LAB 5 – Implementing an ALU

Goals

- Design a practical ALU
- Learn how to extract performance numbers (area and speed)

To Do

- Draw a block level diagram of the MIPS 32-bit ALU, based on the description in the textbook.
- Implement the ALU using Verilog.
- Synthesize the ALU and extract performance numbers.
- Follow the instructions. Paragraphs that have a gray background like the current paragraph denote descriptions that require you to do something.
- To complete the lab you have to show your work to an assistant during any lab session, there is nothing to hand in.

Introduction

In the first few exercises we were dealing with fairly small circuits. In this exercise we tackle something more formidable, which is the heart of a computer – the arithmetic logic unit (ALU). We implement an ALU that is similar to the one described in section 5.2.4 of your textbook. This ALU is part of the small microcontroller we will build in the later exercises.

This exercise is split over two labs: 5 and 6. In this lab (5), we write HDL description of the ALU, and in the second lab (6) we will verify that it works correctly using a testbench.

Until now, we have neglected to investigate performance-related numbers such as area and delay of the circuit. We were simply happy as long as our circuit worked. To be fair, the circuits we designed were very small, and did not really warrant much investigation.

In this exercise, we build a decently-sized block, so we are also interested in how large the circuit is and how fast it will be able to perform the given operations. We will also try to see if our coding style has an effect on these performance numbers.

The ALU

We will design an ALU that can perform a subset of the ALU operations of a full MIPS ALU. You can refer to Appendix B of your textbook to see the full set of operations that MIPS can support. In this exercise, we develop an ALU that takes two 32-inputs A and B and will be able to execute the following seven instructions:

add, sub, slt, and, or, xor, nor

The ALU generates a 32-bit output that we call ‘Result’ and an additional 1-bit flag ‘Zero’ that will be set to ‘logic-1’ if all the bits of ‘Result’ are 0. The different operations will be selected by a 4-bit control signal called ‘AluOp’ according to the following table.

AluOp			
0000	add	A + B	Addition
0010	sub	A - B	Subtraction
0100	and	A and B	Logical and
0101	or	A or B	Logical or
0110	xor	A xor B	Exclusive or
0111	nor	A nor B	Logical nor
1010	slt	(A - B)[31]	Set less than
Others	n.a.	Don't care	

Table 1. Summary of the ALU control

(Note: slt should be 32-bit 1 if A is less than B otherwise 0; the logical operations are meant bitwise)

Just to give an example when the 'AluOp' input is 0101, the function

Result = A or B;

should be calculated. It is easy to see that there are many values of 'AluOp' for which no operation is defined. It is not very important what the circuit does when 'AluOp' has these values, since the 'Result' will simply be ignored in these cases. You can use this to your advantage to simplify the circuit.

Right now, the described operations may look random, but once we learn more about the MIPS instruction set architecture, these choices will make more sense.

Block diagram

The first order of business should be drawing a block diagram, like the one seen in Figure 5.15 of your textbook. This exercise is based on a (more or less) real example; there will not be a clear textbook 'best' solution for the circuit.

The following is one approach to analyze what is needed and come up with a block diagram. You are free to follow this example or come up with your own ideas. It is just important that you think about how the circuit should be implemented.

Let us first examine the different commands. You should see that we have two types of instructions. The three instructions **add**, **sub**, **slt** require arithmetic operations, whereas the four remaining **and**, **or**, **xor**, **nor** are logical operations. So we could perhaps see that we have two separate groups of operations. Now let us look at Table 1 and determine for which values of AluOp we perform an operation from which group. It should be pretty clear that when AluOp(2) is logic-0 we have an arithmetic operation and when AluOp(2) is logic-1 we select a logic operation. This means that the output of either group can be selected by a 2-input multiplexer that is controlled by AluOp(2). Figure 1 shows this distribution.

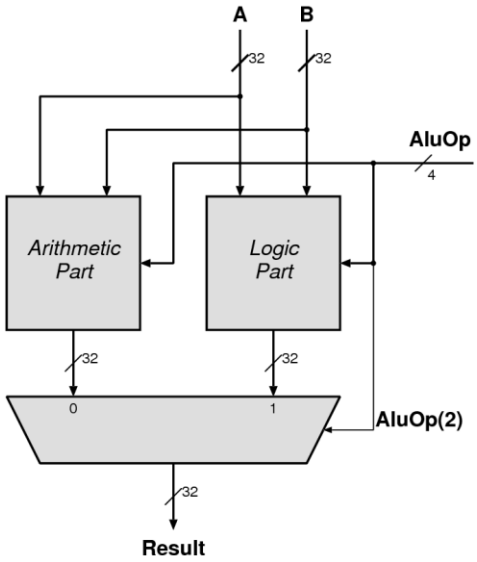

Figure 1. A possible division for the ALU

Now we can take a look at the two groups individually. For the logic group AluOp(1:0) selects one of the 4 simple logic operations. In the next group we realize that we have an addition (**add**) or a subtraction (**sub**, **slt**). We again could observe that AluOp(1) is logic-0 for additions and logic-1 for subtractions. This could allow us to build a structure like the one in Figure 5.15 of your textbook to design an adder-subtracter (controlled with AluOp(1) instead of F(2)). Figure 2 shows this arrangement.

