**CS401-1 MIPS 1 – Processor Building Blocks and the ALU**

NAME(S): Maxa, Scott Battista, Jude

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| **Exercise 1** |  | 15 |
| **Exercise 2** |  | 15 |
| **Exercise 3** |  | 50 |
| **Code Review / Walkthrough** |  | 20 |
| **TOTAL** |  | 100 |

## For this laboratory you will:

* Fill in the answers to the questions in this document and submit it on blackboard for assignment MIPS 1.
* **Make sure you both have a copy of the final document.** For the next assignment we will switch up the groups and your new group will need to decide which route to take.
* Research ALU designs and in groups of two design your own ALU.
* Practice building more complex test benches in VHDL.

## Group project tips:

* Decide on a means of communication. Groups are setup on blackboard and it has tools for communication. Or you could use slack. In any case, make sure you have an efficient communication channel in case something happens.
* Decide on time(s) to meet. You will have time in class today, but not next class period. You will need to work on this outside of class.
* Google “pair programming kindergarten”. Read the PDF by Laurie Williams. Follow these guidelines and treat each other with respect. Listen to one another. If there are issues that can’t be resolved through the tips in this document please talk to your instructor.

## Important: data type conversions in VHDL

**When you examine the code, you will notice the various data type conversions** and **typecasts**. The next figure demonstrates how to convert (and typecast) between various data types in VHDL.



#### Exercise 1: Generic N Bit Adder in VHDL

The following VHDL code implements a generic N bit adder.

1. Using the following VHDL module, make a top-level test bench that uses this code to **implement a 4 bit adder**.
2. Review **example test bench code** from the lecture notes make a test bench for this adder.
3. Pretend you are an Intel engineer. You don’t want to lose your job because the adder fails, so you want to verify it carefully.
4. Write test bench code that can check all combinations of inputs **a** and **b**.

**-------------------------------------------------------------------**

**-- Module Name: adder - Behavioral**

**-------------------------------------------------------------------**

**library IEEE; use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity adder is**

**generic ( N: integer := 8 );**

**port ( a, b: in STD\_LOGIC\_VECTOR( N-1 downto 0 );**

**cin: in STD\_LOGIC;**

**sum: out STD\_LOGIC\_VECTOR( N-1 downto 0 );**

**cout: out STD\_LOGIC );**

**end adder;**

**architecture Behavioral of adder is**

**signal result: STD\_LOGIC\_VECTOR( N downto 0 );**

**begin**

**result <= ( "0" & a ) + ( "0" & b ) + cin;**

**sum <= result( N-1 downto 0 );**

**cout <= result(N);**

**end Behavioral;**

#### Exercise 2: 32 Bit ALU – Comparing ALU Designs for Hardware Utilization Efficiency

1. **The following VHDL code implements a generic N bit ALU from your textbook. Notice there are a couple of changes to accommodate the change to IEEE.NUMERIC\_STD.all …**
2. Load the following ALU into a VHDL project.
3. Run synthesis and implementation and then check the utilization report (submenu “Report Utilization” under “Open Implemented Design”)
4. What is a Slice or LUT? Quickly read about an Artix 7 FPGA slice starting on page 18 of the following document (don’t spend a huge amount of time here): <https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf>

A slice is a set of resources used by an FPGA. For an Artix 7 FPGA, a slice contains four logic generators or lookup tables, 8 storage elements, a wide-function multiplexer, and carry logic. An LUT, or lookup table, is the logic generator used to determine whether a combination of inputs is true or false depending on the LUT used.

