ALU

Learning Goal: Testbenches, Arithmetic operations.

Requirements: Verilator, GTKWave.

1 Introduction

In this lab you will implement a complete Arithmetic-Logic Unit (ALU) and practice writing and using testbenches, as well as making gate-level simulations.

2 ALU Description

An *Arithmetic-Logic Unit* (ALU) is a combinatorial circuit that performs arithmetic and logic operations. It is the central execution unit of a CPU and its complexity can vary.

A simple ALU has two inputs for the operands, one input for a control signal that selects the operation and one output for the result. Figure 1 shows the common representation of an ALU.

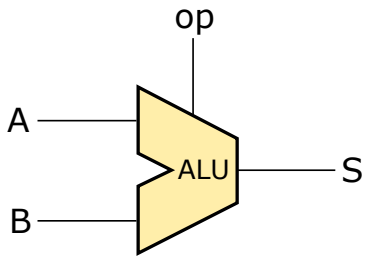

Figure 1: A simple ALU with two inputs, a control signal and an output.

For this lab, you will implement a 32-bit ALU with 4 internal units. The available operations and their corresponding encoding are listed in Table 1. The 6-bit **op** control signal selects which operation to execute. Figure 2 shows the internal architecture of the ALU.

Table 1: ALU operations and their encoding.

Operation	Operation Type	Opcode
$A + B$	Add/Sub	000$\phi\phi\phi$
$A - B$		001$\phi\phi\phi$
$A = B$	Comparison	011000
$A \neq B$		011001
$A < B$ (signed)		011100
$A \geq B$ (signed)		011101
$A < B$ (unsigned)		011110
$A \geq B$ (unsigned)		011111
$A \oplus B$ (XOR)	Logical	100100
$A \vee B$ (OR)		100110
$A \wedge B$ (AND)		100111
$A \ll B$ (SLL)	Shift	110001
$A \gg_l B$ (SRL)		110101
$A \gg_a B$ (SRA)		111101

ϕ = *don't care*, **0**/**1** = special bit

- The two most significant bits (i.e., $\mathbf{op}_{5..4}$) select the operation type (e.g., Add/Sub, Comparison, Logical, Shift).
- The $\mathbf{op}_3$ bit is the **special bit**. It activates the subtraction mode of the **Add/Sub** unit, is always set for the comparison unit, and determines whether a right shift is arithmetic (SRA) or logical (SRL) in the shift unit. It is unused (set to 0) for the logical unit.
- The $\mathbf{op}_{2..0}$ bits select a specific operation within each unit.

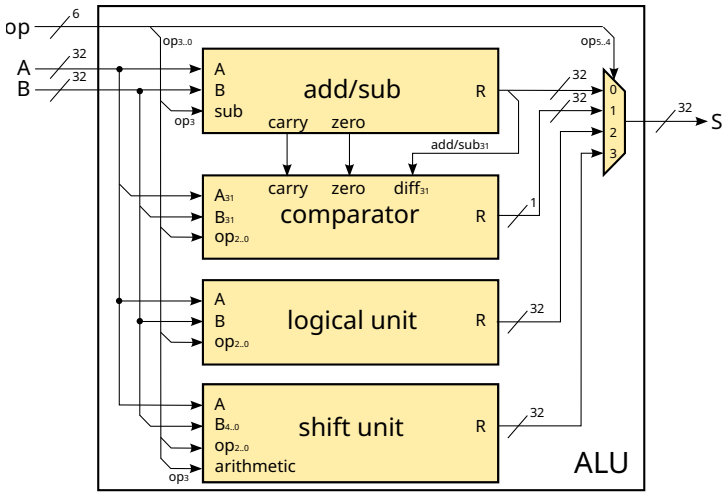

Figure 2: The internal architecture of the ALU.

In the following subsections, each unit is described in more details. For the moment, you can skip these and start with the exercises of Section 3.

