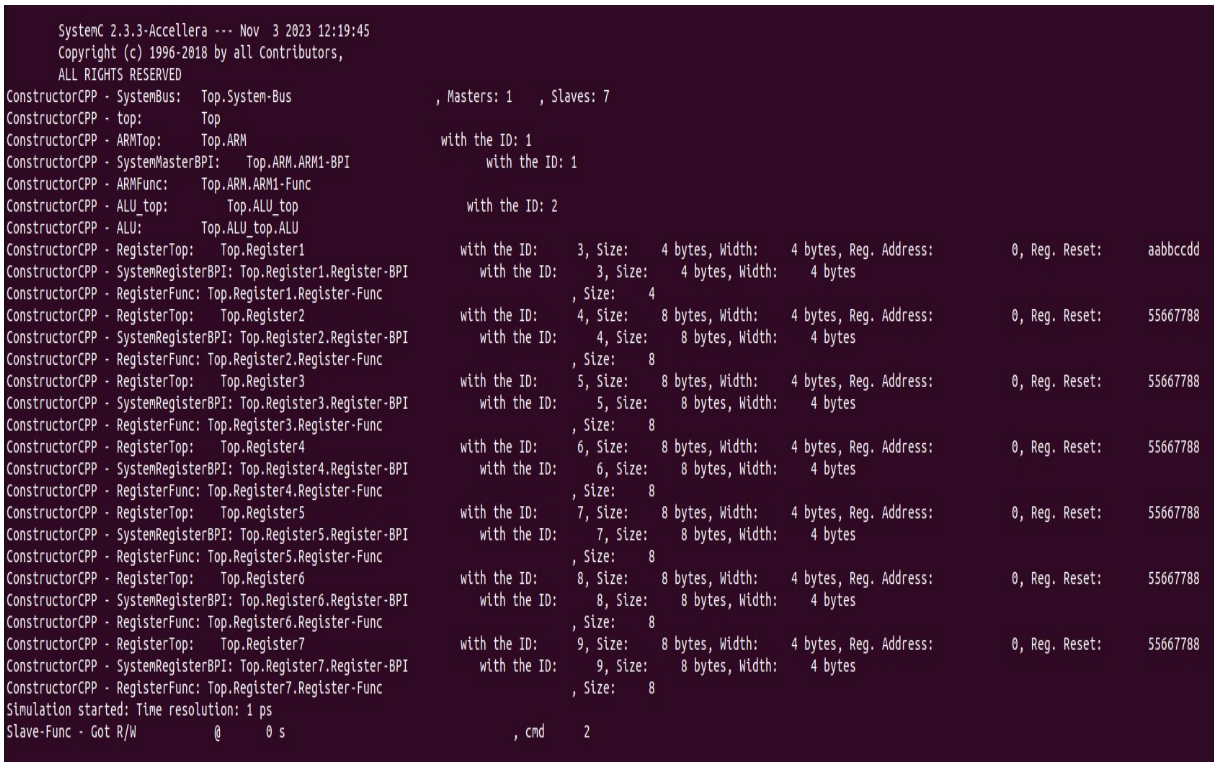

C scanchainTop
tdi: sc_signal<bool>
clk_i: sc_clock
ID: unsigned int
mbpi: SystemMasterBPI<T>*
mscanchain: scanchainFunc<T>*
mstim: Stim*
scanchainTop(nm: sc_module_name, ID: unsigned int)
C scanchainFunc
tdi: sc_in<bool>
tdo: sc_uint<1>
reg: sc_unit<32>
clk_i: sc_in_clk
slaveID : unsigned int
mFuncDatRdEv_e: sc_event&
readMemRdy_e: sc_event
dataIn_s: sc_signal<T>
data_i: sc_export<sc_signal_in_if<T>>
write_o: sc_out<unsigned int>
address_o: sc_out<unsigned int>
data_o: sc_out<T>
scanchainFunc(nm: sc_module_name, mFuncDatRdEv: sc_event&)
getDatThread(): void
scanchainSequenceThread(): void
scanchainWr1Print(addr: unsigned int, data: T): void
scanchainWrCmd(addr: unsigned int, data: T): void
scanchainWrAndPrint(addr: unsigned int, data: T): void
scanchainRdCmd(addr: unsigned int): T
scanchainRdAndPrint(addr: unsigned int): T
scanchainRd1Print(addr: unsigned int, data: T): void
scanchainFuncPrint01(data: T): void
C Stim
tdi: sc_out<bool>
clk_i: sc_in_clk
Stim(nm: sc_module_name)
StimThread(): void
C SystemMasterBPI
T
ID: unsigned int
masterDataReadEvent_e: sc_event&
write_s: sc_signal<unsigned int>
address_s: sc_signal<unsigned int>
dataWr_s: sc_signal<T>
SystemMasterBPI(nm: sc_module_name, ID: unsigned int, mFuncDatRdEv: sc_event&)

