LAB 5 – Implementing an ALU

Goals

- Implement an Arithmetic Logic Unit (ALU) in Verilog.
- Learn how to evaluate the speed and FPGA resource utilization of a circuit in Vivado.

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 evaluate speed and FPGA resource utilization.
- 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 before the deadline, there is nothing to hand in. The required tasks are clearly marked with gray background throughout this document. All other tasks are optional but highly recommended. You can ask the assistants for feedback on the optional tasks.

Introduction

So far, we implemented fairly small circuits using Verilog. In this exercise we tackle something more formidable, which is the heart of a processor – the arithmetic logic unit (ALU). We implement an ALU that is similar to the one described in Section 5.2.4 of the H&H textbook. This ALU is part of the small micro-controller 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 the delay and area (i.e., FPGA resource utilization) 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 circuit, so we are also interested in how fast the circuit is able to perform the arithmetic operations and what fraction of the available FPGA resources it occupies. We will also try to see whether our coding style has an effect on the speed and FPGA resource utilization.

Part 1 – Designing an 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 the H&H 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	Mnemonic	Result =	Description
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 1: You should extend the result of slt to 32 bits (i.e., 32’b0 or 32’b1).)

(Note 2: And, or, xor, nor are bitwise operations.)

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.

Designing the Block diagram

First, you need to draw a block diagram of the ALU, like the one seen in Figure 5.15 of the H&H 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 operations. 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 bitwise operations. Now let us look at Table 1 and determine for which values of AluOp we perform an operation from which type. It should be 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 type can be selected by a 2-input multiplexer that is controlled by AluOp[2]. Figure 1 depicts an ALU design that includes a separate logic block (i.e., arithmetic part and logic part) for each type of operation.

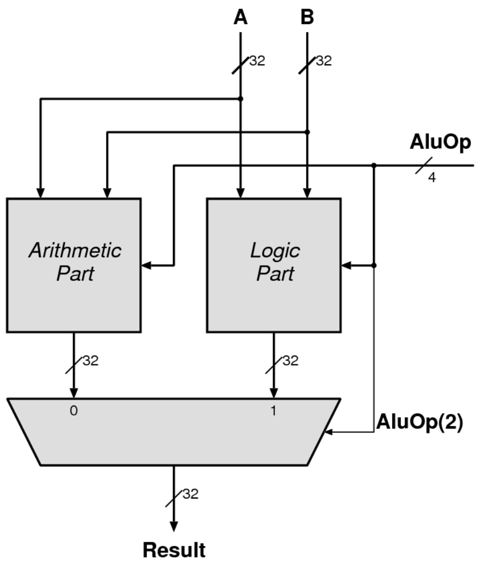


Figure 1. A possible division for the ALU

Now we can take a look at the two types individually. For the logic part, AluOp[1:0] selects one of the 4 simple bitwise operations. In the arithmetic part, we realize that we have an addition (**add**) or a subtraction (**sub**, **slt**). We can see 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 the H&H textbook to design an adder-subtracter (controlled with AluOp[1] instead of F[2]). Figure 2 shows such a design.

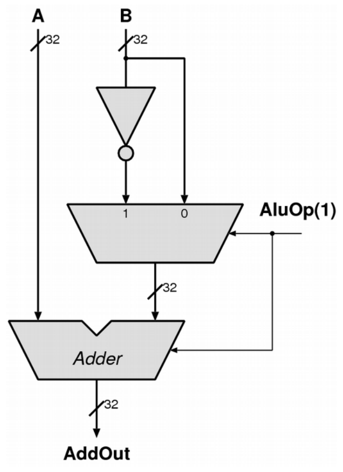


Figure 2. Possible organization for the adder subtracter in ALU

There is one more thing left, depending on the AluOp[3] we can select whether we take only the most significant bit (logic-1, **slt** instruction), or we take the output as it is. We show an example design in Figure 3.

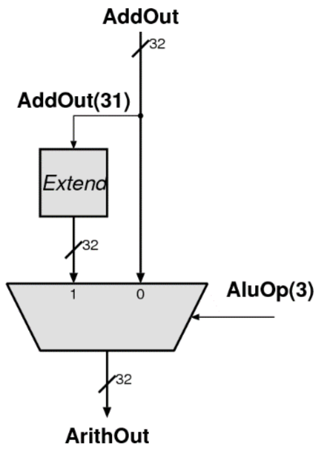


Figure 3. A possible organization to implement slt

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. You may use arbitrary size adders, multiplexers, logic gates, zero/sign extend, comparators and shifters.

