EDA322 Digital Design Lab

LAB2

Designed by: Angelos Arelakis; Stavros Tzilis, Anurag Negi, Ioannis Sourdis

The goal of this lab is to design an Arithmetic Logic Unit (ALU) for an8-bit processor, using the tools you get familiar with in the previous lab. Before starting the lab, please do the preparation as described below.

For this and all other labs it is important that you do not share code with other groups. Your code will be checked for plagiarism at the end of the course.

Preparation

Preparing for the second lab requires to:

1. Complete lab1.
2. Study introduction and paragraph "Arithmetic and Logic Unit (ALU)" at Section 1, in the provided processor's specification document (processor.pdf).
3. Study the lecture material of up to the previous study week.
4. Prepare in a document the following:
    a. Write the VHDL implementation of a 4-to-1 mux (multiplexer) using 2-to-1 muxes as components. Show the component definition and instantiation (port map).
    b. Referring to the code of file "sample.vhdl" (download it from pingpong), answer the following question: At time t=0, input "in1" takes the value "00" and keeps it for a long time. Given the signal assignments of lines 19 and 20, what are the values of signal 'a' and output "out1" during this time (as long as "in1" remains "00")?
5. Read through the Lab PM before starting to do each task.

Introduction

The ALU is one of the most important components in a processor, as it does all the necessary calculations between the operands. It normally does arithmetic operations such as *addition*, *multiplication*, *division* as well as logic operation such as *and*, *or*, etc. However, the Chalmers Accumulator (ChAcc) processor that is going to make use of the ALU, has a reduced set of

instructions, thus the ALU operations are limited to only *add*, *sub*, *and*, *not* and *cmp*. Furthermore, our ALU supports arithmetic operations only between unsigned numbers.

The block diagram of the ALU is depicted in Figure 1. The interface was already presented in processor.pdf. There are four sub-components:

- "Adder": It performs addition/subtraction between the two ALU input data operands
- "And": It performs an *AND* operation between the two ALU input data operands
- "Not": It performs a *Not* operation of ALU_inA only
- "Comparator (cmp)": It compares the two ALU input data operands. Compare is always active irrespective of operation.

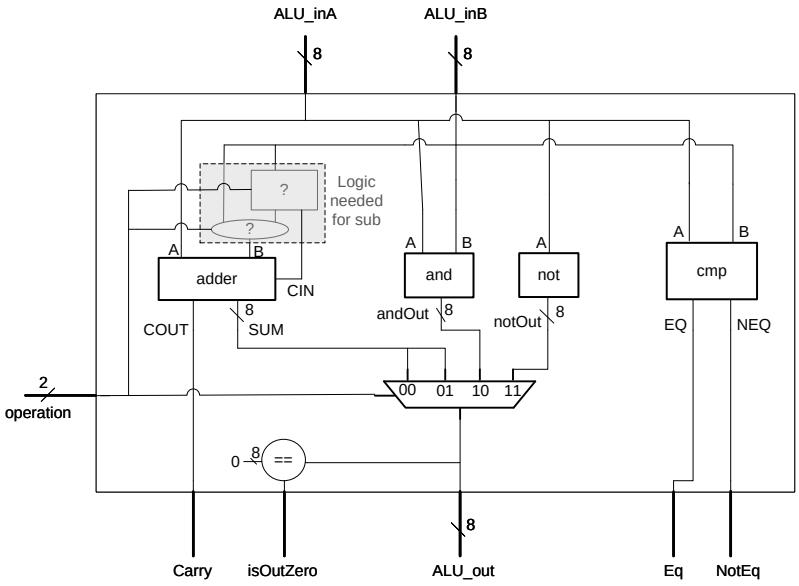

Figure 1: Block diagram of the ALU

This lab **requires** you to do the following tasks:

1. Implement a *ripple carry adder* (RCA) using smaller components such full adders (FA) and verify its correct operation.
2. Implement the comparison operation *cmp*.
3. Task 3 includes the following sub-tasks:
   a. Implement the *AND* and *NOT* operations.
   b. Add subtraction functionality to the implemented adder.
   c. Integrate the implemented sub-components(adder, and, not, cmp) into one ALU unit.
   d. Verify the correct operation of the ALU using the provided test file.