Figure 4.8: Program Logic Class Diagram

Chapter 5

Simulation

5.1 Module Constructors



```
       SystemC 2.3.3-Accellera --- Nov  3 2023 12:19:45
       Copyright (c) 1996-2018 by all Contributors,
       ALL RIGHTS RESERVED
ConstructorCPP - SystemBus:   Top.System-Bus                    , Masters: 1    , Slaves: 7
ConstructorCPP - top:         Top
ConstructorCPP - ARMTop:      Top.ARM                            with the ID: 1
ConstructorCPP - SystemMasterBPI:    Top.ARM.ARM1-BPI                  with the ID: 1
ConstructorCPP - ARMFunc:     Top.ARM.ARM1-Func
ConstructorCPP - ALU_top:        Top.ALU_top                        with the ID: 2
ConstructorCPP - ALU:         Top.ALU_top.ALU
ConstructorCPP - RegisterTop:    Top.Register1                  with the ID:      3, Size:     4 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      aabbccdd
ConstructorCPP - SystemRegisterBPI: Top.Register1.Register-BPI         with the ID:      3, Size:     4 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register1.Register-Func                       , Size:     4
ConstructorCPP - RegisterTop:    Top.Register2                  with the ID:      4, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register2.Register-BPI         with the ID:      4, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register2.Register-Func                       , Size:     8
ConstructorCPP - RegisterTop:    Top.Register3                  with the ID:      5, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register3.Register-BPI         with the ID:      5, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register3.Register-Func                       , Size:     8
ConstructorCPP - RegisterTop:    Top.Register4                  with the ID:      6, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register4.Register-BPI         with the ID:      6, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register4.Register-Func                       , Size:     8
ConstructorCPP - RegisterTop:    Top.Register5                  with the ID:      7, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register5.Register-BPI         with the ID:      7, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register5.Register-Func                       , Size:     8
ConstructorCPP - RegisterTop:    Top.Register6                  with the ID:      8, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register6.Register-BPI         with the ID:      8, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register6.Register-Func                       , Size:     8
ConstructorCPP - RegisterTop:    Top.Register7                  with the ID:      9, Size:     8 bytes, Width:     4 bytes, Reg. Address:           0, Reg. Reset:      55667788
ConstructorCPP - SystemRegisterBPI: Top.Register7.Register-BPI         with the ID:      9, Size:     8 bytes, Width:     4 bytes
ConstructorCPP - RegisterFunc: Top.Register7.Register-Func                       , Size:     8
Simulation started: Time resolution: 1 ps
Slave-Func - Got R/W          @      0 s                              , cmd      2
```

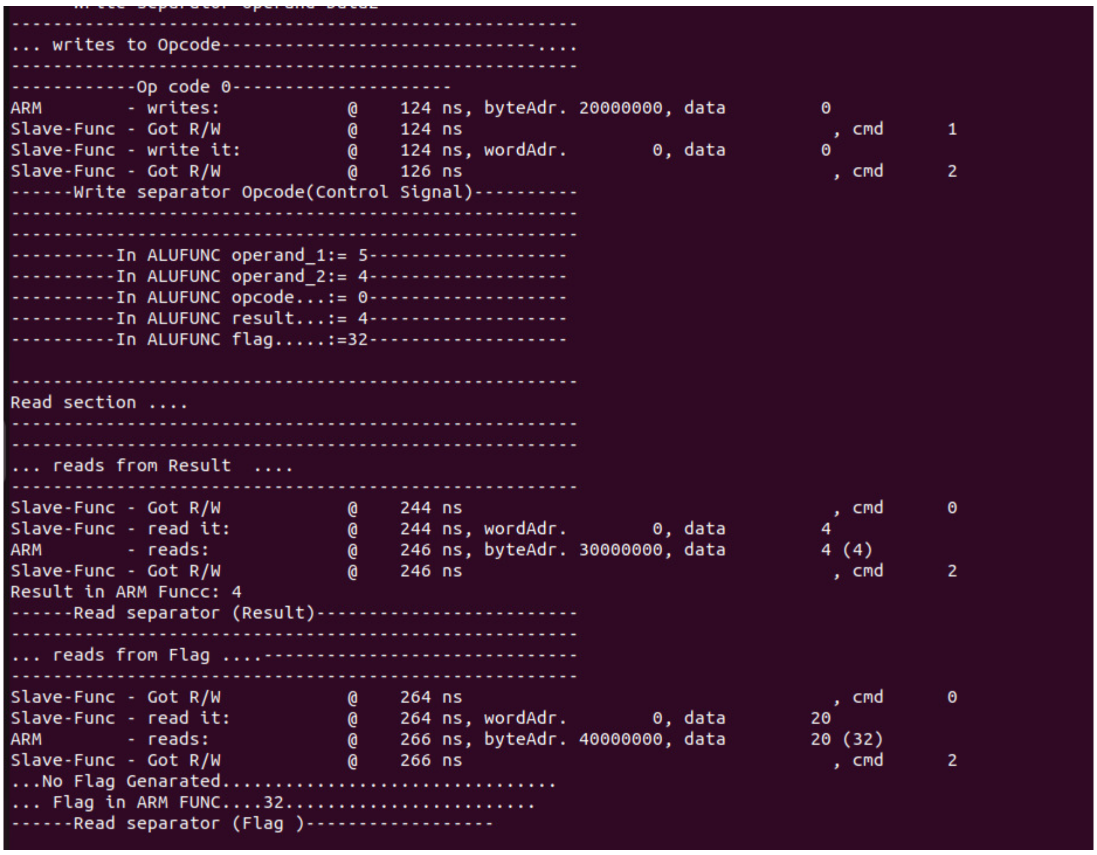
Figure 5.1: Constructor Prints

This provides details about module constructors seven registers instances within the system, labeled as Top.Register1 through Top.Register7. Each register is characterized by its unique identifier (ID), size, width, register address, and reset value. For instance, Register1 (Top.Register1) has an ID of 3, a size and width of 4 bytes, an address of 0, and a reset value. Similar information is provided for the other registers in the system, highlighting variations in size, width, and reset values. The log also indicates the start of the simulation with a time resolution of 1ps. Furthermore, a line in the log mentions the reception of a read/write command with a command ID of 2 by the "Slave-Func" module. This suggests that the system is interacting with external commands or stimuli, with the specific nature of this interaction depending on the system's implementation and the functionality associated with the mentioned command.

