

Assignment 2 Arithmetic Logic Unit

The goal of our second assignment is to implement a simple Arithmetic Logic Unit (ALU) for a 32-bit RISC-V processor that supports the RV32I instruction set architecture (ISA) (see [The RISC-V Instruction Set Manual Volume I](#), pp. 23 – 40 for details). The assignment also introduces a UVM testbench. The Universal Verification Methodology (UVM) is a standard methodology in industry for verifying hardware designs.

Before you start with the tasks, make yourself familiar with the directory for the second assignment (02_ALU) in your forked repository. The folder already contains a Chisel project and provides you with the basic structure for this task. A more detailed introduction to UVM and its components is provided at the end of this assignment.

Specification

The ALU to be designed is a purely combinational Chisel module for an RV32I core that accepts two 32-bit unsigned operands (`operandA`, `operandB`) and a control field `operation` encoded as a `ChiselEnum`. The ALU can perform the following operations:

- ADD: addition
- SUB: subtraction
- AND: logical bitwise AND
- OR: logical bitwise OR
- XOR: logical bitwise XOR
- SLL: shift left logical
- SRL: shift right logical
- SRA: shift right arithmetic
- SLT: set less than
- SLTU: set less than (unsigned interpretation)
- PASSB: pass `operandB` to the result output

Every operation produces a 32-bit result on `aluResult`. Arithmetic addition and subtraction use two's-complement semantics with modulo- 2^{32} wraparound. Bitwise operators act per bit on the inputs. Shift operations use only the low five bits of `operandB` as the shift amount to match RV32 semantics, with `SRL` performing a logical right shift and `SRA` performing an arithmetic right shift by interpreting `operandA` as signed. The comparison operations return 0 or 1 (zero-extended to 32 bits), with `SLT` treating operands as signed and `SLTU` as unsigned, while `PASSB` forwards `operandB` unchanged to the output. The ALU module maintains no internal state. No new inputs are to be added beyond the specified ones. In case you see the need for an additional output, you are allowed to add it but should justify its necessity in the discussion session.

Task 2.1 Preparation

Before you start with the implementation, please answer the following questions:

1. What RV32I ISA instructions use which of the described ALU operations?
2. What should happen, if the ALU receives an input other than the specified operations?
3. What corner cases should a testbench definitely cover for the described implementation?

Task 2.2 Test-Driven Implementation

Implement a simple Arithmetic Logic Unit (ALU) for the RV32I ISA according to the specification above in the provided Chisel project and apply test-driven development.

Test-driven development (TDD) is a design practice in which developers write tests *before* writing the code that will make those tests pass. In the ADS I Class Project, you will use TDD to design the ALU in small, focused steps: Define the expected behavior through a failing test first, then implement just enough code to meet that expectation, and finally refine your solution while ensuring the tests continue to pass. This approach does not only help to clarify requirements early but also leads to cleaner, more reliable, and easier-to-maintain code.

The `src/test/scala/` directory contains a file `alu_tb.scala` that holds a basic Chisel testbench containing only a single test case for an ADD operation. Implement tests for an ALU operation first, then implement the corresponding functionality in `src/main/scala/alu.scala` to make the test pass. Make sure to cover corner cases in your tests. Repeat this process until all ALU operations are implemented and tested.

Implementing first and testing afterwards can lead to writing tests under the influence of your own implementation. If you misunderstood the task or misinterpreted the specification, it is likely that you will also design the tests that way. In the worst case, all tests might pass, although the implementation is faulty. TDD avoids this fallacy by forcing you to think about the expected behavior first and to write tests that reflect this understanding. Only after that, you implement the functionality to make the tests pass. Run `sbt test` after writing tests to verify they fail before implementation

If you are working in a team, the best method is to have the tests and implementation designed by different people. This reduces the influence of the user's own interpretation of the specification and ensures that the design is checked by at least one other person.

Task 2.3 Test your Design in UVM

Writing directed test cases in Chisel (or SystemVerilog) works for small designs, but it does not scale well for large projects. The stimuli are hand-scripted, hard to reuse across different components, and offer little guidance on what is actually covered. Scaling the design therefore leads to an exploding number of test cases with diminishing returns. In industry, people took a different approach and developed verification methodologies and libraries to build reusable, coverage-driven testbenches. The Universal Verification Methodology (UVM) provides reusable, constrained-random sequences to explore corner cases automatically, and methods to assess the gained coverage. Its modular structure makes larger designs both more thorough and more maintainable.

In this task, you will get a first impression of UVM by applying a UVM testbench to verify your ALU implementation. The UVM environment is already provided in the `02_ALU/src/test/uvm/` directory.