The lab also contains the following **optional** tasks:

- Implement a faster adder, i.e. a carry look-ahead adder (CLA), using smaller components such as *generate* and *propagate* functions and verify its correct operation.

- Integrate the CLA with the rest of components in a second ALU unit version.

Start by opening ModelSim (or QuestaSim) and create a new project with name, e.g., Lab2.

Arithmetic operations (Adder) – Task 1

In this part, you are going to implement the unit that performs arithmetic operations (*add, sub)*. As you have seen in the lecture, there are many different types of adders. For instance, the *ripple carry adder* is simple enough to design. On the other hand, the *carry look-ahead adder* is relatively more complicated but can perform much better. In this task, you are going to implement in VHDL a ripple carry adder (RCA) only, using the **dataflow** and the **structural** design styles. You have already seen in the lecture how to design an adder using the behavioral design style.

The ripple-carry adder, which is going to be implemented, is designed as a chain of Full Adders (FA),as is depicted in Figure 2. Each FA has three **inputs**(c_i, *a, b*) and two **outputs**(c_{i+1}, *s)*, as presented in Figure 2. The third input (c_i) is the carry-in and is generated by a previous full adder. Draw the **data-flow** diagram of a FA, based on the truth table of Table 1. You first need to write down the Boolean expressions that describe the functionality of the FA. Then write the VHDL for the FA in a file with name FA.vhdl.

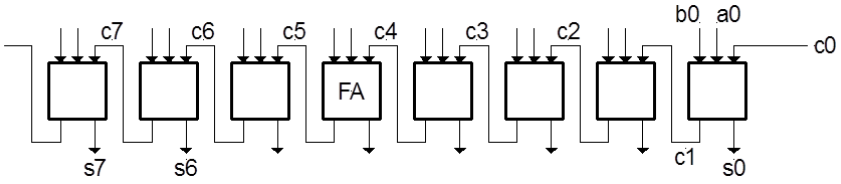

Figure 2: An 8-bit Ripple Carry Adder using FAs

Table 1: Truth table for a FA

c_i	a_i	b_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Alternatively, the FA can be decomposed into even smaller units that are called half adders (HAs). The HA has only two inputs, as is depicted in Figure 3, while its outputs are the sum (s)

and the carry (c).A number of HAs plus logic is needed to build a FA. How is the FA built using the alternative design option based on the HA as components? Draw the block diagram.

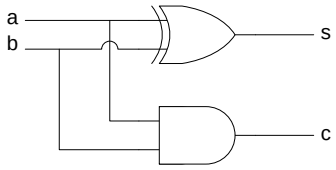

Figure 3: HA adder

A ripple carry adder (RCA) is simply a chain of connected full adders where the carry out of a FA at bit position, say *i,* is the carry in (cin) of the full adder at the bit position *i+1*. Write now the description of the RCA in VHDL using the FAs as components. You must make use of *port maps*(**structural** design style) to make the connections between the FAs. The entity of the ripple carry adder must have the name RCA while the entity's inputs/outputs are given in Table 2.Verify the correctness of your ripple carry adder using ModelSim. Write a do file, where you check the result for some inputs.

Table 2: RCA entity

Name	input/output	Width (bits)
A	Input	8
B	Input	8
CIN	In§put	1
SUM	Output	8
COUT	Output	1

Comparison operation (*cmp*) – Task 2

The *cmp* compares two operands and determines whether they are equal or not, asserting the respective flag:1) Equal (EQ) or 2)Not Equal (NEQ). These flags may be checked by a subsequent instruction, which is a branch instruction (JEQ, JNE) as you will see in the next labs.