Part 2 - Implementation

Once we have a good block diagram it is 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 create a new project (you can call it Lab5). Make sure to select "`xc7a35tcpg236-1`" as your FPGA since otherwise you cannot download the bitstream of your design to the Basys 3 board. Implement the ALU based on your block diagram. Synthesize and implement your design. (We do not transfer the design to FPGA in this lab, therefore we do not provide you a constraints file. Thus, the implementation will run correctly, but the 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.

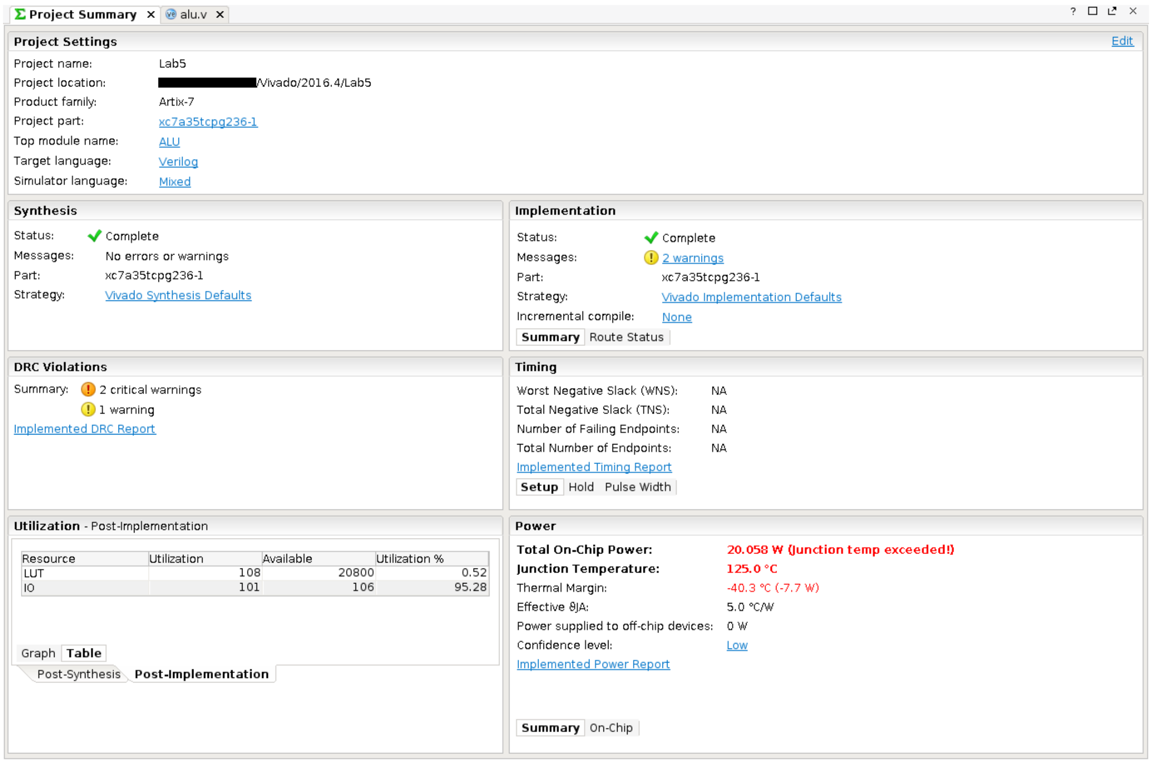
At this point we really do not know if our circuit functions properly. Unlike the other exercises we cannot verify that our circuit works by directly trying it out since there are too many input bits. Instead, we use a testbench to verify the functionality in the next lab (Lab 6).

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 – The Performance of the Circuit

Until now, we did not evaluate the speed and area of our implementation. In this lab, we will learn to check the speed (i.e., max frequency our circuit can run at) and area (i.e., FPGA resource utilization).

We provide instructions for evaluating speed and area using Vivado. 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 (e.g., the *Timing* and *Utilization* panes).


