ALU

Learning Goal: Testbench, arithmetic operations

Requirements: Quartus II Web Edition and ModelSim.

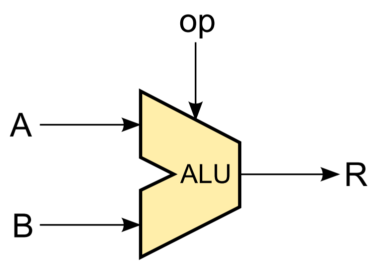
1 Introduction

In this lab you are going to implement a complete ALU and learn how to make testbenches and gate level simulations.

2 ALU description

An ALU (*Arithmetic Logic Unit*) is a combinatorial circuit performing arithmetic and logical operations. It's 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. The following figure is the common representation of an ALU.



For this lab you have to implement a 32-bit ALU with 4 internal units. The available operations and their corresponding encoding are listed in the following table.

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		$11\phi111$

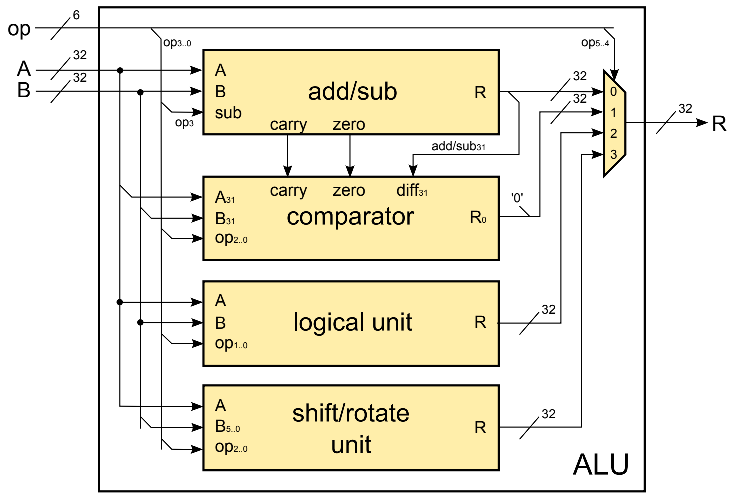
ϕ = *don't care*

- The 6-bit **op** control signal selects one of these operations respectively to this table. Notice that the 2 most significant bits (i.e., **op**$_{5..4}$) select the operation type (e.g., Add/Sub, Comparison, Logical).

- The **op**$_3$ bit is used to activate the **subtraction mode** of the **Add/Sub** unit. Notice that the **subtraction mode** is always activated for the comparisons and ignored for the logical and shift unit. We will see later that the comparator unit needs the result of the subtraction to perform a comparison.

- The **op**$_{2..0}$ bits select a specific operation in a unit.

The following figure shows the internal structure of the ALU.



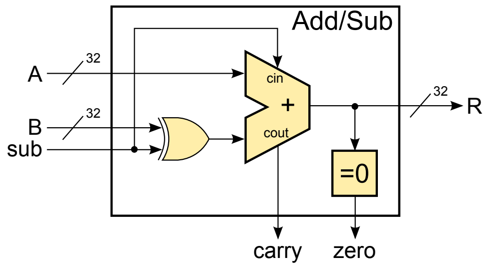
In the following subsections, each unit is described into more details. You can skip these for the moment, and start with the exercises of section 3.

2.1 Add/Sub

The **Add/Sub** unit performs 32-bit additions and subtractions.

- Input **sub** activates the **subtraction mode**.
- Output **carry** is the **carry out** of the internal adder.
- Output **zero** indicates that the result is equal to 0 when it is high.

You can see the internal architecture of the **Add/Sub** unit in the following figure.



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 **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, in **bold**.

$$\begin{array}{rrrr}
3+3=6 & 7+(-1)=6 & 5-4=1 & (-5)-0=-5 \\
 & & & \\
0011 & 0111 & 0101 & 1011 \\
0011 & 1111 & 1011 & 1111 \\
+\quad 0 & +\quad 0 & +\quad 1 & +\quad 1 \\
\hline
\mathbf{0}0110 & \mathbf{1}0110 & \mathbf{1}0001 & \mathbf{1}1011
\end{array}$$

2.2 Comparator

The **Comparator** unit performs *equal*, *not equal*, and signed or unsigned *greater or equal* and *less than* comparisons. It needs the result of the subtraction coming from the **Add/Sub** unit to compute the comparison. This is why the subtraction mode is always set for comparison operations.