5.2 Read-Write by ARM / Arithmetic Operations

5.2.1 32 Bit ALU Operations

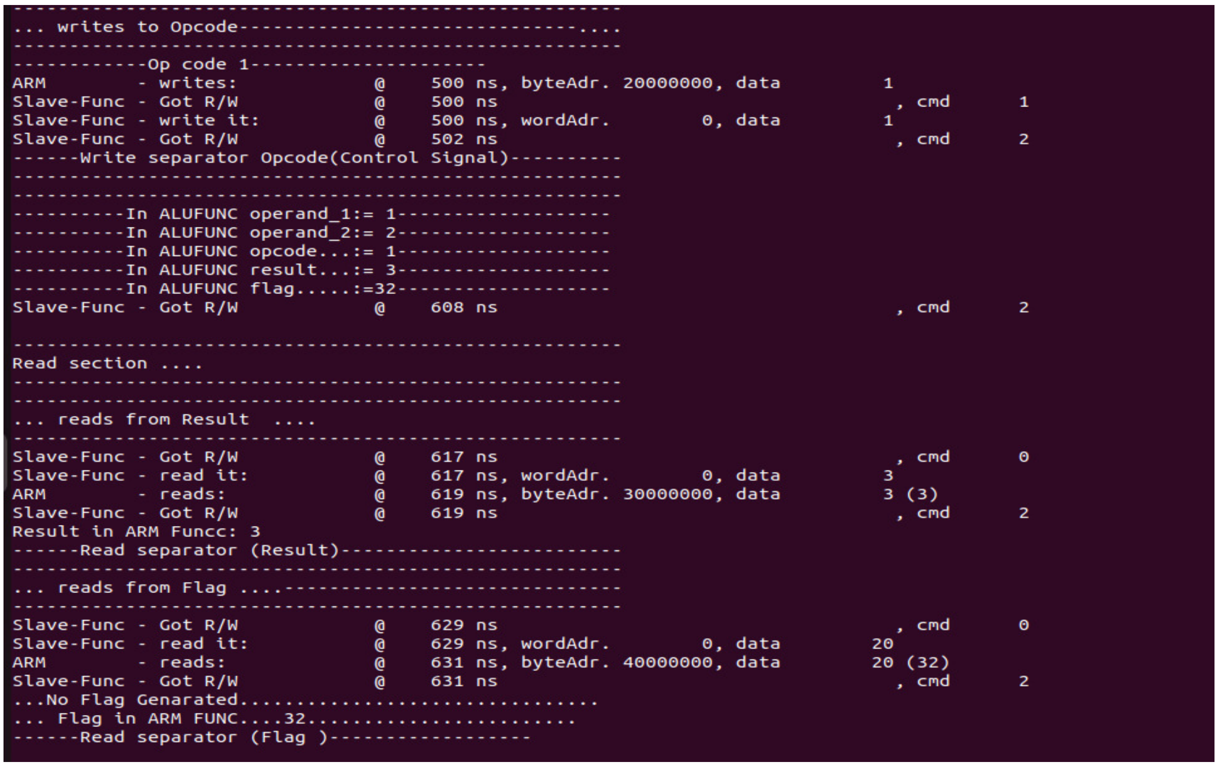
The two inputs, operands_1 is 5 and operand_2 is 4, were read from the ARM function. The AND function was then performed, yielding the desired result. Additionally, the flags operation was successfully displayed no flag generated, as seen in the Figure5.2.



```
------------------------------------------------------
... writes to Opcode------------------------------....
------------------------------------------------------
-----------Op code 0--------------------
ARM        - writes:          @    124 ns, byteAdr. 20000000, data        0
Slave-Func - Got R/W          @    124 ns                                  , cmd      1
Slave-Func - write it:        @    124 ns, wordAdr.        0, data        0
Slave-Func - Got R/W          @    126 ns                                  , cmd      2
------Write separator Opcode(Control Signal)----------
------------------------------------------------------
------------------------------------------------------
----------In ALUFUNC operand_1:= 5-------------------
----------In ALUFUNC operand_2:= 4-------------------
----------In ALUFUNC opcode...:= 0-------------------
----------In ALUFUNC result...:= 4-------------------
----------In ALUFUNC flag.....:=32-------------------

------------------------------------------------------
Read section ....
------------------------------------------------------
------------------------------------------------------
... reads from Result  ....
------------------------------------------------------
Slave-Func - Got R/W          @    244 ns                                  , cmd      0
Slave-Func - read it:         @    244 ns, wordAdr.        0, data        4
ARM        - reads:           @    246 ns, byteAdr. 30000000, data        4 (4)
Slave-Func - Got R/W          @    246 ns                                  , cmd      2
Result in ARM Funcc: 4
------Read separator (Result)-------------------------
------------------------------------------------------
... reads from Flag ....-----------------------------
------------------------------------------------------
Slave-Func - Got R/W          @    264 ns                                  , cmd      0
Slave-Func - read it:         @    264 ns, wordAdr.        0, data       20
ARM        - reads:           @    266 ns, byteAdr. 40000000, data       20 (32)
Slave-Func - Got R/W          @    266 ns                                  , cmd      2
...No Flag Genarated..............................
... Flag in ARM FUNC....32........................
------Read separator (Flag )-----------------
```

