Digital Design Lab (EDA322)

Lab 1

Examiner: Prof. Ioannis Sourdis
TAs: Ahsen Ejaz, Ghaith Abou Dan, Fredrik Jansson, Konstantinos Sotiropoulos,
Magnus Östgren, Neethu Bal Mallya, Panagiotis Strikos

Last Edited: 2024-01-22
Deadline: Your respective lab session during Study Week 3

1 Introduction

The goal of this lab is to design an Arithmetic Logic Unit (ALU) for an 8-bit processor using the tools introduced in the introductory lab. Before starting the lab, please prepare as described below.

1.1 Preparation

1. Complete the lab tutorial and corresponding coding tutorials.
2. Study ***Section 1 (Introduction)*** and ***Section 3.4 (Arithmetic and Logic Unit (ALU))*** in the provided processor specification document (*processor.pdf*). This lab manual references Figures and Tables in *processor.pdf* document. So keep the document handy.
3. Study the lecture material up to the previous study week.
4. Read through this lab manual and *lab_ guidelines.pdf* before starting with the tasks.

1.2 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 full adder.
- Verify and debug a combinational circuit in VHDL using a provided test file.

2 Tasks

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

1. Implement a *ripple carry adder* (RCA) using smaller components such as *full adders* (FA).
2. Implement the comparison operation, *cmp*.
3. Implement the rotate left operation *rol*.
4. This task includes the following sub-tasks:

   (a) Implement the AND operation.
   (b) Add subtraction functionality to the adder implemented in Task 1.
   (c) Integrate the implemented sub-components (Adder, AND, ROL, CMP) into one ALU unit.
   (d) Verify the correct operation of the ALU using the provided testbench.

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 the components in a second ALU unit version.

2.1 Task 1: Ripple Carry Adder (RCA)

In this task, you will implement the unit that performs arithmetic operations (ADD, SUB). There are many different types of adders. For instance, the *ripple carry adder* (RCA) is simple to design. On the other hand, the *carry look-ahead adder* (CLA) is relatively more complicated but can perform much better. In this task, you will implement only RCA using the data flow and the structural design styles in VHDL.

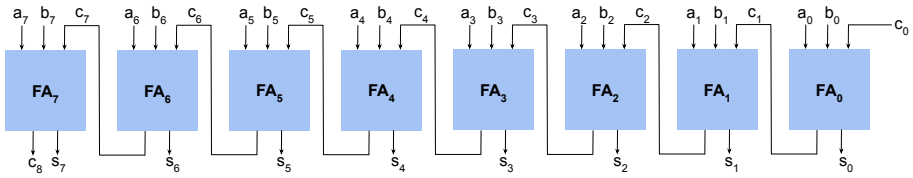

Figure 1: An 8-bit Ripple Carry Adder (RCA) using FAs

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

Table 1: Truth Table for a Full Adder (FA)

RCA is designed as a chain of Full Adders (FA), as depicted in Figure 1. Each FA has three inputs(a_i, b_i, c_i) and two outputs(c_{i+1}, s_i). RCA is simply a chain of connected FAs where the carry in (c_i) of the full adder at the bit position $i+1$ is the carry out of a FA at bit position i.

1. Based on the truth table given in Table 1, write down the Boolean expressions that describe the functionality of the FA and then complete the VHDL for the FA in `fa.vhd`.
2. Complete the description of the RCA in `rca.vhd` using FAs as components. **You must use port maps (structural design style) to connect the FAs.**
3. Verify the correctness of your ripple carry adder using ModelSim. Write a do file (`rca.do`) where you check the result for some inputs.

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

Tips: **Component Instantiation**: In VHDL-93, an entity architecture may be directly instantiated inside another entity architecture without the need for declaration. This is a more compact way, especially for cases where several components are needed. In this case, all the files should be in the 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.

2.2 Task 2 - Comparison operation (*cmp*)

The *cmp* compares the two input operands and determines whether they are equal or not by asserting the *Equal* flag, `E`. As you will see in the following labs, this flag may be checked by a subsequent jump instruction like `JE` (Jump Equal) or `JNE` (Jump Not Equal).

1. Implement the comparator in `cmp.vhd`. **You should not use a behavioral design style, i.e.,** `if (A=B) then E <= '1'`
2. Write a do file (`cmp.do`) where you check the result for some inputs.

Tips: ***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 with the `ieee.std_logic_misc.all` library. For example:

```
allBitsAnded <= AND_REDUCE(mySignal);
allBitsOred <= OR_REDUCE(mySignal);
```

Similar functions are also available for `XOR`, `XNOR`, `NOR` and `NAND`.

2.3 Task 3 - Rotate Left Operation (`ROL`)

The Rotate Left (ROL) operation, is a bitwise rotation where each bit of a binary value is shifted to the left by a single position as presented in Figure 2.