1. How many LUT’s of the FPGA (Lookup tables) are used in the following design? 65
2. How many Slices (out of 1585 ) are used in the following design? 18
3. This ALU has 7 instructions. What is the ratio of slices to instructions? (slices / instruction) 18 / 7 = 2.57

**-**------------------------------------------------

-- Module Name: alu - Behavioral

-------------------------------------------------

**library** IEEE**;**

**use** IEEE**.**STD\_LOGIC\_1164**.all;**

**use** IEEE**.**STD\_LOGIC\_UNSIGNED**.all;**

**use** IEEE**.**NUMERIC\_STD**.all;** -- use this instead of STD\_LOGIC\_ARITH

**entity** alu **is**

**generic** **(** N **:** integer **:=** 32 **);**

**port** **(** a**,** b **:** **in** STD\_LOGIC\_VECTOR**(** N**-**1 **downto** 0 **);**

f **:** **in** STD\_LOGIC\_VECTOR**(** 2 **downto** 0 **);**

Y **:** **out** STD\_LOGIC\_VECTOR**(** N**-**1 **downto** 0 **)** **);**

**end** alu**;**

**architecture** Behavioral **of** alu **is**

**signal** sum**,** bout **:** STD\_LOGIC\_VECTOR**(** N**-**1 **downto** 0 **);**

**begin**

bout **<=** B **when** **(** f**(**2**)** **=** '0' **)** **else** not B**;**

sum **<=** a **+** bout **+** f**(**2**);** -- 2's complement depends on f(2)

**process** **(** a**,** b**,** f**(**1 **downto** 0**),** sum**,** bout **)**

**begin**

**case** f**(**1 **downto** 0**)** **is**

**when** "00" **=>** y **<=** a and bout**;**

**when** "01" **=>** y **<=** a or bout**;**

**when** "10" **=>** y **<=** sum**;**

**when** "11" **=>** y **<=** **(**N**-**1 **downto** 1 **=>** '0'**)** **&** sum**(**N**-**1**);** -- zero extend

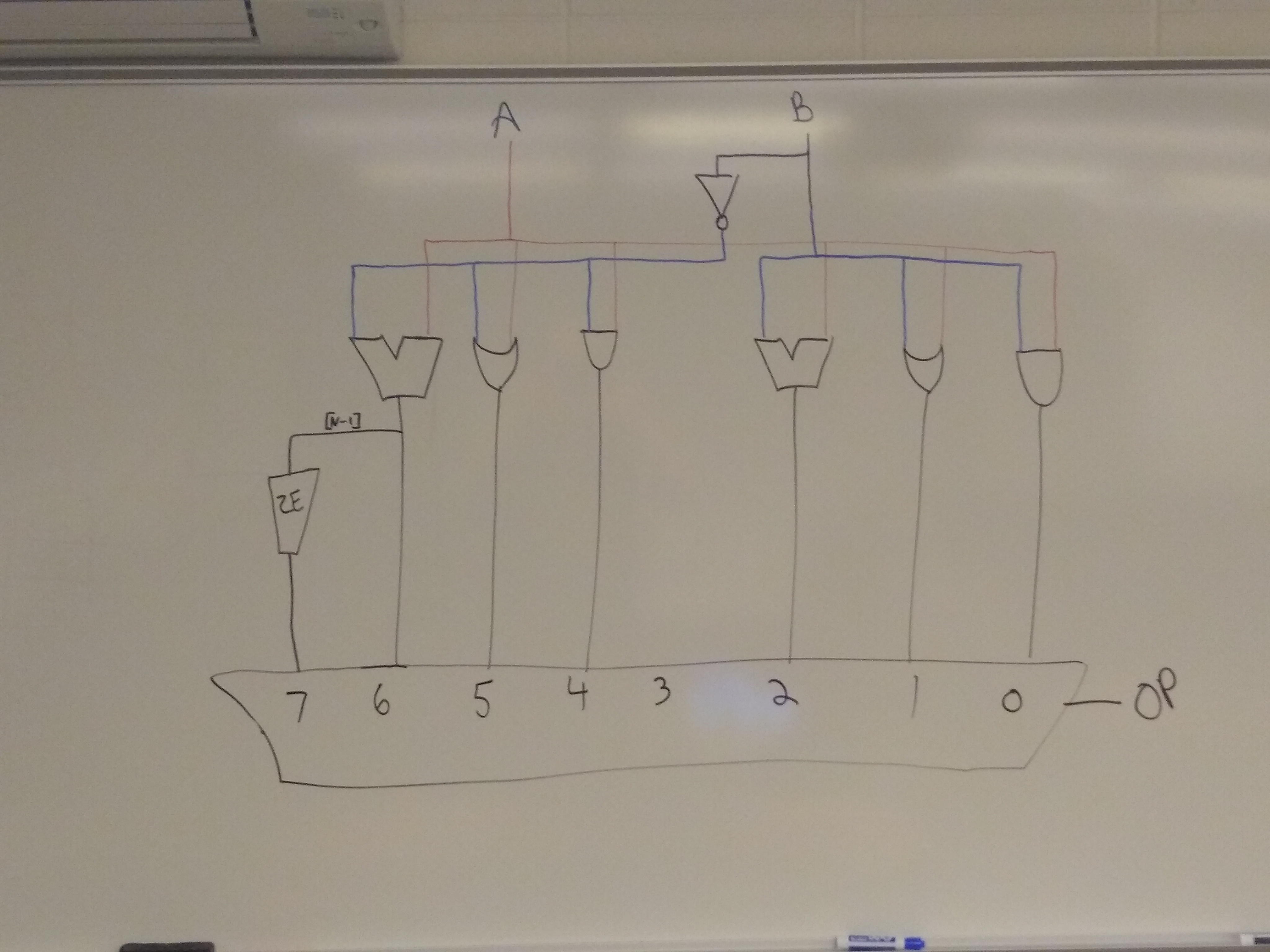
**when** **others** **=>** Y **<=** **(others** **=>** 'X'**);**

