Lab 2: Arithmetic Logic Unit (ALU) Design

In this lab, and the following two labs, we will be implementing a MIPS processor (CPU). The main component of any CPU is the Arithmetic Logic Unit (ALU). This is where all the arithmetic, logic, and relational operations happen. In this Lab, we will be designing and implementing a 32-bit ALU.

1. ALU Description and Operations

Let's have a look at the ALU as a *black box.* Fig.1 below shows a block diagram of the ALU as black box with inputs and outputs.

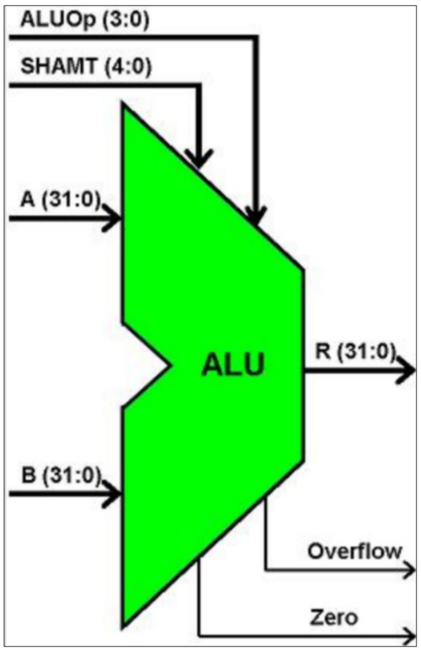


Figure 1: Symbol Representation of the ALU

The description of each of the ports of the ALU is described in the Table 1.

Table 1: ALU ports description

Port	Direction	Description
A(31:0)	In	32-bit input bus carrying the "A" operand for an ALU operation.
B(31:0)	In	32-bit input bus carrying the "B" operand for an ALU operation.
ALUOp(3:0)	In	4-bit input bus which controls which operation the ALU executes.
SHAMT(4:0)	In	5-bit input bus that indicates the number of bits to shift the operand in a shift operation.
R(31:0)	Out	32-bit output bus which will hold the result of the ALU Operation.
Zero	Out	A single bit output which will be set high if the result of an arithmetic operation is equal to zero and low otherwise.
Overflow	Out	A single bit representing if there is an overflow resulting from the operation or not.

The ALU will be capable of carrying out the following MIPS operations, referenced in Table 2. The operation executed will be determined by the ALUOp and SHAMT inputs. The left column describes the operation and its "code" when referenced to the MIPS Instructions Set Architecture (ISA). Don't worry more details on the CPU will come in later labs, but it's just good for you to have a look at the future and know how this component will fit in later on.

Table 2: ALU Operations

Operation	Description
Add Signed (ADD)	R = A + B : Treating A, B, and R as signed two's complement integers.
Add Unsigned (ADDU)	R = A + B : Treating A, B, and R as unsigned integers.
Bitwise AND (AND)	R(i) = A(i) AND B(i).
Bitwise NOR (NOR)	R(i) = A(i) NOR B(i).
Bitwise OR (OR)	R(i) = A(i) OR B(i).
Set on Less Than (SLT)	R = "000...01" if A < B otherwise R = "000....00" : Treating A and B as signed two's-complement integers.
Set on Less Than Unsigned (SLTU)	R = "000....01" if A < B otherwise R = "000....00" : Treating A and B as unsigned integers.
Shift Left Logical (SLL)	R = A << SHAMT : filling in vacated bits with '0'.
Shift Right Arithmetic (SRA)	R = A >> SHAMT : filling in vacated bits with replicas of the sign bit, A(31).
Shift Right Logical (SRL)	R = A >> SHAMT : filling in vacated bits with '0'.
Subtract Signed (SUB)	R = A - B : Treating A, B, and R as signed two's complement integers.
Subtract Unsigned (SUBU)	R = A - B : Treating A, B, and R as unsigned integers.
Bitwise XOR (XOR)	R(i) = A(i) XOR B(i).

2. <u>ALU Components</u>

There are 13 different operations, each of which requires its own unique ALUOp code. Since a 4-bit ALUOp code allows for encoding 16 possible operations, there will be 3 codes which will not be used. Furthermore, the functions can be conveniently grouped into four categories:

- LOGICAL (AND, OR, NOR, XOR);
- ARITHMETIC (ADD, ADDU, SUB, SUBU);
- COMPARISON (SLT, SLTU);
- SHIFT (SLL, SRL, SRA).