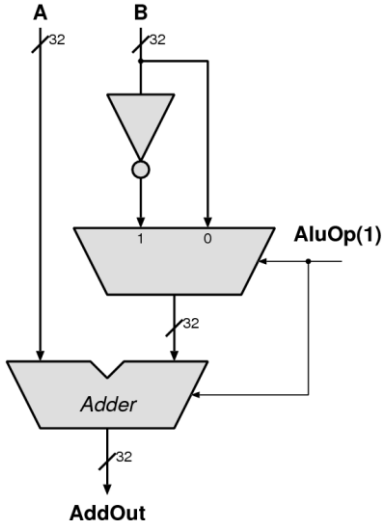

Figure 2. Possible organization for the adder subtracter in ALU

There is one more thing left, depending on the AluOp(3) we could select whether we take only the most significant bit (logic-1, **slt** instruction), or we take the output as it is. This part can be seen in Figure 3.

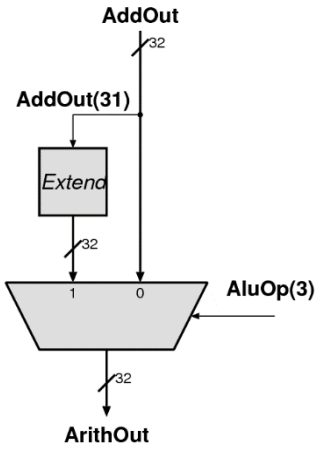

Figure 3. A possible organization to implement slt

Finally, we also have to add a small circuit that will generate the **zero** output when the **result** equals to all zeroes.

This method, of breaking a larger block successively in smaller pieces is called Divide and Conquer and is one of the most important tricks that allow hardware engineers to design very complex circuits.

Note that, in the above example, there are many values of AluOp where the circuit would perform 'strange' operations (1000 for example). This is not really important because the circuit specification says that these inputs were not relevant (this is probably because it can be guaranteed that these inputs will not appear during normal operation).

Draw a block diagram that will implement the ALU operations listed in Table 1. You are free to decide how to implement the ALU and do not have to base the block diagram on the above explanations. Make sure all input and output signals have a unique name. You may use arbitrary size adders, multiplexers, logic gates, zero/sign extend, comparators and shifters.

Implementation

Once we have a good block diagram it is pretty straightforward to implement the circuit in Verilog. Replace each block with a Verilog description and use the signal names in the block diagram.

Start Vivado and begin a new project (you can call it Lab5). Write the Verilog description for the ALU and run the implementation. (We do not transfer the design to FPGA in this lab, therefore we gave you no constraints file. Thus, the implementation will run correctly, but a bitstream generation will fail.)

Hint 1: You can use 32'b0 to represent a 32-bit zero.

Hint 2: In Verilog, you can concatenate multiple bits together using curly braces {}. For example: {2'b10, 1'b1} results in 3'b101.

At this point we really do not know if our circuit functions properly. Unlike the other exercises we cannot verify that our circuit works by directly trying it out since there are too many input bits. Instead, we use a testbench to verify the functionality in the next lab (Lab 6).

Performance of the Circuit

Until now, we did not really care how large or fast our circuit was. We now try to see if we can learn more about our circuit. Were we pursuing a career in digital design, the performance of our circuits would be the main measure by which they would be judged and we would evaluate the effects of various parameters on the performance of the circuit. For the purpose of this course, we do not have the time to go into deeper detail. We just want you to be able to find the performance numbers.

In Vivado after running Implementation, go to ‘Window → Project Summary’. It shows a window similar to the one shown in Figure 4. The design summary window provides many of the important design parameters.