Figure 5.2: The result of AND operation and the stetted flags

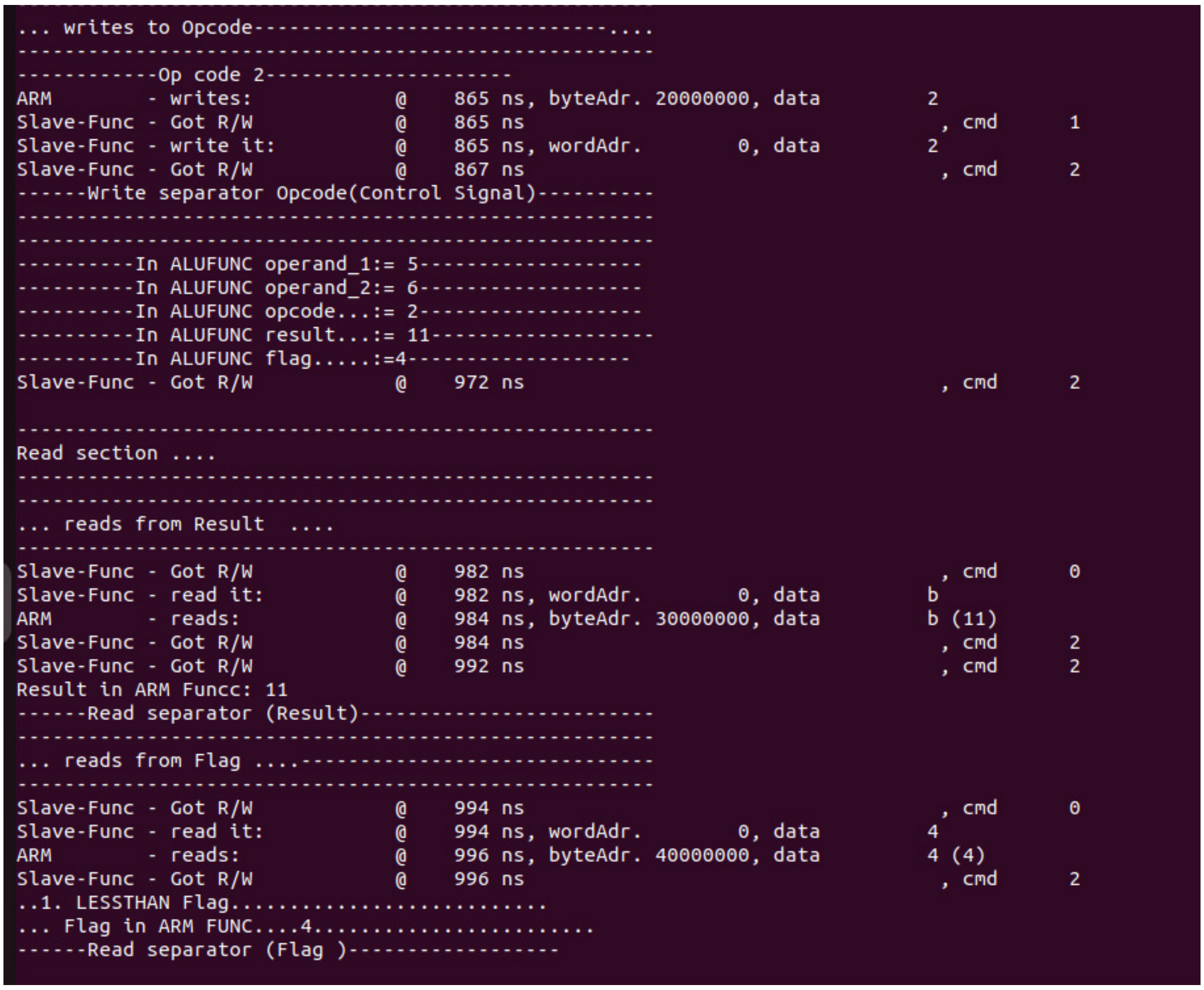
The operands_1 is 1 and operands_2 is 2 were read using the ARM function. After that, the OR function was used to get the desired result. Furthermore, the flag operations were successfully displayed. No flag was generated, as shown in Figure5.3.



```
------------------------------------------------------
... writes to Opcode-----------------------------....
------------------------------------------------------
------------Op code 1--------------------
ARM         - writes:           @    500 ns, byteAdr. 20000000, data         1
Slave-Func - Got R/W            @    500 ns                                   , cmd      1
Slave-Func - write it:          @    500 ns, wordAdr.        0, data         1
Slave-Func - Got R/W            @    502 ns                                   , cmd      2
------Write separator Opcode(Control Signal)----------
------------------------------------------------------
------------------------------------------------------
----------In ALUFUNC operand_1:= 1-------------------
----------In ALUFUNC operand_2:= 2-------------------
----------In ALUFUNC opcode...:= 1-------------------
----------In ALUFUNC result...:= 3-------------------
----------In ALUFUNC flag.....:=32-------------------
Slave-Func - Got R/W            @    608 ns                                   , cmd      2

------------------------------------------------------
Read section ....
------------------------------------------------------
------------------------------------------------------
... reads from Result  ....
------------------------------------------------------
Slave-Func - Got R/W            @    617 ns                                   , cmd      0
Slave-Func - read it:           @    617 ns, wordAdr.        0, data         3
ARM         - reads:            @    619 ns, byteAdr. 30000000, data         3 (3)
Slave-Func - Got R/W            @    619 ns                                   , cmd      2
Result in ARM Funcc: 3
------Read separator (Result)------------------------
------------------------------------------------------
... reads from Flag ....----------------------------
------------------------------------------------------
Slave-Func - Got R/W            @    629 ns                                   , cmd      0
Slave-Func - read it:           @    629 ns, wordAdr.        0, data        20
ARM         - reads:            @    631 ns, byteAdr. 40000000, data        20 (32)
Slave-Func - Got R/W            @    631 ns                                   , cmd      2
...No Flag Genarated................................
... Flag in ARM FUNC....32.........................
------Read separator (Flag )------------------
```

Figure 5.3: The result of OR operation and the stetted flags

The ARM function yielded the operands_1 as 5 and operands_2 as 6. The desired outcome was then obtained by using the addition function. In addition, as Figure5.4 below illustrates, the flags operation—set-less than flag—was successfully displayed.