In order to increase the speed at which our ALU operates, many different computations will be performed simultaneously and the desired result will be selected by a multiplexor at the output end of the ALU. Since we have four operation types, each of which contains no more than four distinct operations, a natural hierarchy presents itself. We will use the highest 2 bits of ALUOp, ALUOp(3 downto 2), to choose between the different categories as seen in the Table 3.

Table 3: Operation Class Encodings

ALUOp(3)	ALUOp(2)	Function Class
0	**0**	Logic
0	**1**	Arithmetic
1	**0**	Comparison
1	**1**	Shift

Within each operation class, the exact operation which is output as the intermediate result is determined by the lower two bits of the ALUOp, ALUOp(1 downto 0). We will examine the encodings of these two bits for each operation class as we discuss that block.

A block diagram showing the top level of the ALU is shown below:

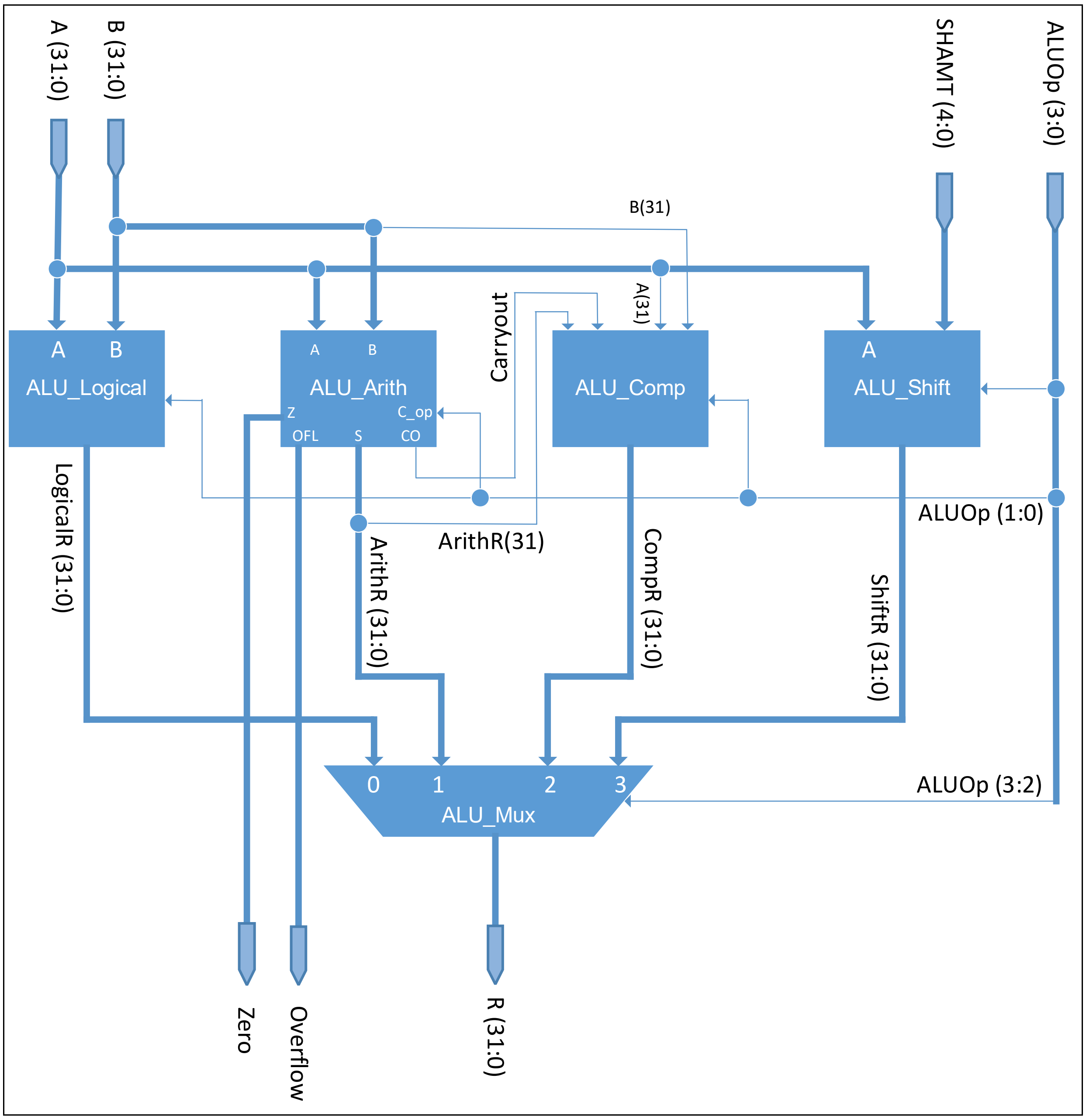


Figure 2: ALU Top Level Diagram

As can be told the input and output ports match those mention in Fig. 1. The Diagram in Fig. 2 should serve as a reference for all the following discussions. It should also be used when you have designed all the blocks and ready to connect them together.

In the following few pages, we are going to discuss each one of these blocks and give the details required their design.

A. Logical Unit:

The logical unit will perform one of four operations: AND, OR, XOR, or NOR. Each of these operations will be performed in parallel on the inputs and the output will be selected by a 32-bit wide 4-1 mux controlled by the ALUOp(1 downto 0) signals. The encodings for the operations are in Table 4:

Table 4: Logic Encodings

ALUOp(1)	ALUOp(0)	Operation
0	**0**	AND
0	**1**	OR
1	**0**	XOR
1	**1**	NOR

This unit have the following in/out ports:

Table 5: Logic Unit Ports Description

Port	Direction	Description
A(31:0)	In	32-bit input bus carrying the "A" operand for an operation.
B(31:0)	In	32-bit input bus carrying the "B" operand for an operation.
Op(1:0)	In	2-bit input bus which controls which operation is selected.
R(31:0)	Out	32-bit output bus which will hold the result of the Logic Operation.

Required:
Write a VHDL code that implements the Logic Unit with the specifications above. Make sure to write a VHDL testbench for it and test its functionality thoroughly.

B. Shifter Unit:

The Shifter unit, as the name implies, is responsible for taking the input operand, A, and shifting it either left or right by a number of bits specified by the SHAMT inputs. SHAMT is interpreted as a 5-bit unsigned integer, thus allowing us to shift the input by zero to thirty-one bits. The direction is specified by the higher order control bit, ALUOp(1):

ALUOp(1)	Operation
0	Shift Left
1	Shift right

In addition to direction, we must also specify whether the shift is arithmetic or logical, but this is only meaningful for right shifts. A logical shift fills the bits vacated at the end with zeros. An arithmetic shift fills the bits vacated at the left end on a right shift with copies of the sign bit, since for signed twos complement integers all bits to the left of the represented number are considered to be the same as the sign. Arithmetic right shifts allow a right shift on a signed integer to be interpreted as a division by 2^{SHAMT}.

The type of shift is specified by the lowest order control bit, ALUOp(0):