There are (at least) two alternatives to implement the above *cmp* operation, using **dataflow** or **structural** design style. Which are they? Select one and implement it. You should not use a **behavioral** design style, e.g., "if (A=B) then EQ='1'".
Implement the comparator *cmp* in a new file and name the entity "cmp". The **inputs** of *cmp* are the two ALU data input operands while the outputs are two 1-bit signals: EQ and NEQ, as depicted in Figure 1.

Finalize the ALU design – Task 3

The goal of the final task is to finalize the design of some components and connect all the different modules into one ALU unit. First, create a new file that is called alu_wRCA.vhdl. Create the entity for the ALU by defining the inputs/outputs and their width, as they are shown in Table 3.This is the top level entity of the ALU or in other words the entity that contains all the functional units of the ALU. See processor.pdf for more details about the ALU inputs/outputs.

Table 3: Input and output signals of the ALU top entity

Name	input/output	Width (bits)
ALU_inA	Input	8
ALU_inB	Input	8
Operation	Input	2
ALU_out	output	8
Carry	output	1
NotEq	output	1
Eq	output	1
isOutZero	output	1

Follow the steps, to complete this task:

1. Write in **dataflow** VHDL the implementation for the rest of two operations *AND* and *NOT* inside the file alu_wRCA.vhdl. The *AND* is performed between *ALU_inA* and *ALU_inB*, while *NOT* has only one input (*ALU_inA*), according to Figure 1.
2. The current implementation of the adder supports *addition* only. Extra logic is needed to support *subtraction*, as also depicted in Figure 1 in the grey box. Derive the functionality of this box and write **dataflow** and/or **structural** VHDL code in alu_wRCA.vhdl. Don't modify the design of your RCA. Use only one instance of the RCA as component to support both addition and subtraction. Hint: Based on 2's complement representation, subtraction is performed as an addition but we have to modify some of the inputs. What modification is needed? How do we select the correct input set for the RCA, provided that we know whether we are going to perform addition or subtraction? The latter is revealed by the *Operation* input (see Table 4)?
3. Write in VHDL, inside the file alu_wRCA.vhdl, the implementation for the multiplexor (mux) 4-to-1 using **dataflow** design style, as you learned in the lecture. The ALU input *operation* helps to determine the mux's output, based on Table 4.
4. Integrate all the different units that you have implemented so far into one ALU using **structural** VHDL (using components), inside file alu_wRCA.vhdl. Use the block diagram of the ALU in Figure 1 to observe how the implemented components are connected.

Table 4: Selected operation based on *operation*

Operation	Operation
00	Add
01	Sub
10	AND
11	NOT

After completing the implementation above, verify the correct functionality of your ALU running the provided testbench (alu_testbench.vhdl) for **2010ns**. If it fails to pass the testbench, debug your design. In order to do this, use the waveform to find where the simulation has stopped. Check the value of the various signals to see if they take the value you expected based on the inputs. You can also add extra signals like intermediate signals from the test top level design component or the various sub-components.

Design a Carry Lookahead Adder – Optional

The ripple carry adder is simple enough to design it but suffers from long delays due to the carry propagation. In the previous 8-bit RCA, the delay (critical path) from the carry in $c_0$tothe carry out is 17 gates. Why is this number correct? Generalize for the case ofan n-bit adder. Obviously, if the adder is 32 bits or 64 bits the delay of the critical path is linearly increased.

One improvement that may have a significant effect in performance is to quickly evaluate if the carry in from a previous stage has a value 0 or 1 [1]. Using the truth table of Table 1, we can easily derive the Boolean equation: $c_{i+1} = x_i y_i + (x_i + y_i)c_i$, which can be re-written as: $c_{i+1} = g_i + p_i c_i$, where $g_i = x_i y_i$ and $p_i = x_i + y_i$. In this stage *i,* the *generate* function *g* generates a carry out if both x and y are 1 and no matter whether there is a carry in. On the other hand, the carry in will be propagated through the function *p* (*propagate* function) if at least one of the x or y is 1. See the book or the lecture notes for more information about the working of the carry look-ahead adder. Using the above equations, we can quickly derive the formula for the carry out of the 1-bit, 2-bit and 3-bit CLA respectively:

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

Surprisingly the critical path for c3, c2 and c1 is the same (3 gates) assuming gates with fan-in of 4. Assuming gates with this fan-in, what is the critical path of the respective 8-bit CLA in gate levels? Explain your answer. Generalize the case for an n-bit CLA.