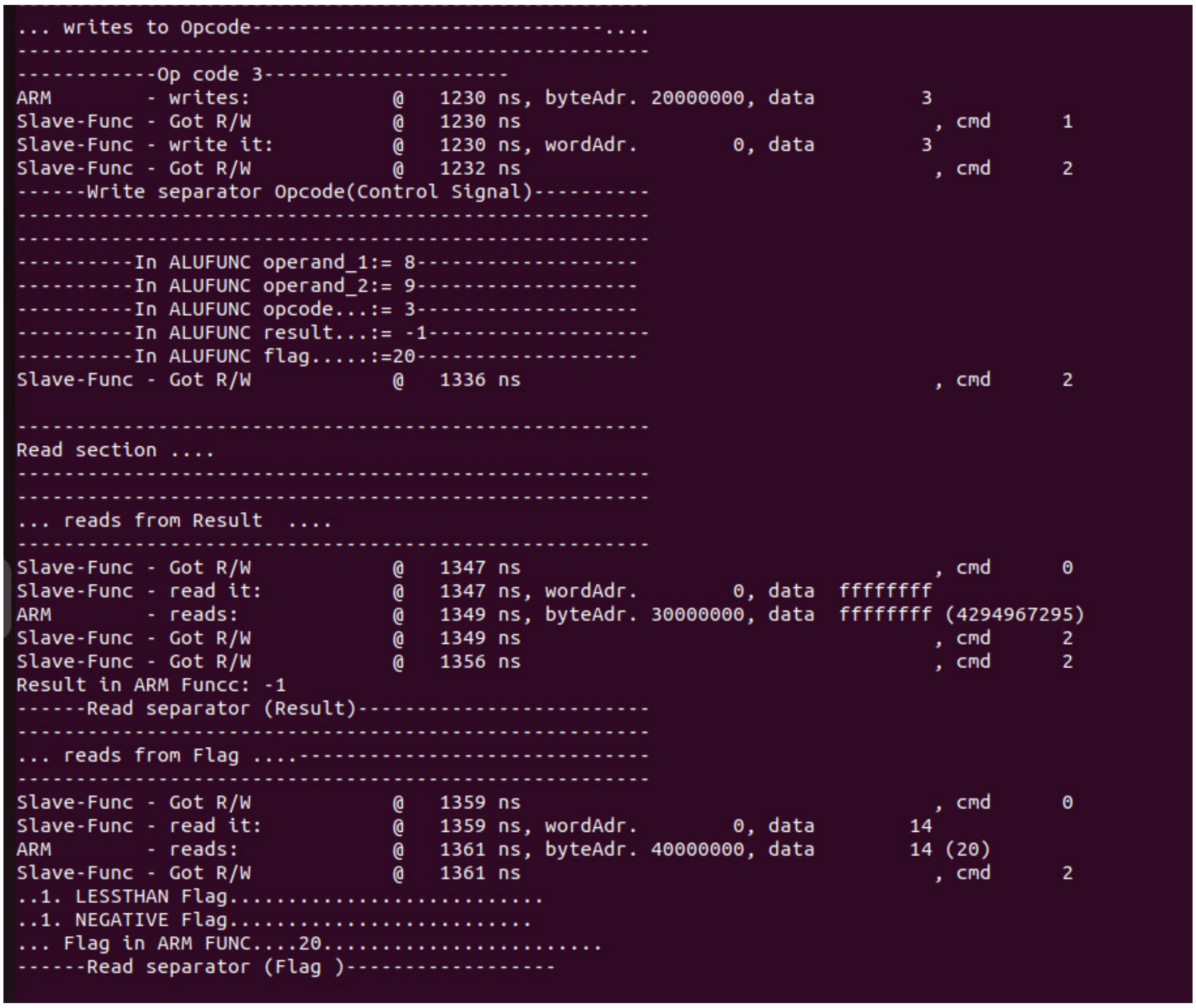


```
... writes to Opcode-----------------------------....
------------------------------------------------------
------------Op code 2---------------------
ARM        - writes:           @    865 ns, byteAdr. 20000000, data         2
Slave-Func - Got R/W           @    865 ns                                   , cmd      1
Slave-Func - write it:         @    865 ns, wordAdr.        0, data         2
Slave-Func - Got R/W           @    867 ns                                   , cmd      2
------Write separator Opcode(Control Signal)----------
------------------------------------------------------
------------------------------------------------------
----------In ALUFUNC operand_1:= 5-------------------
----------In ALUFUNC operand_2:= 6-------------------
----------In ALUFUNC opcode...:= 2-------------------
----------In ALUFUNC result...:= 11-------------------
----------In ALUFUNC flag.....:=4------------------
Slave-Func - Got R/W           @    972 ns                                   , cmd      2

------------------------------------------------------
Read section ....
------------------------------------------------------
------------------------------------------------------
... reads from Result  ....
------------------------------------------------------
Slave-Func - Got R/W           @    982 ns                                   , cmd      0
Slave-Func - read it:          @    982 ns, wordAdr.        0, data         b
ARM        - reads:            @    984 ns, byteAdr. 30000000, data         b (11)
Slave-Func - Got R/W           @    984 ns                                   , cmd      2
Slave-Func - Got R/W           @    992 ns                                   , cmd      2
Result in ARM Funcc: 11
------Read separator (Result)-------------------------
------------------------------------------------------
... reads from Flag ....-----------------------------
------------------------------------------------------
Slave-Func - Got R/W           @    994 ns                                   , cmd      0
Slave-Func - read it:          @    994 ns, wordAdr.        0, data         4
ARM        - reads:            @    996 ns, byteAdr. 40000000, data         4 (4)
Slave-Func - Got R/W           @    996 ns                                   , cmd      2
..1. LESSTHAN Flag...........................
... Flag in ARM FUNC....4........................
------Read separator (Flag )-----------------
```

Figure 5.4: The result of addition operation and the stetted flags

The ARM function yielded the operands_1 as 5 and operands_2 as 6. The desired outcome was then obtained by using the addition function. In addition, as Figure5.4 below illustrates, the flags operation—set-less than flag—was successfully displayed.