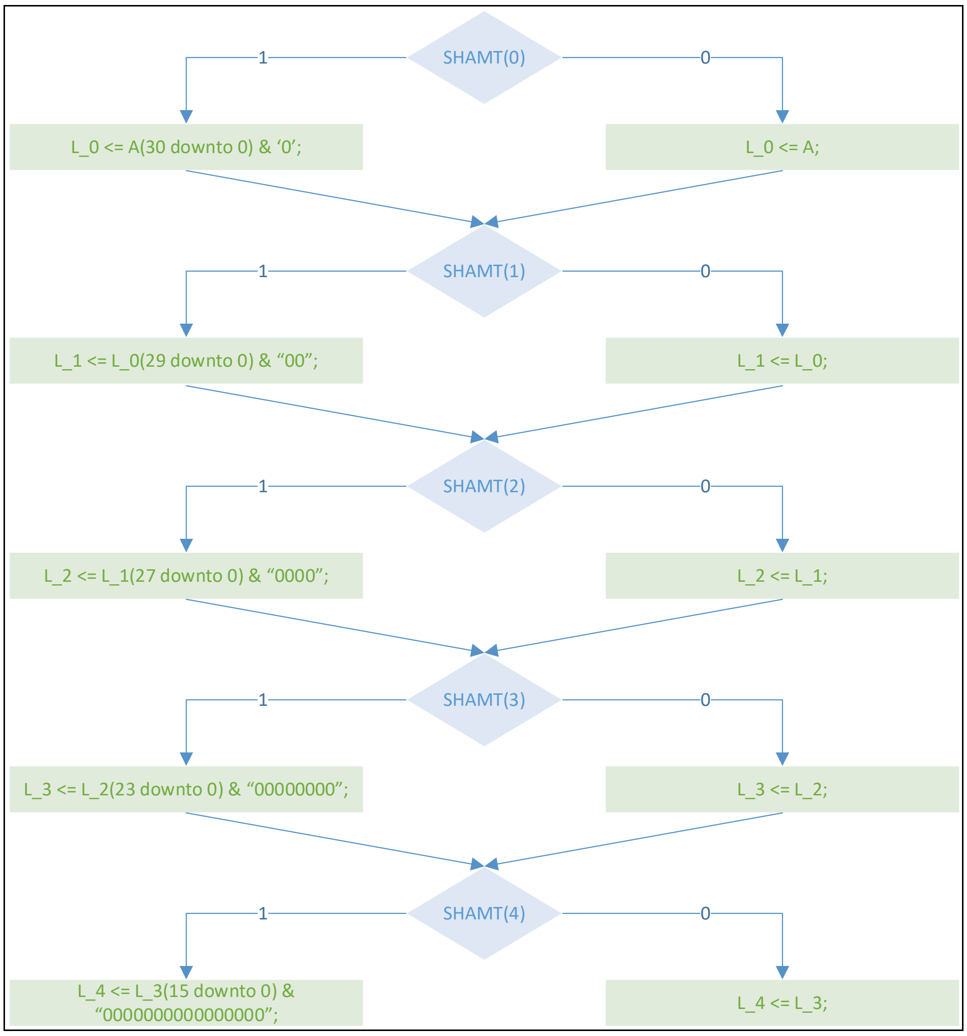
ALUOp(0)	Operation
0	Logical Shift
1	Arithmetic Shift

This unit have the following in/out ports:

Port	Direction	Description
A(31:0)	In	32-bit input bus carrying the "A" operand for an operation.
SHAMT(4:0)	In	5-bit input bus that indicates the number of bits to shift the operand in a shift operation.
ALUOp(1:0)	In	2-bit input bus which controls which operation is selected.
R(31:0)	Out	32-bit output bus which will hold the result of the Logic Operation.

To help you in designing this block, it’s better to illustrate it using flowcharts. It’s really important to understand that the flowcharts here don’t represent sequentially of the statements, but the data flow. Let’s take the shift left case first as it’s easier because we only will be logical shift (no arithmetic shift). The flowchart in the following figure illustrates how to get “ left-shifted” version of A, by “SHAMT” bits.

