ALU

Learning Goal: Testbenches, arithmetic operations.

Requirements: Quartus II Web Edition and ModelSim.

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.

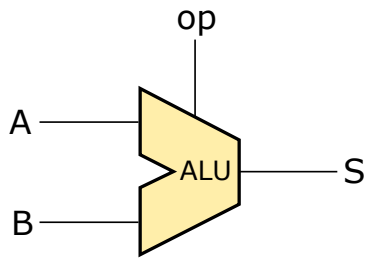

Figure 1: A simple ALU with two inputs, a control signal and an output.

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	$000\phi\phi\phi$
$A - B$		$001\phi\phi\phi$
$A \geq B$ (signed)	Comparison	011001
$A < B$ (signed)		011010
$A \neq B$		011011
$A = B$		011100
$A \geq B$ (unsigned)		011101
$A < B$ (unsigned)		011110
$A\ nor\ B$	Logical	$10\phi\phi00$
$A\ and\ B$		$10\phi\phi01$
$A\ or\ B$		$10\phi\phi10$
$A\ xor\ B$		$10\phi\phi11$
$A\ rol\ B$	Shift/Rotate (Optional)	$11\phi000$
$A\ ror\ B$		$11\phi001$
$A\ sll\ B$		$11\phi010$
$A\ srl\ B$		$11\phi011$
$A\ sra\ B$		$11\phi111$

ϕ = *don't care*

- The two most significant bits (i.e., $\mathbf{op}_{5..4}$) select the operation type (e.g., Add/Sub, Comparison, Logical).
- The $\mathbf{op}_3$ bit activates the **subtraction mode** of the **Add/Sub** unit. The **subtraction mode** is always activated for the comparison unit and ignored for the logical and shift unit.
- The $\mathbf{op}_{2..0}$ bits select a specific operation in a unit.

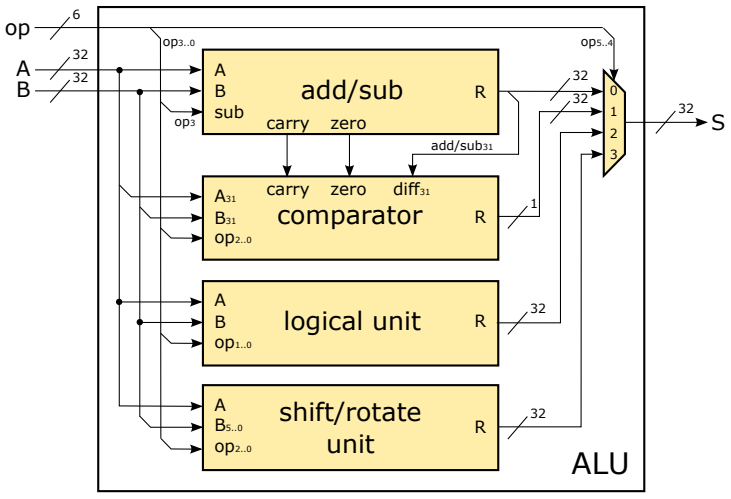

Figure 2: The internal architecture of the ALU.

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.

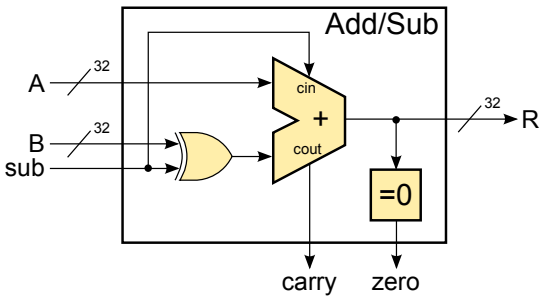

Figure 3: The internal architecture of the **Add/Sub** unit.

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. 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**.

$3 + 3 = 6$	$7 + (-1) = 6$	$5 - 4 = 1$	$(-5) - 0 = -5$
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 the comparisons *equal* and *not equal*, unsigned *greater or equal* and *less than*, and two's complement signed *greater or equal* and *less than*. It computes the comparison 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* and *not equal* comparisons result is directly driven by the **zero** input signal: if **A** equals **B**, then the subtraction result is zero.

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

The *greater or equal* and *less than* unsigned comparisons depend only on the **carry** signal. If **carry** is high, then **A** is greater or equal to **B**; otherwise, **A** is less than **B**. You can find a little 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 1, we have that $D \geq 2^n$, we need to prove that
$A \geq B \Leftrightarrow D \geq 2^n$

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

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

2.2.3 Two's Complement Signed *greater or equal* and *less than*

For these two signed comparisons we need a little more logic. We have $\mathbf{A}_{31}$, $\mathbf{B}_{31}$ and $\mathbf{diff}_{31}$, which are the most significant bit (i.e., the **sign**) of the ALU inputs **A** and **B**, and of the subtraction result, respectively. If **A** is positive (its sign bit is 0) and **B** is negative (its sign bit is 1), trivially we can say that **A**$\geq$**B**; if their signs are the same and the subtraction is positive, we also have **A**$\geq$**B**. In any other case, **A**$<$**B**. From this description, try to extract the logical function for each comparison operation. The solution is given in Subsection 2.2.4.