**end** **case;**

**end** **process;**

**end** Behavioral**;**

1. **The following VHDL was originally given on Wikipedia. Some modifications were made to match the instructions for this ALU design to the design given in part A above.**
2. Reverse engineer the following VHDL code and draw a hardware diagram (similar to that shown above in part A) for this implementation. Try to make the hardware drawing as accurate as possible. You can take a picture of your drawing, crop it, and paste it in this document right here, or you can draw an electronic version and paste it here.:



1. Load the following ALU into a VHDL project. Open the elaborated design schematic and compare it to what you drew in part 1. How does your drawing compare to what VHDL created? Write your answer here:

Both used a naïve approach to hardware implementation, using a separate gate for every operation. One thing we neglected was to have a dedicated subtraction operator and run B to it. Instead we ran ~B to an addition gate, forgetting that we would need an additional 1 value to complete two’s complement addition.

1. Run synthesis and implementation and then check the utilization report (submenu “Report Utilization” under “Open Implemented Design”)
2. How many Slice LUT’s of the FPGA (Lookup tables) are used by this ALU design? 128
3. How many Slices (out of 1585 ) are used in this design? 34
4. How may Slices / Instruction did this ALU achieve? 4.86
5. This ALU has 7 instructions. What is the ratio of slices to instructions? (slices / instruction) 4.86