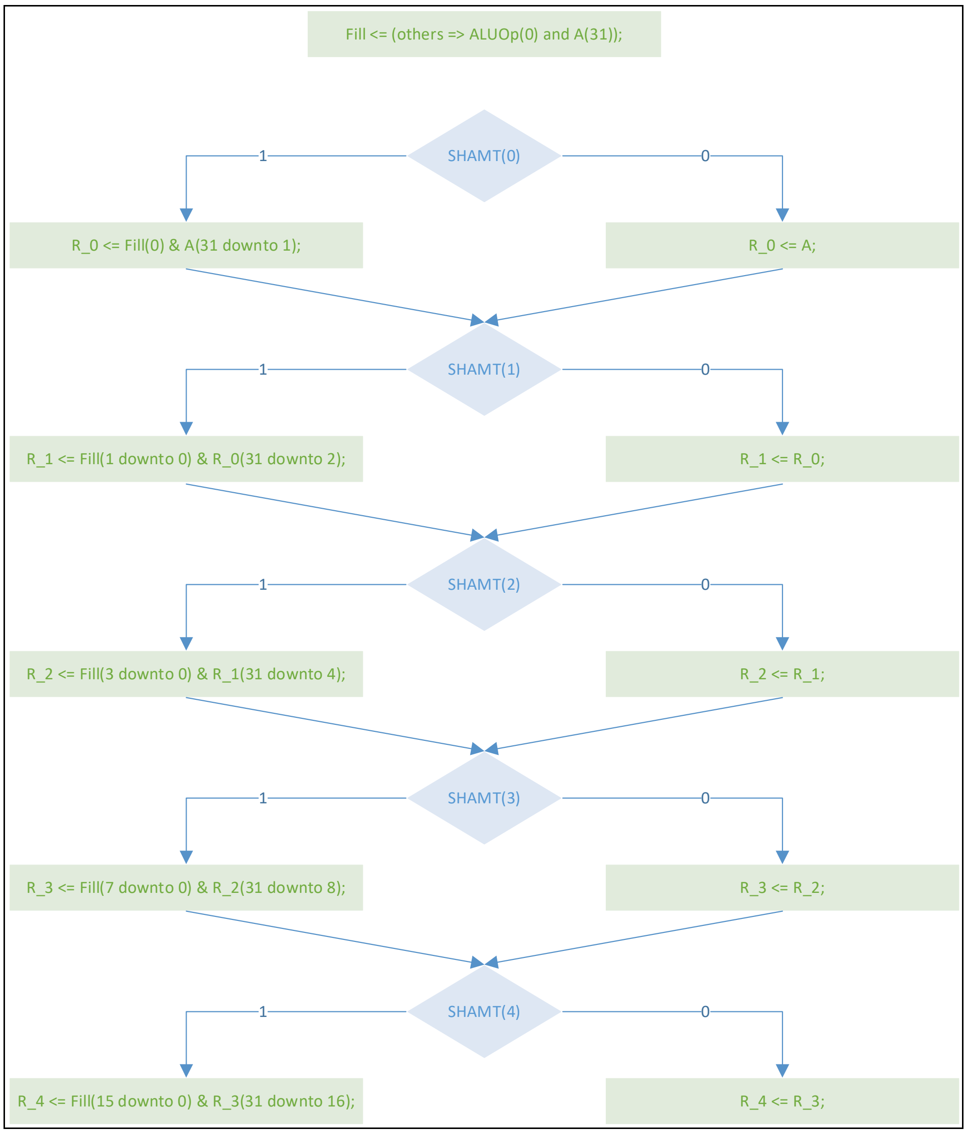
Each “L_(n)” temp signal is shifted version of “L_(n-1)” to the left, by $2^{SHAMT(n)}$, n is [0,4] and that “L_0” is shifted from A. let’s have an example, assume that SHAMT = 01010, which is 10, that means A needs to be shifted to the left 10 bits. This is divided into 5 shift operations, 0-bits left shift between A to L_0, 2-bits left shift between L_0 to L_1, 0-bits left shift between L_1 to L_2, 8-bits left shift between L_2 to L_3, and 0-bits left shift between L_3 to L_4.


SHAMT(0)
1
0
L_0 <= A(30 downto 0) & '0';
L_0 <= A;
SHAMT(1)
L_1 <= L_0(29 downto 0) & "00";
L_1 <= L_0;
SHAMT(2)
L_2 <= L_1(27 downto 0) & "0000";
L_2 <= L_1;
SHAMT(3)
L_3 <= L_2(23 downto 0) & "00000000";
L_3 <= L_2;
SHAMT(4)
L_4 <= L_3(15 downto 0) & "0000000000000000";
L_4 <= L_3;

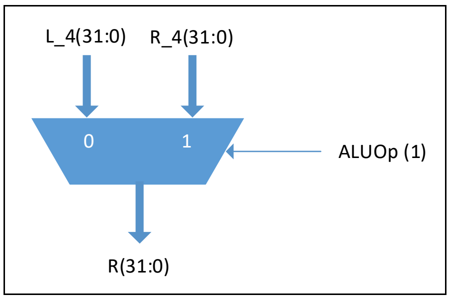
The right shift operation can also be done in a similar way, except that we need to take into consideration whether it arithmetic or logical. A simple way to create a temp signal "Fill" as follows:

```
Fill <= (others => ALUOp(0) and A(31));
```

"Fill" will be zero if ALUOp(0) = 0 (logical shift), or A(31) if ALUOp(0) = 1 (arithmetic shift). Now you can use LSBs of "Fill" to shift right, the same way we used "zeros" when we shifted left. The flowchart in the following figure illustrates how to get " right-shifted" version of A, by "SHAMT" bits.