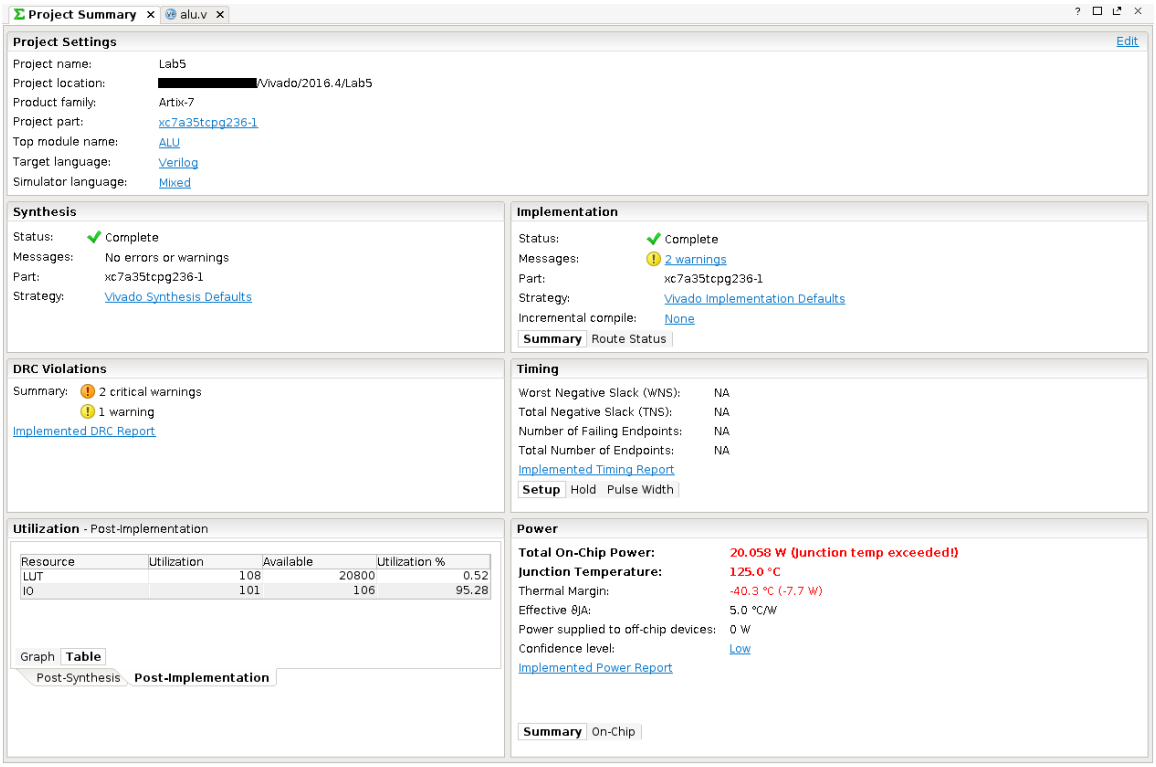

Figure 4. Design Summary window (example)

In the Utilization sub-window (left bottom area), click on the “Table”-button in the “Post-Implementation”-tab. The size of the circuit is expressed in terms of the percent of the total available resources of the FPGA that is utilized for the design. In our case, the number is the number of ‘LUT’s. The above example uses 108 out of 20800 LUTs which is less than 1% of the total.

The timing reports are a little bit trickier. Design tools such as Vivado are not really able to come up with the best possible circuit for your description (this is computationally very challenging). What they will do is to try to come up with a circuit that matches given user constraints. You sort of tell Vivado, “here is the description of the circuit, and I want you to map this description to a circuit so that it works with 50 MHz clock speed”. Vivado tries to satisfy this constraint and reports whether or not it has achieved it. In the Project Summary, it has a section of ‘Timing’, which lists how many of the timing paths violate the given timing constraints. In the above example it is shown as NA (not available). The problem is that we really did not set any constraints for Vivado, so the corresponding reports are not available. If we are actually interested in the timing of the circuit, we must also provide some constraints.

Adding Simple Timing Constraints

All user constraints are added to the XDC file that we have already once used for the connections. If we know how to express timing constraints, we could just go ahead and type in the constraint in a text editor like we did for determining the pins. We can also use a GUI based tool to edit the same file.

In the exercises we always use fairly simple circuits, and adjust the requirements so that exercises can be done easily. In real life, we sometimes need to add many different constraints to get a working circuit. This is why the constraint editor is slightly complex. On the left hand side of the window there is a section named ‘Constraint Type’.

In the Flow Navigator, click on “Implementation → Open Implemented Design → Edit Timing Constraints”. In the newly opened “Timing Constraints”-tab, click in the left tree view on “Exceptions → Set Maximum Delay” and add a new constraint by clicking on the green plus. A new window will pop up as shown in Figure 5. Set “Specify path delay” to 20ns, “From” and “To” to “*” and, click OK.