- a) **Getting Started with UVM** Familiarize yourself with the structure of a UVM testbench as described in the next section, then run the UVM testbench to verify your ALU implementation.

b) Create a Sequencer Item Design a UVM sequence item class for the ALU testbench in `src/test/uvm/alu_seq_item.sv`. The sequence item represents a single transaction to the ALU and serves as the atomic unit of stimulus in the UVM environment. The enum type for the ALU operations is already defined in `alu_tb_config_pkg.sv` as `ALUOp`.

Your sequence item should extend the base class `uvm_sequence_item` and define the following fields with randomized inputs:

- `operandA`: a 32-bit random value representing the first ALU operand
- `operandB`: a 32-bit random value representing the second ALU operand
- `operation`: a random ALU operation from the set of defined operations
- `aluResult`: a 32-bit field to store the expected result from the ALU

Add a constraint named `aluOp_constraint` that restricts the `operation` field to only valid ALU operations.

The UVM macros `uvm_object_utils_begin` and `uvm_object_utils_end` register the class with the UVM factory and define the field utilities. Within these macros, use `uvm_field_int` for integer fields and `uvm_field_enum` for the operation field to enable automatic printing, copying, and comparison.

The skeleton file already provides a `new()` constructor that initializes the UVM object by calling `super.new(name)`, which sets up the UVM infrastructure and registers the object's instance name. The skeleton also includes a `convert2str()` virtual function that returns a formatted string representation of the sequence item, displaying all four fields in hexadecimal and enum format (e.g., "operandA: 0x0000abcd, operandB: 0x12345678, operation: ADD, aluResult: 0x..."). This function is essential for printing transaction details during simulation and aids in debugging.

Complete the class by defining the four randomized fields and the `aluOp_constraint` constraint as described above.

c) Run the UVM Testbench After implementing the sequence item, run the UVM testbench to verify your ALU implementation. UVM testbenches are typically executed using a SystemVerilog simulator (e.g., ModelSim, VCS, or Verilator). Xilinx Vivado also includes a built-in simulator that supports UVM and offers free educational licenses for students. From the chosen simulator, you can run the UVM testbench by calling the `alu_sim.tcl` script located in `02_ALU`. In Vivado, this is done using the following command:

```
1 $ source ./alu_sim.tcl
```

The UVM testbench expects the generated RTL of your ALU design in `generated-src/ALU.v`, so make sure to compile your Chisel project first using `sbt run`.

After running the UVM testbench, check the simulation log for any errors or failed assertions. The testbench results are typically reported in the console output or a log file generated by the simulator (e.g., `vivado.log`) and should contain a summary of passed and failed tests:

```
1 Scoreboard summary: Passes: 283, Fails: 0, Total: 283, Pass rate: 100.00 %
```

d) Coverage Report Have a look into the executed test cases in your simulation report and think about how well your UVM testbench covers all possible cases of the ALU functionality. Find out how UVM handles coverage collection and reporting. You do not need to implement functional coverage for this assignment, but be prepared to present the concept of coverage in UVM during the discussion session.

Introduction to UVM

UVM (“Universal Verification Methodology”) is an industry standard for testing hardware designs. At its core, UVM is a SystemVerilog class library and a set of conventions for building reusable, coverage-driven testbenches for digital hardware. It structures verification around transactions and sequences. By combining constrained-random stimulus with functional coverage, UVM scales from simple IP blocks to complex SoCs and helps to reach verification goals efficiently.

A UVM testbench comprises the following steps:

generate (constrained-random) transactions → drive pins → monitor → check → measure coverage

UVM scales to complex designs by relying on reusable components. Its constrained-random stimuli help to expose corner cases, while coverage metrics help to reach the defined verification goals. In practice, design teams complement constrained-random stimulus with a small set of directed tests to target bring-up scenarios, i. e., the initial system setup, and specific corner cases.

Structure of a UVM Testbench

A UVM testbench is a layered, component-based architecture that separates stimulus generation, interaction with the DUT (design under test), observation, checking, and configuration to enable reusable, coverage-driven verification.

Applying a full-blown UVM testbench to verify your ALU might feel like using a sledgehammer to crack a nut. Indeed, the initial effort to create a UVM environment and to understand all the necessary components is much higher compared to a simple testbench in Chisel or SystemVerilog. UVM, on the other hand, can play out its advantages when applied to complex designs. For example, in an SoC with AXI and APB buses, and a UART connection, you can plug in reusable AXI/APB/UART agents and coordinate end-to-end scenarios with a virtual sequencer. With the help of the UVM factory you can swap an AXI memory model for a “with-stalls” or “error-injecting” variant, exercising timing and robustness corners without editing the environment. Monitors feed scoreboards and functional coverage across protocols, providing measurable closure and scalable reuse that a static HDL testbench cannot deliver.