2.1 Add/Sub

The **Add/Sub** unit performs 32-bit additions and subtractions on unsigned and two's complement signed numbers.

- Input **sub** activates the **subtraction mode**.
- Output **carry** is the **carry out** of the internal adder.
- Output **zero** is high when the result equals 0.

Figure 3 shows the internal architecture of the **Add/Sub** unit.

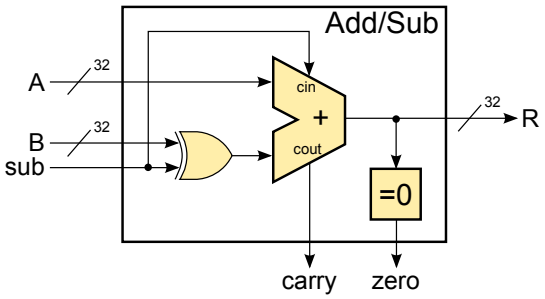

Figure 3: The internal architecture of the **Add/Sub** unit.

When the **subtraction mode** is activated, the second operand should become the two's complement of **B**. This conditional inversion of B can be performed with 32 XOR gates: when **sub** is high, then **B** is inverted; otherwise, it keeps its original value. This is shown in Figure 3 where every bit from the **B** input is XORed with the **sub** (you can do this in Verilog by replicating **sub** 32 times and XORing with **B**).

The conditional increment in case of a **subtraction mode** can be done by connecting the **sub** signal directly to the **carry in** of the adder. As a result we have $A + \overline{B} + 1$ which is equivalent to $A - B$.

Some 4-bit operation examples are illustrated below. The first operand corresponds to **A**, the second to the XOR output (i.e., either **B** or *not* **B**) and the third is the **carry in** input of the adder, which is equal to the **sub** input. The result holds the **carry out** bit in its most significant bit shown in **bold**.

Finally—if you didn't already know—to generate the carry from adding up two 32-bit numbers, you can zero-extend the two operands to 33 bits and then add them in Verilog, the 33rd bit of the result is the desired carry.

$3 + 3 = 6$	$7 + (-1) = 6$	$5 - 4 = 1$	$(-5) - 0 = -5$
0011	0111	0101	1011
0011	1111	1011	1111
$+$ 0	$+$ 0	$+$ 1	$+$ 1
00110	**1**0110	**1**0001	**1**1011

2.2 Comparator

The **Comparator** unit performs six comparison operations: *equal*, *not equal*, signed *less than*, signed *greater or equal*, unsigned *less than*, and unsigned *greater or equal*. It computes these comparisons using the result of the subtraction coming from the **Add/Sub** unit. Thus, the subtraction mode is always set for comparison operations.

- Input $\mathbf{op}_{2..0}$ selects the type of comparison.
- Output **r** is the result of the comparison (0=false, 1=true).
- Inputs **zero**, **carry**, $\mathbf{a}_{31}$, $\mathbf{b}_{31}$ and $\mathbf{diff}_{31}$ are used to perform the comparison.

2.2.1 *Equal* and *not equal*

The *equal* (EQ) comparison result is directly driven by the **zero** input signal: if **a** equals **b**, then the subtraction result is zero. The *not equal* (NEQ) result is simply the inverse of **zero**.

2.2.2 Unsigned *less than* and *greater or equal*

a is less than **b** in unsigned arithmetic (LTU), iff **carry** is low and **zero** is low. Therefore, **a** is greater or equal to **b** (GEU) iff **carry** is high or **zero** is high. You can find a proof below to convince yourself.

Proof:
Let n be the bitwidth of the **adder** inputs, A and B.
Let D be the subtraction output including the carry out. Its bitwidth is $n+1$.