Fill <= (others => ALUOp(0) and A(31));
SHAMT(0)
1
0
R_0 <= Fill(0) & A(31 downto 1);
R_0 <= A;
SHAMT(1)
1
0
R_1 <= Fill(1 downto 0) & R_0(31 downto 2);
R_1 <= R_0;
SHAMT(2)
1
0
R_2 <= Fill(3 downto 0) & R_1(31 downto 4);
R_2 <= R_1;
SHAMT(3)
1
0
R_3 <= Fill(7 downto 0) & R_2(31 downto 8);
R_3 <= R_2;
SHAMT(4)
1
0
R_4 <= Fill(15 downto 0) & R_3(31 downto 16);
R_4 <= R_3;

Finally, you either select “R_4” or “L_4” depending on “ALUOp(1)” value.



Required:
Write a VHDL code that implements the Shifter Unit with the specifications above. Make sure to write a VHDL testbench for it and test its functionality thoroughly. To help with the testing, we have included the following table test vectors to test Shifter functionality. The table lists a single test pattern for the input A which will be tested with several possible shifts. Although not a completely rigorous testing of the Shifter functionality, it should serve to expose almost any error in your design. One important case has been left out of the test, however, which you will also need to test. We have tested the right logical and arithmetic shifts with a vector that, if interpreted as two's complement, is a negative number. We have not tested to make sure that the right shifts will work correctly with a positive number. Change the

test pattern for the input, A, so that the highest order bit is a '0' instead of a '1'. Run this pattern for several different shift amounts of both arithmetic and logical right shift and verify that the behavior is correct.

A input	SHAMT	SLL Result	SRL Result	SRA Result
X"FEDCBA98"	00000	X"FEDCBA98"	X"FEDCBA98"	X"FEDCBA98"
X"FEDCBA98"	00001	X"FDB97530"	X"7F6E5D4C"	X"FF6E5D4C"
X"FEDCBA98"	00010	X"FB72EA60"	X"3FB72EA6"	X"FFB72EA6"
X"FEDCBA98"	00011	X"F6E5D4C0"	X"1FDB9753"	X"FFDB9753"
X"FEDCBA98"	00100	X"EDCBA980"	X"0FEDCBA9"	X"FFEDCBA9"
X"FEDCBA98"	00101	X"DB975300"	X"07F6E5D4"	X"FFF6E5D4"
X"FEDCBA98"	00110	X"B72EA600"	X"03FB72EA"	X"FFFB72EA"
X"FEDCBA98"	00111	X"6E5D4C00"	X"01FDB975"	X"FFFDB975"
X"FEDCBA98"	01000	X"DCBA9800"	X"00FEDCBA"	X"FFFEDCBA"
X"FEDCBA98"	01001	X"B9753000"	X"007F6E5D"	X"FFFF6E5D"
X"FEDCBA98"	01010	X"72EA6000"	X"003FB72E"	X"FFFFB72E"
X"FEDCBA98"	01011	X"E5D4C000"	X"001FDB97"	X"FFFFDB97"
X"FEDCBA98"	01100	X"CBA98000"	X"000FEDCB"	X"FFFFEDCB"
X"FEDCBA98"	01101	X"97530000"	X"0007F6E5"	X"FFFFF6E5"
X"FEDCBA98"	01110	X"2EA60000"	X"0003FB72"	X"FFFFFB72"
X"FEDCBA98"	01111	X"5D4C0000"	X"0001FDB9"	X"FFFFFDB9"
X"FEDCBA98"	10000	X"BA980000"	X"0000FEDC"	X"FFFFFEDC"
X"FEDCBA98"	10001	X"75300000"	X"00007F6E"	X"FFFFFF6E"
X"FEDCBA98"	10010	X"EA600000"	X"00003FB7"	X"FFFFFFB7"
X"FEDCBA98"	10011	X"D4C00000"	X"00001FDB"	X"FFFFFFDB"
X"FEDCBA98"	10100	X"A9800000"	X"00000FED"	X"FFFFFFED"
X"FEDCBA98"	10101	X"53000000"	X"000007F6"	X"FFFFFFF6"
X"FEDCBA98"	10110	X"A6000000"	X"000003FB"	X"FFFFFFFB"
X"FEDCBA98"	10111	X"4C000000"	X"000001FD"	X"FFFFFFFD"
X"FEDCBA98"	11000	X"98000000"	X"000000FE"	X"FFFFFFFE"
X"FEDCBA98"	11001	X"30000000"	X"0000007F"	X"FFFFFFFF"
X"FEDCBA98"	11010	X"60000000"	X"0000003F"	X"FFFFFFFF"
X"FEDCBA98"	11011	X"C0000000"	X"0000001F"	X"FFFFFFFF"
X"FEDCBA98"	11100	X"80000000"	X"0000000F"	X"FFFFFFFF"
X"FEDCBA98"	11101	X"00000000"	X"00000007"	X"FFFFFFFF"
X"FEDCBA98"	11110	X"00000000"	X"00000003"	X"FFFFFFFF"
X"FEDCBA98"	11111	X"00000000"	X"00000001"	X"FFFFFFFF"