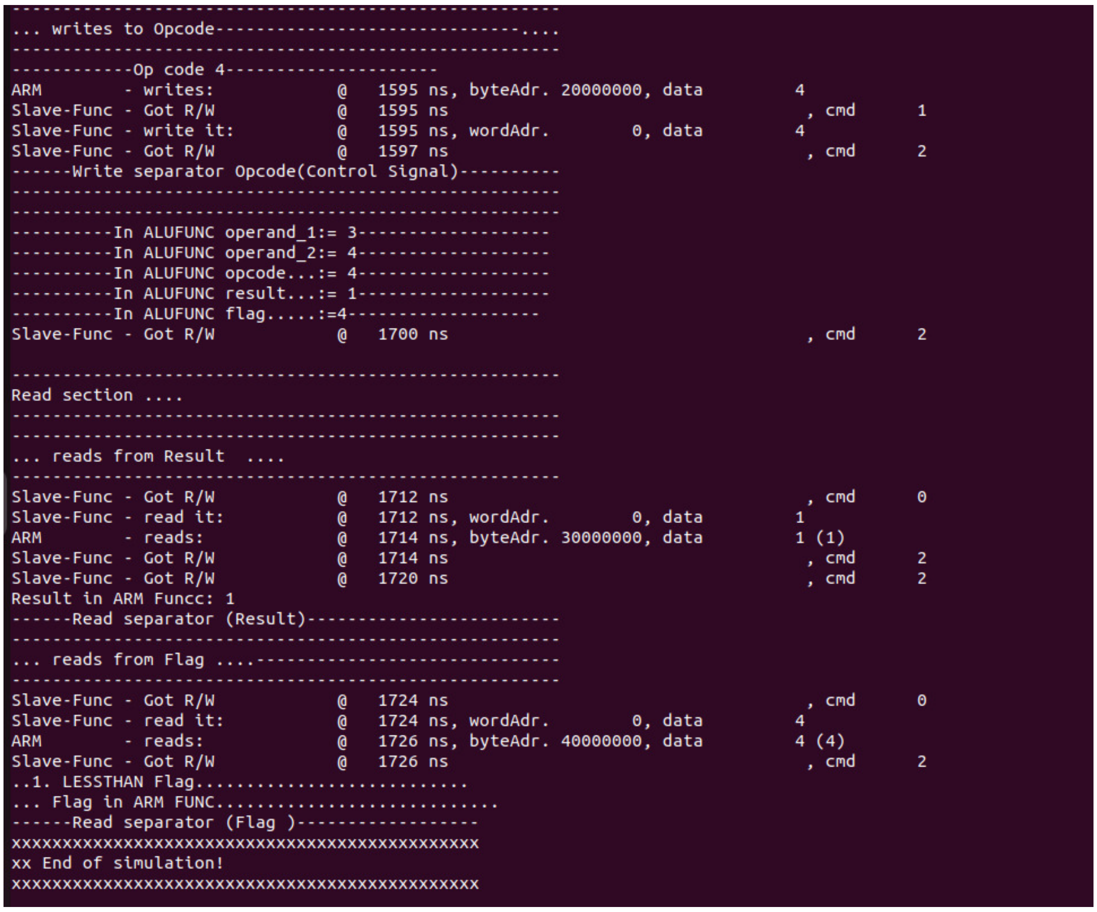
operands_1 8 and operand_2 9 from the ARM function. After that, the subtraction function was used to get the intended result. Additionally, as Figure5.5 below shows, the operation of setting negative and set-less than flags was successfully displayed.



```
... writes to Opcode-----------------------------....
------------------------------------------------------
------------Op code 3--------------------
ARM        - writes:            @   1230 ns, byteAdr. 20000000, data         3
Slave-Func - Got R/W            @   1230 ns                                   , cmd      1
Slave-Func - write it:          @   1230 ns, wordAdr.        0, data         3
Slave-Func - Got R/W            @   1232 ns                                   , cmd      2
------Write separator Opcode(Control Signal)----------
------------------------------------------------------
------------------------------------------------------
----------In ALUFUNC operand_1:= 8-------------------
----------In ALUFUNC operand_2:= 9-------------------
----------In ALUFUNC opcode...:= 3-------------------
----------In ALUFUNC result...:= -1-------------------
----------In ALUFUNC flag.....:=20-------------------
Slave-Func - Got R/W            @   1336 ns                                   , cmd      2

------------------------------------------------------
Read section ....
------------------------------------------------------
------------------------------------------------------
... reads from Result  ....
------------------------------------------------------
Slave-Func - Got R/W            @   1347 ns                                   , cmd      0
Slave-Func - read it:           @   1347 ns, wordAdr.        0, data  ffffffff
ARM        - reads:             @   1349 ns, byteAdr. 30000000, data  ffffffff (4294967295)
Slave-Func - Got R/W            @   1349 ns                                   , cmd      2
Slave-Func - Got R/W            @   1356 ns                                   , cmd      2
Result in ARM Funcc: -1
------Read separator (Result)------------------------
------------------------------------------------------
... reads from Flag ....----------------------------
------------------------------------------------------
Slave-Func - Got R/W            @   1359 ns                                   , cmd      0
Slave-Func - read it:           @   1359 ns, wordAdr.        0, data        14
ARM        - reads:             @   1361 ns, byteAdr. 40000000, data        14 (20)
Slave-Func - Got R/W            @   1361 ns                                   , cmd      2
..1. LESSTHAN Flag..........................
..1. NEGATIVE Flag.........................
... Flag in ARM FUNC....20........................
------Read separator (Flag )------------------
```

Figure 5.5: Flags that were displayed and the outcome of the subtraction operation