If **carry** is 0 and **zero** is 0, we have that $0 < D < 2^n$.
If **carry** is 1 or **zero** is 1, we have that $D \geq 2^n$.
Therefore, we need to prove that
$A < B \Leftrightarrow 0 < D < 2^n$

(1) $D = A + \overline{B} + 1$ — *Definition of D*
(2) $\overline{B} = 2^n - 1 - B$ — *Arithmetic way to find* $\overline{B}$

(1)+(2) $D = A - B + 2^n$
$\Rightarrow A < B \Leftrightarrow 0 < D < 2^n$

2.2.3 Signed *less than* and *greater or equal*

For these two signed comparisons (LT and GE), we need to consider the sign bits of the inputs ($\mathbf{a}_{31}$ and $\mathbf{b}_{31}$) and the sign of the difference ($\mathbf{diff}_{31}$).

For *less than* (LT):

- If **a** is negative and **b** is positive, **a** is less than **b**.
- If **a** and **b** have the same sign, we check if the difference is negative.

For *greater than or equal* (GE):

- If **a** is positive and **b** is negative, **a** is greater than **b**.
- If **a** and **b** have the same sign, we check if the difference is non-negative.

2.2.4 Summary of Comparison Operations

The comparator unit supports six comparison operations, each corresponding to a RISC-V branch instruction. The logical functions for these operations are summarized in Table 2.

Table 2: Comparison operations and their logical functions.

Operation	Opcode	Logical Function
$A = B$	000	$zero$
$A \neq B$	001	$\overline{zero}$
$A < B$ (signed)	100	$(A_{31} \wedge \overline{B_{31}}) \vee ((A_{31} \overline{\oplus} B_{31}) \wedge \overline{diff}_{31})$
$A \geq B$ (signed)	101	$(\overline{A_{31}} \wedge B_{31}) \vee ((A_{31} \overline{\oplus} B_{31}) \wedge \overline{diff}_{31})$
$A < B$ (unsigned)	110	$\overline{carry} \wedge \overline{zero}$
$A \geq B$ (unsigned)	111	$carry \vee zero$

These operations utilize the **zero** and **carry** signals from the subtraction result, as well as the sign bits of the inputs (A_{31} and B_{31}) and the sign of the difference (**diff**$_{31}$) for signed comparisons. The comparator efficiently implements all necessary comparisons for RISC-V branch instructions, supporting both signed and unsigned integer comparisons.

2.3 Logic Unit

The **Logic** unit performs $\oplus$ (XOR), $\vee$ (OR), and $\wedge$ (AND) bitwise operations. The 3-bit **op** signal selects the operation according to Table 3.

Table 3: Logic operations.

Operation	Opcode
$A \oplus B$	100
$A \vee B$	110
$A \wedge B$	111

The Logic unit supports the basic bitwise operations required by the RISC-V ISA. These operations are performed on a bit-by-bit basis across the entire 32-bit width of the input operands. The result of the selected operation is output as a 32-bit value.

2.4 Shift Unit

The **Shift** unit can shift operand **A** by **B** bits.

- Input **B** defines how many positions we should shift **A**. Only the 5 least significant bits of **B** are used, allowing shifts of 0 to 31 bits.
- Input **op** selects the operation according to Table 4.
- The **arithmetic** input (special bit) determines whether right shifts are logical or arithmetic.

Table 4: Shift operations.

Operation	Description	Opcode	Special Bit
$A \ll B$ (SLL)	Shift Left Logical	001	ϕ
$A \gg_l B$ (SRL)	Shift Right Logical	101	0
$A \gg_a B$ (SRA)	Shift Right Arithmetic	101	1

For all shift operations, **A** is shifted (moved) to the left or to the right by the number of positions defined by **B**. The bits that are shifted out are discarded. The behavior of each shift operation is as follows:

- **SLL** (Shift Left Logical): Zeros are shifted in from the right.
- **SRL** (Shift Right Logical): Zeros are shifted in from the left.
- **SRA** (Shift Right Arithmetic): The sign bit (most significant bit) is replicated and shifted in from the left, preserving the operand's sign.