C. Arithmetic Unit:

The Arithmetic unit is required to support four integer arithmetic operations: Signed Addition (ADD), Unsigned Addition (ADDU), Signed Subtraction (SUB), and Unsigned Subtraction (SUBU). For signed operations, the inputs and output will be treated as 32-bit signed twos-complement integers. For your convenience and without getting into much details about the implementation of such a unit, The Arithmetic Unit is given to in the IP repository “ECE1195_repo” given with this. You can add the IP repository to your project and instantiate the “Arith_Unit” Component. This repo also has another

component that we will use later on in some advanced labs.
The following table describes the in/out ports of the Arithmetic Unit and their description:

Port	Direction	Description
A(31:0)	In	32-bit input bus carrying the "A" operand for an operation.
B(31:0)	In	32-bit input bus carrying the "B" operand for an operation.
C_op(1:0)	In	2-bit input bus which controls which operation is selected.
CO	Out	A single bit representing if there is a carryout resulting from the operation or not.
S(31:0)	Out	32-bit output bus which will hold the result of the Arithmetic operation.
OFL	Out	A single bit representing if there is an overflow resulting from the operation or not.
Z	Out	A single bit representing if S resulting from an operation is equal to zero or not.

The “C_op” control which operation is selected and it’s explained in the table below

C_Op(1)	C_Op(0)	Operation
0	**0**	Signed ADD (ADD)
0	**1**	Unsigned ADD (ADDU)
1	**0**	Signed SUB (SUB)
1	**1**	Unsigned SUB (SUBU)

The “C_op” can be directly connected to “ALUOp(1:0)” to achieve the required function.

Note: Although that the Arithmetic Unit was already given to you, it’s really helpful to have a look at the VHDL implementation of it to understand how it’s implemented.

Required:
When you are done implementing all the units, add the “ECE1195_repo” to your top project and instantiate an instance of “Artih_Unit” and connect it with the rest of the blocks.

D. Comparator Unit:

For the comparison operations, Set on Less Than (SLT) and Set on Less Than Unsigned (SLTU), we wish to determine whether the input A is less than the input B. If it is, we wish to set the result to X"00000001". If it is not, we wish to set the result to X"00000000".
The in/out ports for the Comparison Unit are:

Port	Direction	Description
A_31	In	A single bit carrying the 31st bit of operand A (the sign bit)
B_31	In	A single bit carrying the 31st bit of operand B (the sign bit)
S_31	In	A single bit carrying the 31st bit of Arith_Unit result (the sign bit)
CO	In	A single bit carrying the carryout bit from the Arith_Unit
ALUOp(1:0)	In	2-bit input bus to control the operation
R(31:0)	Out	32-bit output bus which will hold the result of the Comparison operation.

Let's start by looking at the calculation we wish to perform: A < B. Using a little simple algebra, this expression can be rewritten as A - B < 0. If we were to subtract B from A, then all that would remain is the relatively trivial matter of determining whether the result is less than zero. Instead of a large comparator which will consume a number of gates, we can once again reuse Arith_Unit outputs with a minimal amount of logic.

First, we must make sure that our existing Arithmetic Unit is performing a subtraction operation. This will occur when ALUOp(1 DOWNTO 0) is either "10" or "11". Fortunately, we only have two comparison operations to perform. SLT will require a signed subtraction, thus it will be encoded as ALUOp(1 DOWNTO 0) = "10". SLTU will require an unsigned subtraction, thus it will be encoded as ALUOp(1 DOWNTO 0) = "11". The remaining two encodings for this unit will be undefined.

The encodings are summarized in the following table:

ALUOp(1)	ALUOp(0)	Operation
1	**0**	SLT
1	**1**	SLTU

Now, the question is how do we determine if the result is less than zero based on the four inputs which we have defined to the Comparison Unit: CarryOut from the Arith_Unit, Bit 31 of the subtractor result (S_31), Bit 31 of the input A (A_31), and Bit 31 of the input B (B_31).