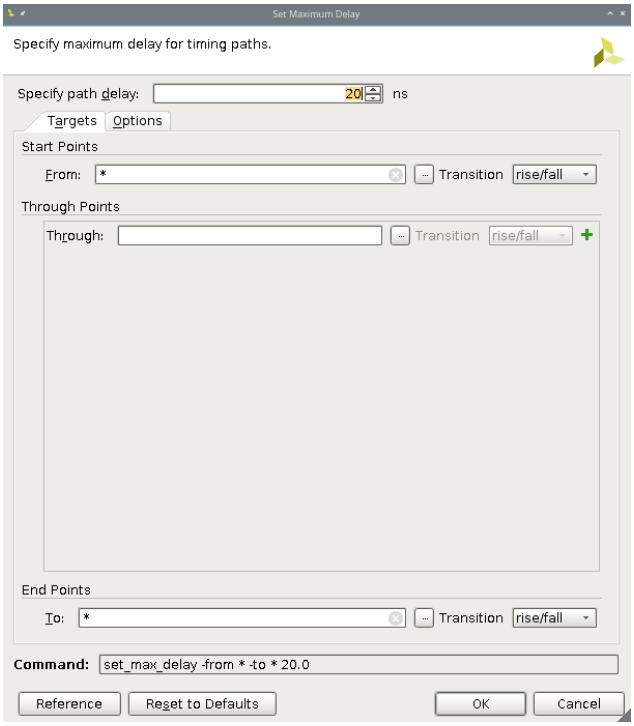

Figure 5. Constraints for the ALU

This essentially tells Vivado that you want to take a maximum of 20 ns to propagate a signal from any input to any output. Press “Ctrl-S” to save and if necessary, create an additional constraint profile. You will see that a XDC file has been added to the design. If you open the file with a text editor, you realize that it is really a simple line:

```
set_max_delay -from * -to * 20.000
```

If you know how the constraint can be expressed, it is usually much easier (and faster) to type in the constraints in a text editor. However, it is not always easy to figure out what exactly to type.

Now that we have a constraint, re-run the implementation. We are interested in the project summary, especially the timing report.

Now in the ‘Project Summary’ section, the entries in ‘Timing’ field should have switched from NA to feasible values. We should see that our constraint of 20 ns was achieved, the slack is the difference between this number and the constraint 20ns (around 1ns in this case).

More detailed reports can be found in “Taskbar → Window → Reports”. For the timing report, select in its tree structure "Implementation -> Route Design -> Timing Summary Report". The slower paths are shown. You see from which input pin it begins, which locations it goes through, and where it ends. At each step you see how much delay comes due to the logic operation, and how much due to routing.

Investigate the different reports to find the answers for the questions on the report sheet. Show the assistants your result in this part.

Last Words

It is possible to design a digital circuit without first developing a block diagram on paper. However, it is always easier to write a hardware description of a circuit that exists as a block diagram. After all, the 'hardware description' is just a translation of the circuit idea into the syntax of the specific language.

Synthesis tools can convert your hardware idea into a working circuit and can report performance on all related numbers. However, if you do not have an expectation of the architecture and the performance, you cannot judge whether or not these are good numbers.

In class we have learned that usually adders are the most critical elements when it comes to determining the performance of an arithmetic circuit. A high-performance adder can be a costly block. In our example, three operations (add, sub, slt) are based on an adder. A naive implementation would have a separate adder for each of these operations, resulting in a relatively large circuit. We should make sure that all three operations are realized by sharing one adder (at least if we are concerned about the area cost of the circuit).

Modern synthesis tools are pretty clever and do most of the work for you. Moreover, they are continuously improving. Chances are very good that they automatically figure out what is the best implementation for your code. Unfortunately, they are far from perfect, and for larger designs with complex functionality (in designs where things matter), experienced design engineers are still indispensable.

Digital Circuits Laboratory Exercises

Lab 5 Implementing an ALU

Date		Grade
Group Number		
Names		Assistant

Part 1

Draw the block diagram of the ALU that implements the functions in Table 1

Part 2

Until now, we have always verified our circuits by exhaustively testing them. Assume that we can test 1 input every second, how long would it take us to test our ALU by trying each and every possible input combination. Please consider only the 7 valid combinations for the AluOp in Table 1. Provide the calculations.

Part 3

(Show to assistant.) Fill in the table below with the results from your circuit and show it to the assistants.

Number of 4 Input LUTs	
Number of bonded IOBs	
Which pin of the FPGA is the output 'zero' connected? (pin name)	
Where does the longest path start from	
Where does the longest path end	
How long is the longest path	
How much of the longest path is routing	
How many levels of logic is in the longest path	

Part 4

If you have any comments about the exercise please add them here: mistakes in the text, difficulty level of the exercise, anything that will help us improve it for the next time.