From the ARM function, the operand_1 as 3 and operand_2 as 4 is obtained. After that, the desired outcome was obtained by using the set-less-than function. Furthermore, the operation of set-less than flag was successfully displayed, as shown in figure 5.6



```
-------------------------------------------------------
... writes to Opcode-----------------------------....
-------------------------------------------------------
------------Op code 4--------------------
ARM        - writes:            @   1595 ns, byteAdr. 20000000, data        4
Slave-Func - Got R/W            @   1595 ns                                  , cmd      1
Slave-Func - write it:          @   1595 ns, wordAdr.        0, data        4
Slave-Func - Got R/W            @   1597 ns                                  , cmd      2
------Write separator Opcode(Control Signal)----------
-------------------------------------------------------
-------------------------------------------------------
----------In ALUFUNC operand_1:= 3------------------
----------In ALUFUNC operand_2:= 4------------------
----------In ALUFUNC opcode...:= 4------------------
----------In ALUFUNC result...:= 1------------------
----------In ALUFUNC flag.....:=4------------------
Slave-Func - Got R/W            @   1700 ns                                  , cmd      2

-------------------------------------------------------
Read section ....
-------------------------------------------------------
-------------------------------------------------------
... reads from Result  ....
-------------------------------------------------------
Slave-Func - Got R/W            @   1712 ns                                  , cmd      0
Slave-Func - read it:           @   1712 ns, wordAdr.        0, data        1
ARM        - reads:             @   1714 ns, byteAdr. 30000000, data        1 (1)
Slave-Func - Got R/W            @   1714 ns                                  , cmd      2
Slave-Func - Got R/W            @   1720 ns                                  , cmd      2
Result in ARM Funcc: 1
------Read separator (Result)-------------------------
-------------------------------------------------------
... reads from Flag ....------------------------------
-------------------------------------------------------
Slave-Func - Got R/W            @   1724 ns                                  , cmd      0
Slave-Func - read it:           @   1724 ns, wordAdr.        0, data        4
ARM        - reads:             @   1726 ns, byteAdr. 40000000, data        4 (4)
Slave-Func - Got R/W            @   1726 ns                                  , cmd      2
..1. LESSTHAN Flag..........................
... Flag in ARM FUNC............................
------Read separator (Flag )-----------------
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xx End of simulation!
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.6: The result of set-less-than operation and the stetted flags

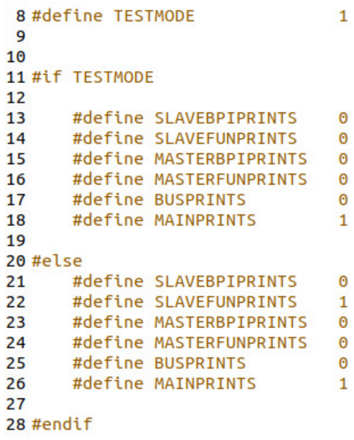
5.2.2 Test Results

Operand1	Operand2	Opcode	Results	Flags
5	4	AND	4	No Flag
1	2	OR	3	No Flag
5	6	ADD	B(11)	Less Than Flag
8	9	SUB	FFFFFFFF (-1)	Less Than Flag, Negative Flag
3	4	COMPARE LESS THAN	Prints 1 for True	Less than Flag

Table 5.1: Test Results of ALU operations

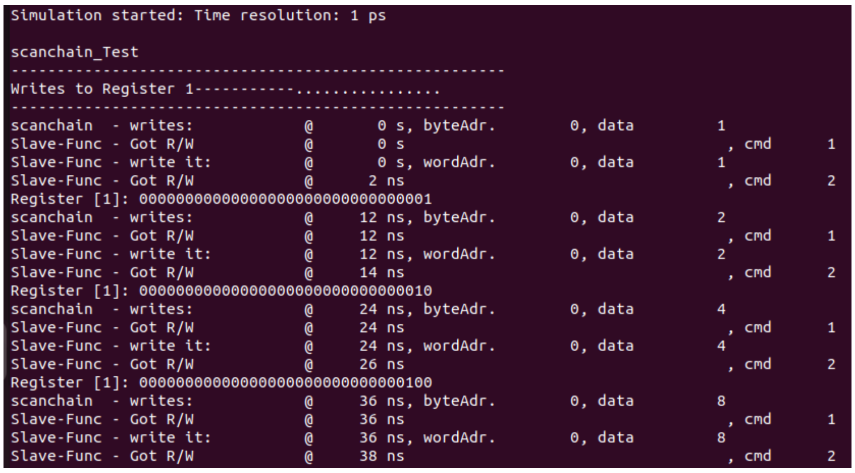
5.3 Scan Chain

- Scan chain mode or test mode is activated by enabling the Test Mode in ProcPack.h as in Figure 5.7.
- Enabling the SLAVEFUNPRINTS and MAIN PRINTS displays the slave-func timing at each position of data communication and writing to the register along with the address and data positioning value at each shift level as shown in Figure 5.8.
- Figure 5.9 displays the shifting of 1-bit TDI from LSB to MSB in a 32-bit Register of reg1. Once the data is shifted in each register parallely, the TDO is fetched from the MSB. The same operation occurs with the other registers.



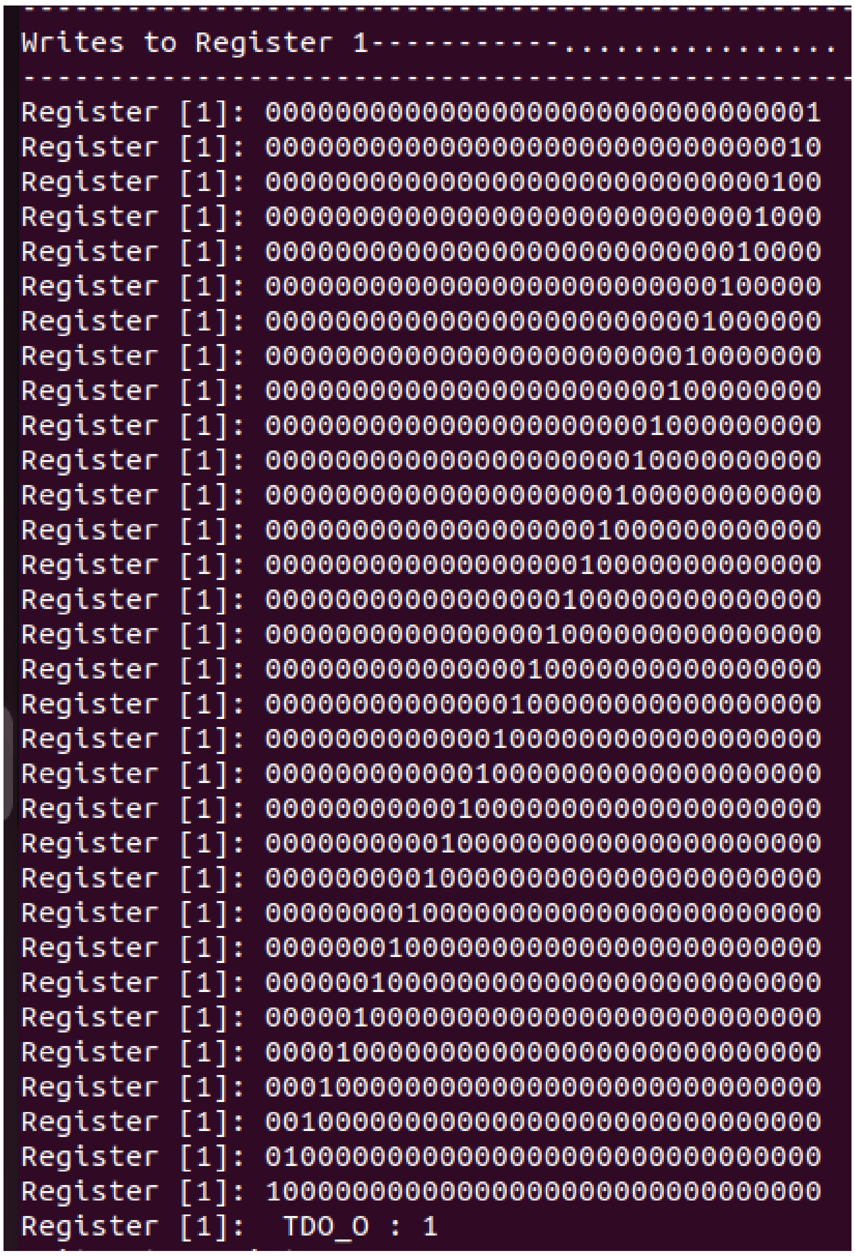
```
#define TESTMODE            1