Then write the VHDL to describe the hardware for this 8-bit carry look-ahead adder. You first need to derive the Boolean expressions. Moreover, the CLA entity (names of inputs/outputs) must be the same as the RCA entity (except for the entity name). Verify the correctness of your design using ModelSim as you did for the RCA.

Copy the previous alu_wRCA.vhdl into a new file alu_wCLA.vhdl and replace the RCA component with the CLA one. Verify the correct operation of the ALU with the new adder.

Demonstrations

Tasks to be done for successfully completing this lab:
1. Write a VHDL module which implements the ripple carry adder (RCA). Add it to your ModelSim project and simulate using a "do" file.
2. Write a VHDL module that implements the *cmp* operation. Add it to your ModelSim project and simulate using a "do" file.
3. Write a VHDL module that implements the ALU of 5 operations (*add*, *sub*, *and*, *not*, *cmp*) by connecting the RCA adder and the rest of modules into one ALU unit. Simulate using ModelSim and by running the provided test file for 2010ns. Show the results to your instructor.

Evaluation:
The instructor will check for the following:

Task#	Coding style	Simulation
1	X	
2	X	
3	X	X

Make sure that when you are done with the lab, you have demonstrated all checked aspects of each task. This is necessary for successful completion of the lab.

Lab report:
In the section "ALU design" of the final lab report, describe briefly what you did in this lab and what you have learnt. In addition, discuss your findings and observations during this lab. Summarize your answers to the questions in the lab PM and present the block diagrams that you have to draw. Remember to always explain your design choices and mention any assumptions. Finally, make use of figures and tables.

Learning outcome:
After completing this lab, you should be able to:

- Implement simple components using **dataflow** and **structural** VHDL.
- Implement a ripple carry adder using smaller sub-modules, like FA.

- Verify and debug a combinatorial circuit in VHDL using a provided testbench.

Hints and Tips:

For….. Generate

A very useful concurrent VHDL statement for instantiating several copies of a component is *"for … generate"*. Try to think how you can use it to design the ripple-carry adder with FA components.

Assigning a value to a vector

In VHDL it is possible to assign a value to each bit of a vector. Also, some other forms of value assignments are available for convenience (e.g. using syntaxes like *others*, *downto*or *upto*)
For example:

```
Q <= "00000001"; → Q <= (0 => '1', others => '0');
Q <= "10000010"; → Q <= (7|1 => '1', others => '0');
Q <= "00011110"; → Q <= (4 downto 1 =>'1', others=>'0');
Q <= "00000000"; → Q <= (others=>'0');
```

Component Instantiation

In VHDL-93, an entity-architecture may be directly instantiated inside another entity-architecture without the need of declaration. This is more compact way especially for cases that several components are needed. In this case all the files should be in a same folder. The syntax in this method is as follows:

```
U1: entity work.nameOfComponent(nameOfArch)
Generic map(
…..)
Port map (
……
);
```

Where U1 is the label, *nameOfComponent* is the entity and *nameOfArch* is the architecture of the component. Putting nameOfArch is optional.

or_reduce and and_reduce

Depending on the way you decide to implement your design, you might find it useful to use *or_reduce* and *and_reduce*which are basically employed to *or/and* all the bits of one vector. This is possible in VHDL-2008insideieee.numeric_std library for signed and unsigned vectors. You can change the VHDL version in the properties of each .vhd file you have.

Example:

```
allBitsAnded <= and mySignal;
allBitsOred  <= or mySignal;
```

References

[1]: Brown S. and Vranesic Z., "*Fundamentals of digital logic with VHDL design*", Second Edition, ISBN 007-124482-4.