It's important to note that SRL and SRA share the same opcode (101). The **arithmetic** input (special bit) distinguishes between these two operations:

- When the special bit is 0, the operation is SRL (logical right shift).
- When the special bit is 1, the operation is SRA (arithmetic right shift).

3 Exercise

You have to implement the described ALU for the RV32I architecture. Download the project template and open it in VSCode. The top-level module is defined in the `alu.v` file, which you should not modify.

Your task is to complete the implementation of the following Verilog files:

- `add_sub.v`: Implement the Add/Sub unit as described in Section 2.1.
- `comparator.v`: Implement the Comparator unit as described in Section 2.2.
- `logic_unit.v`: Implement the Logic unit as described in Section 2.3.
- `mux4x32.v`: Implement the 4-to-1 multiplexer as shown in Figure 2.
- `shift_unit.v`: Implement the Shift unit as described in Section 2.4.

Do not create new files or new Verilog modules. Follow the provided templates for each file and implement the required functionality within these existing modules.

Note: While there are testbench files provided in the `testbench` folder, completing these is optional and not part of the graded assignment. However, we encourage you to use and modify these testbenches to verify the correctness of your implementations.

3.1 The Logic Unit

- Open the `logic_unit.v` file.
- Complete the code of the **Logic** unit referring to its description in Subsection 2.3.

3.1.1 Testbench as Simulation Input Vector

For the simulation, we will use a Verilog testbench file with Verilator. This particular Verilog file contains an instantiation of the design unit that you want to simulate and generates the simulation input vector.

Project Structure: The project consists of three main folders:

- **dump:** Where simulation output files (like VCD files) will be saved
- **testbench:** Contains the testbench files for each module
- **verilog:** Contains the Verilog source files for each module

All commands should be run from the root directory of the project.

- The **testbench** folder contains a testbench template for the logic unit simulation. Open the `tb_logic_unit.v` file and observe the code. It contains a **logic_unit** module instance and an initial block for generating stimuli.
- Complete the initial block for the logic unit verification by providing the missing inputs (values of **op**) required to test the logic operations outlined in Table 3.
- Compile the project files using Verilator. Open a terminal and run:

```
verilator --binary --trace -Wno-fatal --top-module tb_logic_unit -o
↪  Vtb_logic_unit testbench/tb_logic_unit.v verilog/logic_unit.v
```

- Run the simulation using:

```
./obj_dir/Vtb_logic_unit
```

- This will generate a `tb_logic_unit.vcd` file in the current directory.
- Open the generated VCD file with GTKWave. If you have GTKWave already opened, you can drag and drop the file into the GTKWave window to achieve the same result.

```
gtkwave dump/tb_logic_unit.vcd
```

  Note: The `--top-module` flag specifies the top-level module for simulation. In this case, it's the testbench module `tb_logic_unit`. The top module is the highest-level module in your design hierarchy, which instantiates and connects all other modules.

- Add the signals to the wave view in GTKWave as shown in Figure 4.

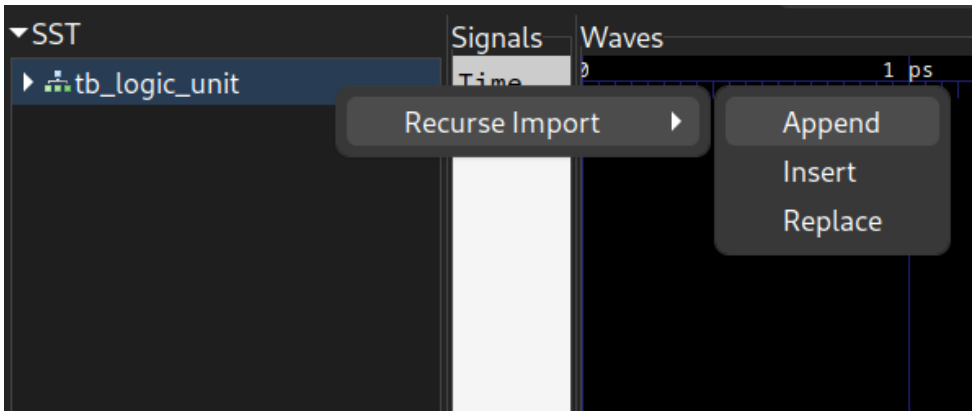

Figure 4: Adding signals to the GTKWave interface

- Zoom out to the desired time frame and observe the signals. You should have a similar output to the one shown in Figure 5 (Note: The image shows a completed testbench, your output will differ).