-------------------------------------------------

-- Module Name: alu - Wikipedia

-------------------------------------------------

**library** IEEE**;**

**use** IEEE**.**STD\_LOGIC\_1164**.all;**

**use** IEEE**.**STD\_LOGIC\_UNSIGNED**.all;**

**use** IEEE**.**NUMERIC\_STD**.all;**

**entity** alu **is**

**generic** **(** N **:** integer **:=** 32 **);**

**port** **(** -- the alu connections to external circuitry:

A **:** **in** signed**(**N**-**1 **downto** 0**);** -- operand A

B **:** **in** signed**(**N**-**1 **downto** 0**);** -- operand B

OP **:** **in** unsigned**(**2 **downto** 0**);** -- opcode

Y **:** **out** signed**(**N**-**1 **downto** 0**));** -- operation result

**end** alu**;**

**architecture** behavioral **of** alu **is**

**signal** diff **:** signed**(**N**-**1 **downto** 0**);**

**begin**

diff **<=** A **-** B**;**

**process** **(** OP **)**

**begin**

**case** OP **is** -- decode the opcode and perform the operation:

**when** "000" **=>** Y **<=** A and B**;** -- A & B

**when** "001" **=>** Y **<=** A or B**;** -- A | B

**when** "010" **=>** Y **<=** A **+** B**;** -- A + B

**when** "011" **=>** Y **<=** **(others** **=>** 'X'**);** -- not implemented

**when** "100" **=>** Y **<=** A and not B**;** -- A & ~B

**when** "101" **=>** Y **<=** A or not B**;** -- A | ~B

**when** "110" **=>** Y **<=** diff**;** -- A - B

**when** "111" **=>** Y **<=** **(**N**-**1 **downto** 1 **=>** '0'**)** **&** diff**(**N**-**1**);** -- SLT

**when** **others** **=>** Y **<=** **(others** **=>** 'X'**);** -- not implemented

**end** **case;**

**end** **process;**

**end** behavioral**;**

1. **Which design uses less resources on the FPGA (has a smaller slice/instruction ratio)? How was this reduction in resources achieved given the fact that both designs are implementing exactly the same ALU instructions?**

The design in part A used fewer slices per instruction. The more efficient design reused gates by using a MUX to select between B and ~B rather than feeding each version of B to a different network of logic gates. It also cleverly avoids the need for a subtraction operator by using two’s complement addition and using the high bit of the opcode as the +1 required by two’s complement addition. The less efficient design uses a new gate for every comparison rather than reusing gates and manipulating the inputs.

#### Exercise 3: 32 Bit ALU Project (Groups of 2)

Think about the ALU and the operations you would like it to support in your microprocessor. There are several competing concerns when designing an ALU:

* The number of hardware resources used to implement the design (e.g. total number of logic gates),
* The number of useful operations that the ALU can perform.
* The efficiency of those operations. For example, we can implement any algorithm using just an adder, however, that may not be efficient enough for the types of programs we want to run. (e.g. if we need to do a large number of multiplications, then we need a program to compute multiplication which is not as efficient as hardware based multiplication.)
* ***Advanced, not required***: Do you want floating point operations? You don’t need floating point operations to meet the requirements for this class. However, they are both cool and costly. Floating point operations take a significant amount of hardware resources. You can either attempt to design your own floating point computations, or you can use Vivado’s Intellectual Property (IP) core library that has a floating point unit. With either approach there are a lot of details, and it will take you a significant amount of time to figure out. Here is the documentation for the IP floating point library: <https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf>

Obviously we need enough instructions to make the processor capable of doing useful work in an efficient manner. Think about a generic C++ program. What sorts of operations are used (e.g. arithmetic, logic, comparisons, etc.)? What ALU operations do you need?

In groups of two, complete the following steps and put your answers in this document:

1. What ALU instructions do you want ( or need )?

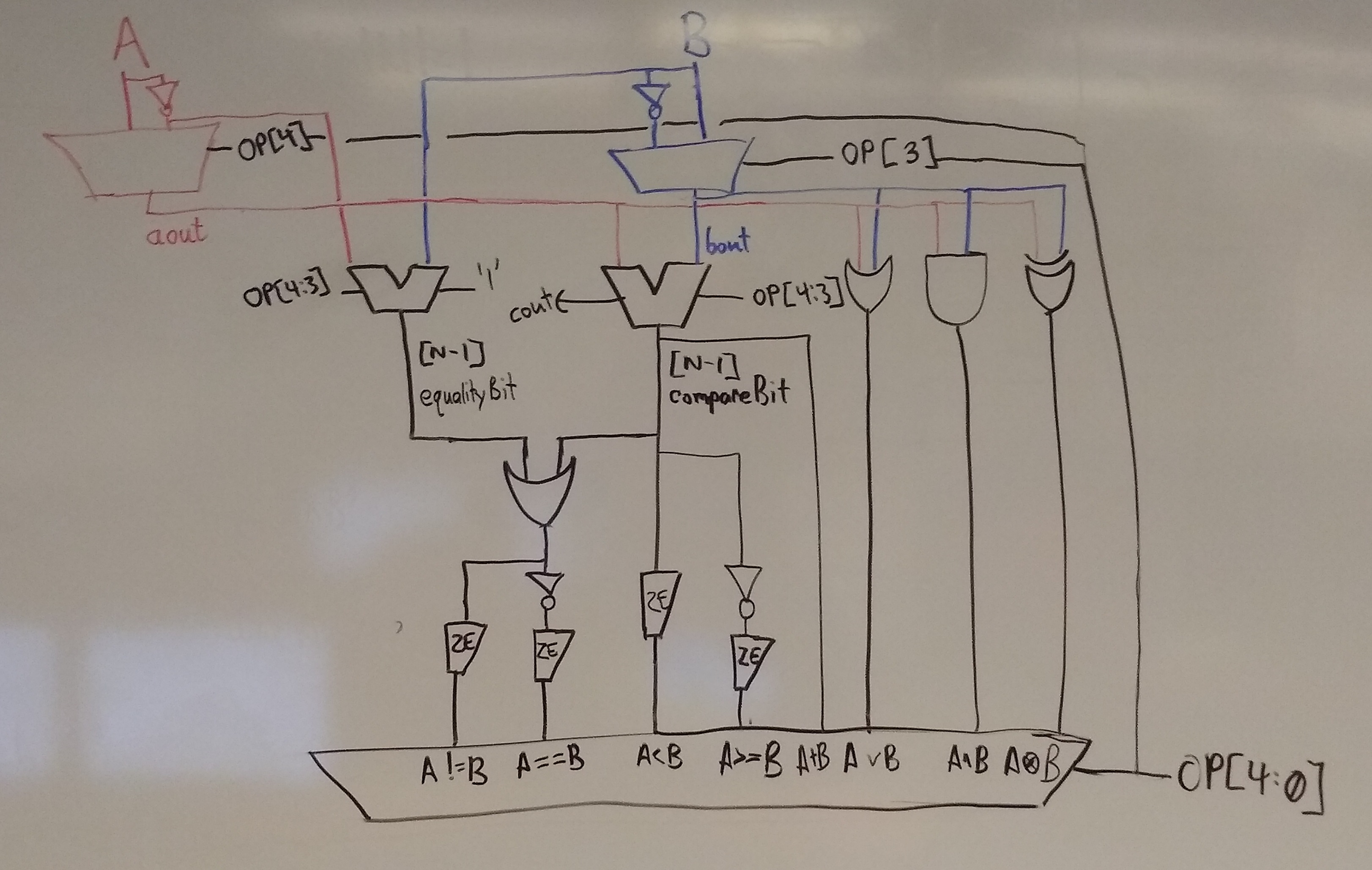
* Xilinx (the maker of our Artix-7 FPGA) has a softcore processor called MicroBlaze. If you open the following document and search for ALU, on page 10 of the document you will see a block diagram of their processor. You will also see the ALU diagram which shows some of the operations it implements. <https://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf>
* Intel has designed a Nios II processor for its family of FPGA. You can see what ALU operations they support in this document: <https://www.intel.com/content/www/us/en/programmable/documentation/iga1420498949526.html#iga1409259983159>

In your group of two, decide what ALU operations you feel you will need in order to make your processor capable of having a useful instruction set. List them here:

A and B, A and ~B, ~A and B, ~A and ~B, A or B, A or ~B, ~A or B, ~A or ~B, A + B, A - B, B – A, A == B, A != B , A >= B, A <= B,

A > B, A < B, A xor B, A xor ~B, ~A xor B, ~A xor ~B.