- Input **op** selects the type of comparison.
- Output **R** is the result of the comparison (0=false, 1=true).
- Inputs **zero**, **carry**, **A**$_{31}$, **B**$_{31}$ and **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**, subtraction result will be 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 Signed *greater or equal* and *less than*

For these two signed comparisons we need a little more logic. We have **A**$_{31}$, **B**$_{31}$ and **diff**$_{31}$ 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 and **B** is negative, trivially we can say that **A**$\geq$**B**; if their signs are the same and the subtraction is positive, we have also that **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 in the next subsection.

2.2.4 Summary

The following table gives a summary of the comparison operations with their respective logical functions and encoding.

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	$\overline{carry}$

Note: we suggest you to set the default operation to $A = B$ (i.e., for undefined opcode 0 and 7).

2.3 Logic unit

This unit performs AND, OR, NOR and XOR logical bitwise operations. The **op** 2-bit signal selects the operation according to the following table.

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

2.4 Shift/Rotate unit

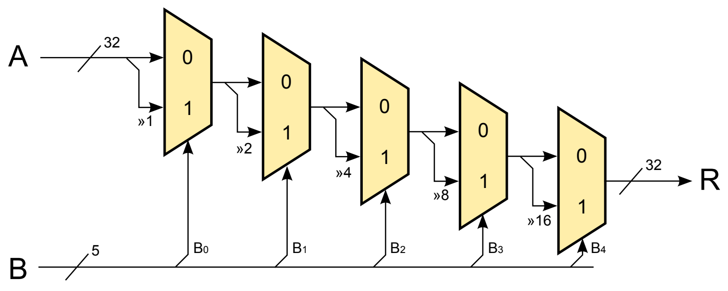
The **Shift/Rotate** unit can shift or rotate operand **A** by **B** bits.

- Input **B** defines by how many position we should shift **A**. Only the 5 least significant bits of **B** are used.
- Input **op** selects the operation according to the following table.

Operation	Opcode
$A\ rol\ B$	000
$A\ ror\ B$	001
$A\ sll\ B$	010
$A\ srl\ B$	011
$A\ sra\ B$	111

For the *shift* operations, **A** is shifted (moved), to the left or to the right by the number of position 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. The following figure illustrates an example of a barrel shifter.



3 Exercise

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

- Open the `logic_unit.vhd` file.
- Complete the code of the logic unit referring to its description, in subsection 2.3.

3.1 Testbench as simulation input vector

For the simulation, we will use a VHDL testbench file. This particular VHDL file will contain an instantiation of the design unit that you want to simulate and will generate the simulation input vector.

- Open the modelsim project (alu/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. You can notice that its entity is empty and that it includes a **logic unit** instance. The 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.

- Compile the project files.

- Start the simulation of the logic unit **testbench** `tb_logic_unit.vhd` (and NOT 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 only have to type directly in the ModelSim console `run -all`. The `-all` option makes it run until there is no change in the stimuli anymore.

3.2 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 ca 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"`. If it is not, it will reports a warning message.

- Insert an **`ASSERT`** statement in the testbench process after the **`NOR`**, as shown in the previous example.

- Compile and restart the simulation (`restart -f`).

- Add other **`ASSERT`** statements to verify the remaining logical operations.

- Compile and restart the simulation.

3.3 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, **Comparator** and **Multiplexer**.

- A complete testbench is provided in the `tb_ALU.vhd` file. This testbench will test all the available units of the system. Simulate the ALU with this testbench and ignore any error coming from the **Shift/Rotate** unit.

3.4 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 you with two examples of implementation of a barrel shifter.
- Verify your design by running the `tb_ALU.vhd` testbench.

Here, we give you 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.

3.4.1 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;
```

3.4.2 Second example:

```
sh_left_loop: process (a, b)
  variable v : std_logic_vector(31 downto 0);
begin
  v := a; -- v initialization
  for i in 0 to 4 loop
    if (b(i)='1') then
      -- x**y = x power of y
      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 (not including the testing subsections) and section 3.3 (`ALU.vhd`, `add_sub.vhd`, `comparator.vhd`, `logic_unit.vhd`, `multiplexer.vhd` and `shift_unit.vhd`). You can use `shift_unit.vhd` provided in the lab template in case you haven't implemented it. This unit will not be graded in the evaluation.