In our class project, we focus on the core functionalities of UVM and provide you with a first impression of what this verification methodology is capable of. This section introduces the basic components of a UVM testbench by using the UVM environment for the ALU in `02_ALU/src/test/uvm/` as an example.

Sequence Item: A sequence item is the atomic unit of stimulus, carrying randomized fields and constraints. A sequence item for the ALU consists of the two operands and the operation type. In more complex designs, the sequence type could, e.g., also be an ISA instruction or a bus transaction. You can find the sequence item in `alu_seq_item.sv`. Figure 1 shows the definition and main components of a sequence item for our ALU. New sequence items extend the base class `uvm_sequence_item` and define the inputs and the corresponding range of values. For example, the two operands passed to the ALU can be any (`rand`) 32-bit value, while the ALU operation is constrained to take any of the defined operations (`aluOp_constraint`). This is what we call “constrained random” in testing or verification environments.

Sequence: The sequencer takes the defined sequence items and produces a stream of transactions. Thereby, the sequencer controls stimulus patterns and length, which can also be made constrained-random. The sequencer is located in `alu_sequencer.sv`. Figure 2 depicts a simple example sequence that generates a sequence of N items. Note that the iteration count can also be randomized.

Sequencer: The sequencer arbitrates between sequences and delivers items to the driver via TLM (“Transaction Level Modeling”) handshakes. In our class project environment, the sequencer is integrated in the `alu_agent.sv` file.

```
1 class alu_seq_item extends uvm_sequence_item;
2   rand bit [31:0] operandA, operandB;
3   rand ALUOp operation;
4   bit [31:0]aluResult;
5
6   `uvm_object_utils_begin(alu_seq_item)
7     `uvm_field_int (operandA, UVM_DEFAULT)
8     `uvm_field_int (operandB, UVM_DEFAULT)
9     `uvm_field_enum (ALUOp, operation, UVM_DEFAULT)
10    `uvm_field_int (aluResult, UVM_DEFAULT)
11  `uvm_object_utils_end
12
13  constraint aluOp_constraint{
14    operation inside {ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU, PASSB};
15  }
16
17  [...]
18
19 endclass
```

Figure 1: Example of an ALU sequence item

```
1 class alu_sequence extends uvm_sequence;
2   `uvm_object_utils(alu_sequence)
3
4   function new(string name = "alu_sequence");
5     super.new(name);
6   endfunction
7
8   rand int iteration_count;
9   constraint iteration_count_c{
10     soft iteration_count N;
11   }
12
13   virtual task body();
14     for (int i=0; i<iteration_count; i++) begin
15       alu_seq_item item = alu_seq_item::type_id::create(item);
16       start_item("item");
17       item.randomize();
18       `uvm_info(get_type_name(), UVM_MEDIUM)
19       finish_item(item);
20     end
21     `uvm_info(get_type_name(), UVM_MEDIUM)
22   endtask
23
24 endclass
```

Figure 2: A simple sequence that generates N items

Driver: The driver (`alu_driver.sv`) converts high-level transactions (i.e., items) into pin-level activity on the virtual interface. For example, the operations that we defined for the sequence items in Figure 1 are described as “ADD”, “SUB”, etc., but need to be assigned to the DUT as bit-level signals. The driver provides the proper stimulus to the DUT, while checking the responses is done by the monitor and scoreboard.

Monitor: The monitor in `alu_monitor.sv` reads the output data on the interface, reconstructs observed transactions, and publishes them via an analysis port. It can also use these information to trigger functional coverage and checkers or scoreboards.

Agent: The agent (`alu_agent.sv`) bundles sequencer, driver, and monitor and can be active (drives) or passive (observes). In larger scenarios, it allows reuse across tests and environments with consistent configuration points.

Scoreboard: The scoreboard implements a reference model and compares expected vs. observed behavior. By keeping track of the sequences test cases and expected results, it can report mismatches. The scoreboard is located in `alu_scoreboard.sv`. For a better portability, the scoreboard should act on transaction level.

Environment: The environment in `alu_env.sv` instantiates agents, scoreboards, monitors, and connects TLM networks.

Test: `alu_test.sv` is the top-level configuration point that selects the environment, sets config database values, and initiates sequences.

Interface & Config: The interface (`alu_if.sv`) carries DUT pins and clocking blocks for clean timing. The config package (`alu_tb_config_pkg.sv`) defines parameters, typedefs, and enums in a central file.

Top Module: `alu_tb.sv` instantiates the DUT and its interfaces, generates clock and reset signals, sets virtual interfaces into the config database, then calls `run_test()`. It should contain no verification logic beyond wiring.