1. Design the hardware for your ALU that will implement the instructions you have chosen. Draw the hardware diagram. Identify ALL the signals: The ALU inputs, outputs, and control signals:



1. Implement the VHDL for your ALU. Include a neatly formatted version of your VHDL code here:

-------------------------------------------------

-- Module Name: nBitAlu - Behavioral

-------------------------------------------------

library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.STD\_LOGIC\_UNSIGNED.all;

use IEEE.NUMERIC\_STD.all; -- use this instead of STD\_LOGIC\_ARITH

entity nBitAlu is

generic ( N : integer := 32 );

port ( a, b : in STD\_LOGIC\_VECTOR( N-1 downto 0 );

op : in STD\_LOGIC\_VECTOR( 4 downto 0 );

y : out STD\_LOGIC\_VECTOR( N-1 downto 0 ) );

end nBitAlu;

architecture Behavioral of nBitAlu is

signal sum, aout, bout, equality : STD\_LOGIC\_VECTOR( N-1 downto 0 );

signal compareBit, equalityBit : STD\_LOGIC;

begin

-- generate muxen to choose between A/~A and B/~B

aout <= a when ( op(4) = '0' ) else not a;

bout <= b when ( op(3) = '0' ) else not b;

sum <= aout + bout + (op(4) or op(3)); -- 2's complement depends on op(4) for A and op(3) for B

compareBit <= sum(N-1);

equality <= b-a;

equalityBit <= equality(N-1);

process ( a, b, op(4 downto 0), sum, equality, aout, bout, compareBit, equalityBit)

begin

case op(4 downto 0) is

when "00000" => y <= aout and bout; -- a && b

when "01000" => y <= aout and bout; -- a && ~b

when "10000" => y <= aout and bout; -- ~a && b

when "11000" => y <= aout and bout; -- ~a && ~b

when "00001" => y <= aout or bout; -- a || b

when "01001" => y <= aout or bout; -- a || ~b

when "10001" => y <= aout or bout; -- ~a || b

when "11001" => y <= aout or bout; -- ~a || ~b

when "00010" => y <= sum; -- a + b

when "01010" => y <= sum; -- a - b

when "10010" => y <= sum; -- b - a

when "01011" => y <= (N-1 downto 1 => '0') & compareBit; -- a < b

when "10011" => y <= (N-1 downto 1 => '0') & compareBit; -- a > b

when "01100" => y <= (N-1 downto 1 => '0') & not compareBit; -- a >= b

when "10100" => y <= (N-1 downto 1 => '0') & not compareBit; -- a <= b

when "01101" => y <= (N-1 downto 1 => '0') & not(compareBit or equalityBit); -- a == b

when "01110" => y <= (N-1 downto 1 => '0') & (compareBit or equalityBit); -- a != b

when "00111" => y <= aout xor bout; -- a xor b

when "01111" => y <= aout xor bout; -- a xor ~b

when "10111" => y <= aout xor bout; -- ~a xor b

when "11111" => y <= aout xor bout; -- ~a xor ~b

when others => y <= (others => 'X');

end case;

end process;

end Behavioral;

1. Design VHDL test bench code that checks every instruction your ALU implements. This does not have to exhaustively check every single possible addition. Just make a simple test that checks every ALU operation. Include a neatly formatted version of your VHDL test bench code here:

--------------------------------------------------------------------------------

-- Module Name: Example of a programmed test bench

-- Project Name: Test all inputs for 16 bit and gate

--------------------------------------------------------------------------------

library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.STD\_LOGIC\_UNSIGNED.all;

use IEEE.NUMERIC\_STD.all; -- use this instead of STD\_LOGIC\_ARITH

ENTITY testAlu IS

END testAlu;

ARCHITECTURE behavior OF testAlu IS

COMPONENT nBitAlu

generic (N : integer := 4);

port ( a, b: in STD\_LOGIC\_VECTOR( N-1 downto 0 );

op: in STD\_LOGIC\_VECTOR;

y: out STD\_LOGIC\_VECTOR( N-1 downto 0 )

);

END COMPONENT;

--Inputs

signal a : STD\_LOGIC\_VECTOR(3 downto 0) := (others => '0');

signal b : STD\_LOGIC\_VECTOR(3 downto 0) := (others => '0');

signal op : STD\_LOGIC\_VECTOR(4 downto 0);

--Outputs

signal y : STD\_LOGIC\_VECTOR(3 downto 0) := (others => 'X');