2.2.4 Summary

Table 2 gives a summary of the comparison operations with their respective logical functions and encoding. *Note: we suggest that you set the default operation to $A = B$ (i.e., for undefined opcodes 0 and 7).*

Table 2: Comparison operations.

Operation	Opcode	Logical Function
$A \geq B$ (signed)	001	$(\overline{A}_{31} \cdot B_{31}) + \overline{diff}_{31} \cdot (\overline{A}_{31} \oplus B_{31})$
$A < B$ (signed)	010	$(A_{31} \cdot \overline{B}_{31}) + diff_{31} \cdot (\overline{A}_{31} \oplus B_{31})$
$A \neq B$	011	$\overline{zero}$
$A = B$	100	$zero$
$A \geq B$ (unsigned)	101	$carry$
$A < B$ (unsigned)	110	$\overline{carry}$

2.3 Logic Unit

The **Logic** unit performs AND, OR, NOR and XOR logic bitwise operations. The 2-bit **op** signal selects the operation according to Table 3.

Table 3: Logic operations.

Operation	Opcode
$A\ nor\ B$	00
$A\ and\ B$	01
$A\ or\ B$	10
$A\ xor\ B$	11

2.4 Shift/Rotate Unit

The **Shift/Rotate** unit can shift or rotate 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.
- Input **op** selects the operation according to Table 4.

Table 4: Shift/Rotate operations.

Operation	Description	Opcode
$A\ rol\ B$	rotate left	000
$A\ ror\ B$	rotate right	001
$A\ sll\ B$	shift left logical	010
$A\ srl\ B$	shift right logical	011
$A\ sra\ B$	shift right arithmetic	111

For the *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. For *logical shifts* (i.e., *sll* and *srl*), zeros are shifted in; for *right arithmetic shifts* (*sra*) the sign bit is replicated to preserve the operand's sign. For the *rotation* operations (i.e., *rol* and *ror*), also called *circular shift*, the bits that are shifted out are reinjected at the other end of the word. Note that **A** *rotated left* by **B** is equivalent to **A** *rotated right* by (-**B**).

The **Shift/Rotate** unit contains several *barrel shifters* to perform the different operations. A barrel shifter is a sequence of multiplexers, where each stage of multiplexer can shift its input by a power of 2. Figure 4 illustrates an example of a barrel shifter.

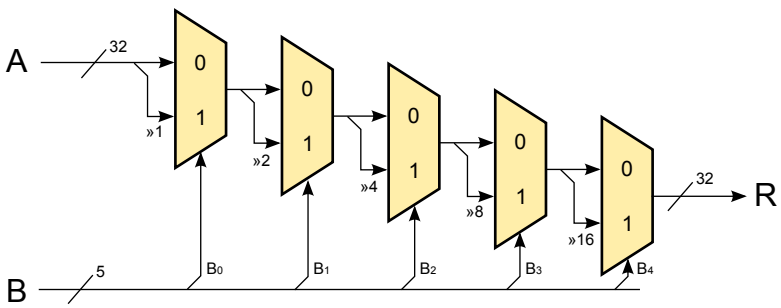

Figure 4: A 5-bit barrel shifter.

3 Exercise

You have to implement the described ALU. Download the project template and open it (quartus/ALU.qpf). The top level architecture is given in the `ALU.bdf` file.

3.1 The Logic Unit

- Open the `logic_unit.vhd` 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 VHDL testbench file. This particular VHDL file contains an instantiation of the design unit that you want to simulate and generates the simulation input vector.

- Open the modelsim project (modelsim/alu.mpf).
- The **testbench** folder contains a testbench example for the logic unit simulation. Open the `tb_logic_unit.vhd` file and observe the code. It contains an empty entity, but it includes a **logic_unit** unit instance. The given process generates some stimuli on the **logic_unit** inputs during a time interval defined by `wait for 20 ns`. At the end of the process, the **wait** statement stops the process to avoid an infinite loop execution.
- Complete the process for the logic unit verification by providing the missing inputs (values of **op**) required to test the logic operations outlined in Table 2.3. For now, ignore the comments in the code related to inserting the **ASSERT** statement—this will be explained in the following section.
- Compile the project files.
- Start the simulation of the logic unit **testbench** `tb_logic_unit.vhd` (NOT of the logic unit itself).
- Add the signals to the **wave**.
- The simulation input vector is already defined by the testbench, therefore, to start the simulation, you just have to type `run -all` in the ModelSim console. The `-all` option makes it run until there is no change in the stimuli anymore.