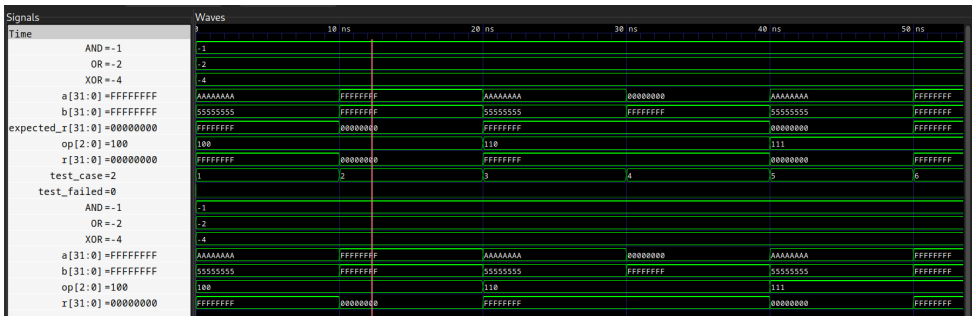

Figure 5: GTKWave interface showing Logic Unit signals

- If you need any additional help with GTKWave, refer to the GTKWave tutorial on the course's Moodle page.

For the full ALU simulation later, you'll use a similar process. The Verilator command will be:

```
verilator --binary --trace -Wno-fatal --top-module tb_alu -o Vtb_alu
↪  testbench/tb_alu.v verilog/*.v
```

3.1.2 Testbench for Verification

The testbench can also be used for automated verification. For the verification, we will use `$display` statements and conditional checks.

```
// Test XOR operation
a = 32'hAAAAAAAA; b = 32'h55555555; op = XOR;
#20; // wait for circuit to settle
if (r !== 32'hFFFFFFFF) begin
    $display( Error: XOR operation failed );
    $display( Expected: 32'hFFFFFFFF, Got: %h , r);
end else begin
    $display( XOR operation passed );
end
```

In this example, we verify whether the **logic_unit** output result is correct. The result of the XOR operation should be equal to `32'hFFFFFFFF`. If not, it will display an error message.

To complete the `tb_logic_unit.v` file, do the following:

- Add similar checks for OR and AND operations.
- Add a check for an undefined operation (should default to all zeros).
- Compile and run the simulation again to verify your logic unit.

3.2 The Add/Sub Unit, the Comparator, the Multiplexer, and the Shift Unit

- Referring to the description provided in section 2, complete the Verilog code of the **Add/Sub** unit (2.1), the **Comparator** (2.2), the **Multiplexer** (Figure 2), and the **Shift** (2.4) unit.
- Open the testbench in the `tb_alu.v` file. It is almost complete. You need to add the missing lines of code to test the comparison operations from Table 2.
- Simulate the ALU with this testbench using Verilator and view the results in GTKWave if needed.

We encourage you to write additional testbenches to verify the correctness of your other units as an additional exercise. Note that you can create a Makefile to automate the compilation and simulation process if you're comfortable with it.

4 Submission

Submit all Verilog files related to the exercises in Section 3 (`add_sub.v`, `comparator.v`, `logic_unit.v`, `mux4x32.v` and `shift_unit.v`).

Once you submit the files, you will receive a report describing the tests that were applied to your design and the results of those tests (success or failure). This is a preliminary submission so the result of the tests will not affect your grade and in case of failure, you will have some additional infos about what was the first test that failed of every testbench.