BEGIN

uut: nBitAlu PORT MAP ( a => a, b => b, op => op, y => y );

stim\_proc: process

begin

-- AND

a <= "1010";

wait for 1 ns;

b <= "1011";

wait for 1 ns;

op <= "00000";

wait for 1 ns;

assert y = "1010" report "Failed A and B";

op <= "01000";

wait for 1 ns;

assert y = "0000" report "Failed A and ~B";

op <= "10000";

wait for 1 ns;

assert y = "0001" report "Failed ~A and B";

op <= "11000";

wait for 1 ns;

assert y = "0100" report "Failed ~A and ~B";

-- OR

op <= "00001";

wait for 1 ns;

assert y = "1011" report "Failed A or B";

op <= "01001";

wait for 1 ns;

assert y = "1110" report "Failed A or ~B";

op <= "10001";

wait for 1 ns;

assert y = "1111" report "Failed ~A or B";

op <= "11001";

wait for 1 ns;

assert y = "0101" report "Failed ~A or ~B";

-- SUM

op <= "00010";

wait for 1 ns;

assert y = "0101" report "Failed A + B";

op <= "01010";

wait for 1 ns;

assert y = "1111" report "Failed A - B";

op <= "10010";

wait for 1 ns;

assert y = "0001" report "Failed B - A";

-- LT / GT

op <= "01011";

wait for 1 ns;

assert y = "0001" report "Failed A < B";

op <= "10011";

wait for 1 ns;

assert y = "0000" report "Failed A > B";

op <= "10000";

wait for 1 ns;

assert y = "0001" report "Failed ~A and B";

op <= "11000";

wait for 1 ns;

assert y = "0100" report "Failed ~A and ~B";

-- LTE / GTE

op <= "01100";

wait for 1 ns;

assert y = "0000" report "Failed A >= B (<)";

b <= "1010";

wait for 1 ns;

assert y = "0001" report "Failed A >= B (==)";

op <= "10100";

wait for 1 ns;

assert y = "0001" report "Failed A <= B (==)";

b <= "1011";

wait for 1 ns;

assert y = "0001" report "Failed A <= B (<)";

-- ==

op <= "01101";

wait for 1 ns;

assert y = "0000" report "Failed A == B (!=)";

b <= "1010";

wait for 1 ns;

assert y = "0001" report "Failed A == B (==)";

b <= "1011";

wait for 1 ns;

-- !=

op <= "01110";

wait for 1 ns;

assert y = "0001" report "Failed A != B (!=)";

b <= "1010";

wait for 1 ns;

assert y = "0000" report "Failed A != ~B (==)";

b <= "1011";

wait for 1 ns;

-- XOR

op <= "00111";

wait for 1 ns;

assert y = "0001" report "Failed A xor B";

op <= "01111";

wait for 1 ns;

assert y = "1110" report "Failed A xor ~B";

op <= "10111";

wait for 1 ns;

assert y = "1110" report "Failed ~A xor B";

op <= "11111";

wait for 1 ns;

assert y = "0001" report "Failed ~A xor ~B";

wait; -- Need this line to keep the sim from looping

end process;

END;

1. Design efficiency. What is the slices / instruction for your ALU design? 49 slices / 21 instructions = 2.33
2. Swap designs with another group. Critique each other’s ALU implementation.

#### Exercise 4: Code Walkthrough ALU Presentations

On the due date for this design, groups of two will take 5 minutes (max) and present their design to the class. You will need to present the following. Don’t bring up your ALU in vivado. Instead, take screen snips to prove your test bench works, etc. Do not use more than 4 or 5 slides to summarize your design.

Minute 1: What instructions you chose and why you chose them.

Minute 2: A neat, hardware diagram of your ALU with inputs and output clearly labelled.

Minute 3: A list of the ALU operation codes (opcodes)

Minutes 4 – 5: An overview of your VHDL for the ALU and the VHDL for your test bench. Be sure and present the rough design efficiency metric (slices per instruction). **NOTE: this is a rough estimate. Some instructions could be complex (e.g. floating point) and take a significant amount of resources. Another aspect for comparing designs would be to compare clock cycles per instruction…**