#if TESTMODE

    #define SLAVEBPIPRINTS    0
    #define SLAVEFUNPRINTS    0
    #define MASTERBPIPRINTS   0
    #define MASTERFUNPRINTS   0
    #define BUSPRINTS         0
    #define MAINPRINTS        1

#else
    #define SLAVEBPIPRINTS    0
    #define SLAVEFUNPRINTS    1
    #define MASTERBPIPRINTS   0
    #define MASTERFUNPRINTS   0
    #define BUSPRINTS         0
    #define MAINPRINTS        1

#endif
```

Figure 5.7: Test Mode



```
Simulation started: Time resolution: 1 ps

scanchain_Test
------------------------------------------------------
Writes to Register 1----------................
------------------------------------------------------
scanchain  - writes:          @       0 s, byteAdr.       0, data        1
Slave-Func - Got R/W          @       0 s                                , cmd      1
Slave-Func - write it:        @       0 s, wordAdr.       0, data        1
Slave-Func - Got R/W          @      2 ns                                , cmd      2
Register [1]: 00000000000000000000000000000001
scanchain  - writes:          @     12 ns, byteAdr.       0, data        2
Slave-Func - Got R/W          @     12 ns                                , cmd      1
Slave-Func - write it:        @     12 ns, wordAdr.       0, data        2
Slave-Func - Got R/W          @     14 ns                                , cmd      2
Register [1]: 00000000000000000000000000000010
scanchain  - writes:          @     24 ns, byteAdr.       0, data        4
Slave-Func - Got R/W          @     24 ns                                , cmd      1
Slave-Func - write it:        @     24 ns, wordAdr.       0, data        4
Slave-Func - Got R/W          @     26 ns                                , cmd      2
Register [1]: 00000000000000000000000000000100
scanchain  - writes:          @     36 ns, byteAdr.       0, data        8
Slave-Func - Got R/W          @     36 ns                                , cmd      1
Slave-Func - write it:        @     36 ns, wordAdr.       0, data        8
Slave-Func - Got R/W          @     38 ns                                , cmd      2
```

Figure 5.8: Simulation Output with Slave-Func



```
-------------------------------------------------
Writes to Register 1----------...............
-------------------------------------------------
Register [1]: 00000000000000000000000000000001
Register [1]: 00000000000000000000000000000010
Register [1]: 00000000000000000000000000000100
Register [1]: 00000000000000000000000000001000
Register [1]: 00000000000000000000000000010000
Register [1]: 00000000000000000000000000100000
Register [1]: 00000000000000000000000001000000
Register [1]: 00000000000000000000000010000000
Register [1]: 00000000000000000000000100000000
Register [1]: 00000000000000000000001000000000
Register [1]: 00000000000000000000010000000000
Register [1]: 00000000000000000000100000000000
Register [1]: 00000000000000000001000000000000
Register [1]: 00000000000000000010000000000000
Register [1]: 00000000000000000100000000000000
Register [1]: 00000000000000001000000000000000
Register [1]: 00000000000000010000000000000000
Register [1]: 00000000000000100000000000000000
Register [1]: 00000000000001000000000000000000
Register [1]: 00000000000010000000000000000000
Register [1]: 00000000000100000000000000000000
Register [1]: 00000000001000000000000000000000
Register [1]: 00000000010000000000000000000000
Register [1]: 00000000100000000000000000000000
Register [1]: 00000001000000000000000000000000
Register [1]: 00000010000000000000000000000000
Register [1]: 00000100000000000000000000000000
Register [1]: 00001000000000000000000000000000
Register [1]: 00010000000000000000000000000000
Register [1]: 00100000000000000000000000000000
Register [1]: 01000000000000000000000000000000
Register [1]: 10000000000000000000000000000000
Register [1]:  TDO_O : 1
```

Figure 5.9: Simulation Output with TDI shift

Chapter 6

Test and Debug

6.1 Scan Chain

- Scan chain allows for the serial loading and capturing of data in a shift-register model, providing a way to observe and manipulate the internal states of the design. It involves creating a chain of flip-flops that allows for the efficient scanning of data into and out of the registers..
- In our design, we have implemented Scan Chain for a set of registers and captured the data via scan-in and scan-out flow parrallely to all the registers.
- Overall, the performance of the RISC-V ALU is good, and the expected outcome and flag status are obtained. Additionally, the status of the flags and the outcomes of arithmetic and logic operations are written back into the ARM.

Bibliography

[1] Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Book by David A Patterson and John L. Hennessy

[2] VLSI Test Principles and Architectures. Book by Laung-Terng Wang, Cheng-Wen Wu and Xiaoqing Wen

[3] IEEE Standard Test Access Port and Boundary-Scan Architecture (2008)