<u>Signed Comparitor:</u>

Let's start with the Signed version. There are four possible combinations of the signs of the inputs: both are positive; both are negative; A is positive and B is negative; A is negative and B is positive. When the signs of the inputs are different, our problem is trivial. If A is negative and B positive, then A MUST be less than B. Conversely, if A is positive and B is negative then A CAN NEVER be less than B. When the inputs have the same sign, we need to look at the result of the signed subtraction. Two inputs with the same sign can never cause an overflow in subtraction, so we do not have to concern ourselves with an incorrect answer due to that. If the inputs are of the same sign, then whenever A is smaller than B the result of subtracting A - B MUST be a negative number. So, if the sign of the result, SignR, is negative we have A < B and if it is positive then we have A > B.

<u>Unsigned Comparitor:</u>

For the unsigned version of the comparison we have a simple test. In unsigned representation, A and B are both positive numbers. Subtracting two positive numbers will result in a positive number if A > B or a negative number if A < B. However, we cannot represent a negative number with an unsigned integer: if we try we will have an overflow condition. Overflow for unsigned subtraction means that the result is a negative number (it is impossible to subtract two unsigned numbers and get a result too large to represent). We can test for the occurrence of overflow in unsigned subtraction by looking at the Carry Out bit. In unsigned addition, the existence of a Carry Out indicates overflow. In unsigned subtraction, however, the situation is reversed: Carry Out high is the normal state and Carry Out low indicates overflow. Thus, since we are performing unsigned subtraction in our SLTU, if there is no Carry Out, then we had an overflow and A - B is less than zero and A < B is true. If there is a Carry Out, then A - B is greater than zero and A < B is false.

The best way to represent the previous explanation is by printing out the Truth Table of this Unit, as shown in the following Table:

ALUOp(1)	ALUOp(0)	A_31	B_31	S_31	CO	R
0	0	x	x	x	x	X"00000000"
0	1	x	x	x	x	X"00000000"
1	0	0	0	0	x	X"00000000"
1	0	0	0	1	x	X"00000001"
1	0	1	1	0	x	X"00000000"
1	0	1	1	1	x	X"00000001"
1	0	1	0	x	x	X"00000001"
1	0	0	1	x	x	X"00000000"
1	1	x	x	x	1	X"00000000"
1	1	x	x	x	0	X"00000001"

* x here means don’t care.

As can be told from the truth table, we are only changing the LSB of R, R(0), and the rest of the bits are always 0. The best way to implement the comparison block to draw the Karnaugh maps for R(0), determine the equations and implement it using a dataflow architecture.

<u>Required:</u>
Write a VHDL code that implements the Comparison Unit with the specifications above. Make sure to write a VHDL testbench for it and test its functionality thoroughly.
Although the Comparison Unit is difficult to fully test in a stand-alone manner due to the fact that it requires the Arithmetic Unit to function correctly, it makes sense to write a simple testbench just to verify the correct implementation of the above truth table. The full test of the Comparison Unit, with actual full 32-bit operands, can be done at the end when you are testing the full ALU unit.

What to do:

1. Design each individual unit and test them thoroughly, as indicated by the **required** section under each unit.
2. Design the ALU, per the diagram shown in Fig. 2. Make sure that the ports' name in your entity matches the letter case and naming in Table 1. Create the top level ALU using structural-architecture VHDL code.
3. Write a VHDL testbench to verify the functionality of the ALU. You should at least test each ALU operation once (refer to Table 2).
4. Synthesize, Implement and Generate Bitstream for your ALU, then write C/C++ testbench to fully verify the functionality of your FPGA-configured design. You should at least test each ALU operation once (refer to Table 2). You will be needing more `regmap` registers for this Lab, so make sure to configure and take notes of their numbers and bit-width.

Deliverables:

1. You will be delivering all the VHDL and C codes that you have written, including the design files, testbenches, and the C codes. The submission will on Canvas Assignment for Lab 2. You can just Zip the top Vivado project folder that has all the files that was mentioned above. (please make sure that it has all the files required).
2. Check-offs will be done on the due date for this assignment. The rubric for the check-offs will be provided later.