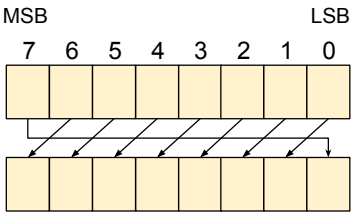

Figure 2: Circular left rotation of one bit (ROL)

The one-bit ROL can be implemented as a combinational circuit that takes an input vector and produces the rotated output. The basic structure of a ROL circuit involves shifting the bits one position to the left and wrapping the leftmost bit to the rightmost position.

For the implementation, you should not use the `rol` keyword.

2.4 Task 4 - Finalize the ALU design

This task aims to finalize the design of ALU. See *Table 7 (processor.pdf) and Table 8 (processor.pdf)* for more details about the ALU inputs/outputs. The entity for the ALU is given in `alu_wRCA.vhd`. This is the top-level entity of the ALU or, in other words, the entity that contains all the functional units of the ALU. You must complete the five operations (ADD, SUB, AND, ROL, CMP) by connecting the RCA adder and the rest of the modules into one ALU unit. **Inside `alu_wRCA.vhd`, write VHDL code to satisfy the following sub-tasks:**

1. The current implementation of the adder supports only addition. Extra logic is needed to support subtraction, as also depicted in *Figure 3 (processor.pdf)* in the grey box. Derive the functionality of this box and write dataflow and/or structural VHDL code. **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. Use `alu_op` to modify the input (`alu_inB`) for RCA as necessary.
2. Implement the functionality for Logical AND operation as given in *Table 8 (processor.pdf)*.

3. Implement a 4-to-1 multiplexer (mux) using dataflow design style, as you learned in the lecture. The ALU input, `alu_op`, helps to determine the mux's output, based on *Table 8 (processor.pdf)*.
4. Integrate all the different units that you have implemented so far into one ALU using structural VHDL (using components). Use the block diagram of the ALU in *Figure 3 (processor.pdf)* to observe how the implemented components are connected.

After completing the implementation above, verify the correct functionality of your ALU running the provided testbench (`alu_testbench.vhd`) using *run -all*.

If not already set, change the VHDL version of the testbench to 2008, by right-clicking on the file, selecting 'Properties', and on the 'VHDL' tab, selecting 'Use 1076-2008'.

Make sure you have all the testvector files (`*.tv`) in the same project directory. The files are available in Canvas (*Files* > *Labs* > *Lab 1* > *lab1_files*). If it fails to pass the testbench, debug your design using the waveform. Check the value of the various signals to see if they take the value you expected based on the inputs. You can add extra signals, like intermediate signals from the test top-level design component or the various sub-components.

Tips: **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[1]:

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');

2.5 Optional Task - Design a Carry Lookahead Adder

The ripple carry adder is simple enough to design but suffers long delays due to the carry propagation. In the previous 8-bit RCA, the delay (critical path) from the carry-in c_0 to the carry-out is 17 gates. Generalize for the case of an n-bit adder. If the adder is 32 bits or 64 bits, the critical path delay is linearly increased.

One improvement that may significantly affect performance is to quickly evaluate if the carry-in from a previous stage has a value of 0 or 1 [1]. Using the truth table of Table 1, we can easily derive the Boolean equation: $c_{i+1} = a_i b_i + (a_i + b_i)c_i$, which can be re-written as: $c_{i+1} = g_i + p_i c_i$, where $g_i = a_i b_i$ and $p_i = a_i + b_i$. In this stage i, the generate function g generates a carry-out if both a and b are 1, 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 a or b 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:

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

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.vhd` into a new file `alu_wCLA.vhd` and replace the RCA component with the CLA one. Verify the correct operation of the ALU with the new adder.

3 Demonstration and Evaluation

The lab will be evaluated according to the checked aspects in the table below. To demonstrate your successful completion of Lab 1, keep all the essential files and simulation results ready to be presented to a TA.

[1] All of these examples assume that Q is a `std_logic_vector(7 downto 0)`, the bit order is important.

Task#	Files	Coding Style	Simulation
1	`rca.vhd`, rca.do	✓	✓(Using rca.do)
2	`cmp.vhd`, cmp.do	✓	✓(Using cmp.do)
3	`rol.vhd`, rol.do	✓	✓(Using rol.do)
4	`alu_wRCA.vhd`	✓	✓(Using `alu_testbench.vhd`)

– **The demo must be completed during your registered lab session**. Should you require an exception outside your registered lab session, discuss it with the TAs.
– **Note that there is no code upload required for this lab.**

References

[1] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw-Hill, Inc., USA, 2nd edition, 2000.