3.1.2 Testbench for Verification

The testbench can also be used for automated verification. For the verification, we will use the `ASSERT` statement which is reserved for testing purposes.

```
-- A NOR B
op <= "00";
wait for 20 ns; -- wait for circuit to settle

assert r(3 downto 0) = "0001"      -- Should be true
   report "Incorrect NOR Behavior" -- Message to display
   severity warning;               -- Message is a warning
```

The **ASSERT** statement is followed by a condition. If the condition is false, it will report a message to the ModelSim console. The **REPORT** option defines the message to be displayed in case of a false condition. The **SEVERITY** option defines the message type: it can be a `NOTE`, a `WARNING`, an `ERROR` or a `FAILURE`.

In this example, we verify whether the **logic_unit** output result is correct. The result of the operation `"1010" NOR "1100"` should be equal to `"0001"`. Otherwise, it will report a warning message. You may also write a generic condition formulation:

```
-- A NOR B
op <= "00";
wait for 20 ns; -- wait for circuit to settle

assert r = (a nor b)                -- Should be true
   report "Incorrect NOR Behavior" -- Message to display
   severity warning;                -- Message is a warning
```

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

- Insert an **ASSERT** statement in the testbench process after the **NOR**, as shown in the previous example.
- Compile and restart the simulation (console command: `restart -f`).
- Add other **ASSERT** statements to verify the remaining logical operations.
- Compile and restart the simulation.

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

- Referring to the description provided in section 2, complete the VHDL code of the **Add/Sub** unit (2.1), the **Comparator** (2.2) and the **Multiplexer** (Figure 2).
- Open the testbench in the `tb_ALU.vhd` 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 and ignore any error coming from the **Shift/Rotate** unit.

3.3 The Shift/Rotate Unit (Optional)

In this optional section, you will implement the **Shift/Rotate** unit.

- Complete the VHDL code of the **Shift/Rotate** unit. To help you, we provide two examples of implementation of a barrel shifter.
- Verify your design by running the `tb_ALU.vhd` testbench.

Next, we give two equivalent examples of VHDL implementations of a barrel shifter. The first example is done with sequential *if* statements and the second one with a loop.

These examples use *variables* to simplify the code. Remember that a *variable* does not behave like a *signal* and is only accessible from its process. The value of a signal is refreshed at the end of a process, while a variable is modified instantly inside the process, which allows one to progressively modify it through the process.

3.3.1 First example:

```
sh_left : process(a, b)
    variable v : std_logic_vector(31 downto 0);
begin
    -- The variable v will contain the intermediate value.
    -- For each bit of b, we check if we have to shift v.
```

```
    v := a;                          -- v initialization
    -- shift by 1
    if (b(0) = '1') then
        v := v(30 downto 0) & '0';
    end if;
    -- shift by 2
    if (b(1) = '1') then
        v := v(29 downto 0) & (1 downto 0 => '0');
    end if;
    -- shift by 4
    if (b(2) = '1') then
        v := v(27 downto 0) & (3 downto 0 => '0');
    end if;
    -- shift by 8
    if (b(3) = '1') then
        v := v(23 downto 0) & (7 downto 0 => '0');
    end if;
    -- shift by 16
    if (b(4) = '1') then
        v := v(15 downto 0) & (15 downto 0 => '0');
    end if;

    shift_left <= v;
end process;
```

3.3.2 Second example:

```
-- shift_left
sh_left : process(a, b)
    variable v : std_logic_vector(31 downto 0);
begin
    v := a;
    for i in 0 to 4 loop
        if (b(i) = '1') then
            v := v(31 - (2 ** i) downto 0) & ((2 ** i) - 1 downto 0 => '0');
        end if;
    end loop;
    shift_left <= v;
end process;
```

4 Submission

Submit all VHDL files related to the exercises in Section 3 (`ALU.vhd`, `add_sub.vhd`, `comparator.vhd`, `logic_unit.vhd`, `multiplexer.vhd`, `tb_ALU.vhd`, `tb_logic_unit.vhd` and `shift_unit.vhd`). You can use the `shift_unit.vhd` file provided in the lab template, in case you have not implemented it yourself. This will not affect your score.

At least one submission has to be made in order to receive a grade on the first project "Designing a Multicycle Processor". 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). Additionally, you will receive a score. However, this score will **NOT** be taken into account for the grade on the first project.