Project Summary × alu.v ×

Project Settings Edit

Project name:	Lab5
Project location:	[redacted]/Vivado/2016.4/Lab5
Product family:	Artix-7
Project part:	xc7a35tcpg236-1
Top module name:	ALU
Target language:	Verilog
Simulator language:	Mixed

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a35tcpg236-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	✓ Complete
Messages:	2 warnings
Part:	xc7a35tcpg236-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

Summary | Route Status

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing

Worst Negative Slack (WNS):	NA
Total Negative Slack (TNS):	NA
Number of Failing Endpoints:	NA
Total Number of Endpoints:	NA

Implemented Timing Report
Setup | Hold | Pulse Width

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	108	20800	0.52
IO	101	106	95.28

Graph | Table
Post-Synthesis | Post-Implementation

Power

Total On-Chip Power:	20.058 W (Junction temp exceeded!)
Junction Temperature:	125.0 °C
Thermal Margin:	-40.3 °C (-7.7 W)
Effective ϑJA:	5.0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report
Summary | On-Chip

Figure 4. Design Summary window (example)

In the *Utilization* pane (left bottom area), click on the "Table" button in the "Post-Implementation" tab. The size of the circuit is expressed in terms of the fraction of the total available resources of the FPGA that used for the design. For instance, the above example uses 108 out of 20800 Look-up Tables (LUTs) of the FPGA, which is less than 1% of the total.

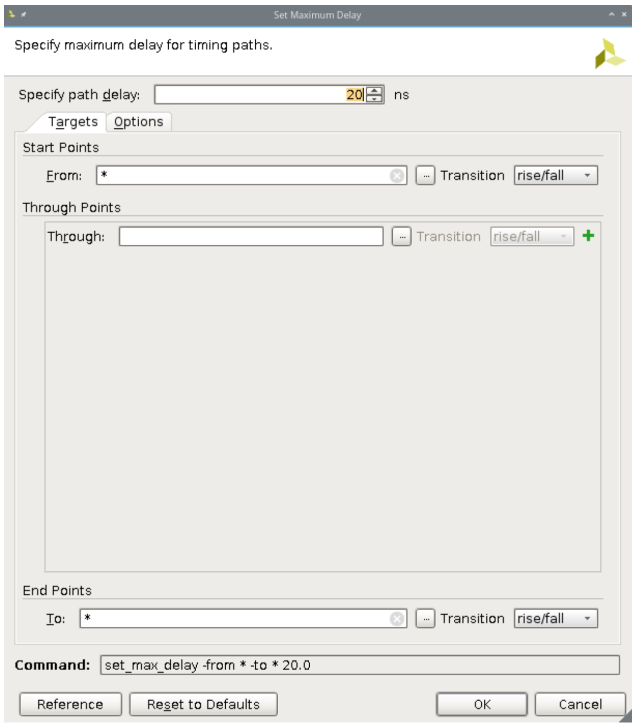
Getting the timing report in Vivado is slightly more complicated . Design tools such as Vivado are not really able to come up with the best possible circuit implementation for a given Verilog description as the placement and routing procedures are computationally expensive. Instead, the tools try to come up with a circuit that satisfies the given user constraints. In other words, you need to tell Vivado, "here is the description of the circuit, and I want you to implement this description so that it works with a clock frequency of 50 MHz". 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). That is because we did not set any timing constraints, so Vivado cannot report the timing. We will add a timing constraint to set the maximum delay that we would like our ALU to have.

Adding Simple Timing Constraints

All user constraints are included in an XDC file that we have previously used for connecting the input/output ports of the top module to the FPGA pins. Make sure to add an XDC file into your project. 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.

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 sign. A new window will pop up as shown in Figure 5. Set "*Specify path delay*" to 20ns, "*From*" and "*To*" to "***" and, click *OK*.


Set Maximum Delay

Specify maximum delay for timing paths.

Specify path delay: 20 ns

Start Points
From: *   Transition rise/fall

Through Points
Through: Transition rise/fall

End Points
To: *   Transition rise/fall

Command: set_max_delay -from * -to * 20.0

Figure 5. Constraints for the ALU

This 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 the file 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 will notice that it includes 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.

Since we have constrained our design, we can re-run the implementation to generate the timing report, which we can see in the *Project Summary*.

After implementing the design, you should see values in the ‘*Timing*’ pane in the ‘*Project Summary*’. You should see that your constraint of 20 ns was achieved. The *slack* (around 1 ns in this case) is the difference between the delay that the circuit actually has and the constraint 20 ns .

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 report provides you the slow paths. You see from which input pin each path begins, which locations it goes through, and where it ends. At each step you see how much delay comes due to a logic operation and routing.

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

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	

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 quite sophisticated 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.