P1: ALU

9/12/2023

73/100 Points

Offline Score: **73/100**

Add Comment

Anonymous Grading: **no**

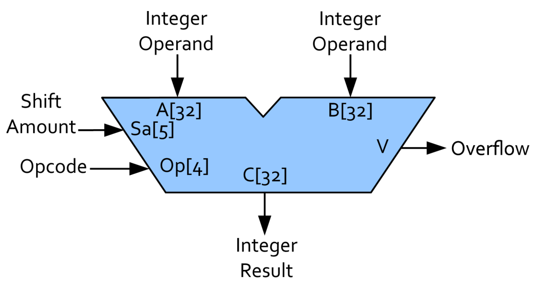
8/28/2023

Details

Scores on Canvas incorporate most but not all regrade requests. Synced to Canvas from CMS on 11/15 @ 10:00 PM

ALERT: Changes to the writeup have been made in green

Reminder: This is an individual project. Do not show your work to anyone or look at the work of anyone (in the class or online). Everything you submit must be implemented by you personally.



Overview

In this project, you'll design a RISC-V ALU. **RISC-V** is an instruction set architecture (ISA) that defines the functions a computer can carry out through assembly instructions. An **ALU**, or arithmetic and logic unit, performs many of the core computations dictated by those instructions. So in short, you'll be building a key component of a CPU. Throughout this project and the next, you'll move from small, special-purpose circuits to the full, complex CPU in all its glory!

The program you will use to create your ALU is **Logisim**, a free hardware-design and circuit-simulation tool. If you haven't yet, you can download it from the Electronic Resources (https://canvas.cornell.edu/courses/54972/pages/electronic-resources) . Logisim comes with libraries containing basic gates, memory chips, multiplexers and decoders, and other simple components.

However, for this assignment you may ONLY use the following Logisim elements:

- Anything in the **Wiring** folder *except*: constants, and anything smaller/less abstract than a gate including but not limited to the resistor, power, ground and transistor elements.
- Anything in the **Base** folder (wires, text, etc.)
- Anything in the **Gates** folder *except:* the even parity, odd parity, controlled buffer elements, and PLAs.
- Anything in the **Plexers** folder

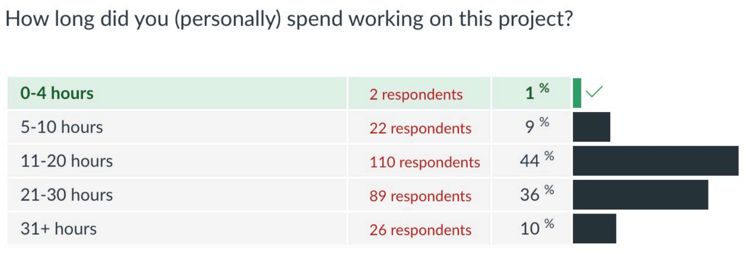
Important - Use this guideline (https://canvas.cornell.edu/courses/54972/pages/logisim-design-guidelines) for logisim FAQs and best practices for circuit design to avoid losing points!

What to Submit

All of the following should be subcircuits in a single Logisim circuit file. To get you started we've provided an ALU_template.circ (https://canvas.cornell.edu/courses/54972/files/8369402?wrap=1) (https://canvas.cornell.edu/courses/54972/files/8369402/download?download_frd=1) file with all the appropriate inputs and outputs specified.

- A single Logisim project file containing your ALU and all needed subcomponents. **Please ensure that your circuit has no external dependencies!**
    - We will use Logisim's test vector function to test your circuits. In order for it to work correctly, you must ensure that the Logisim circuits containing your solutions are named **precisely** "LeftShift32", "Shift32", "Add32", and "ALU32", and the inputs/outputs are named "A", "B", "Op", "Sa", "C", and "V".

- A design document that details the implementation of your circuit.
- A README with your name, NetID, critical path length, and gate count.
- Three text files containing your test vectors, one for each of the Shifter, Adder, and ALU circuits.
- When you are done, we will ask you to fill out the **P1 Completion Survey (https://canvas.cornell.edu/courses/54972/assignments/522610)** .
- **Heads up!** Last semester, students who filled out the survey reported the following:



How long did you (personally) spend working on this project?

0-4 hours	2 respondents	1 %	✓
5-10 hours	22 respondents	9 %	
11-20 hours	110 respondents	44 %	
21-30 hours	89 respondents	36 %	
31+ hours	26 respondents	10 %	

Circuit Design

Circuit 1: Add32

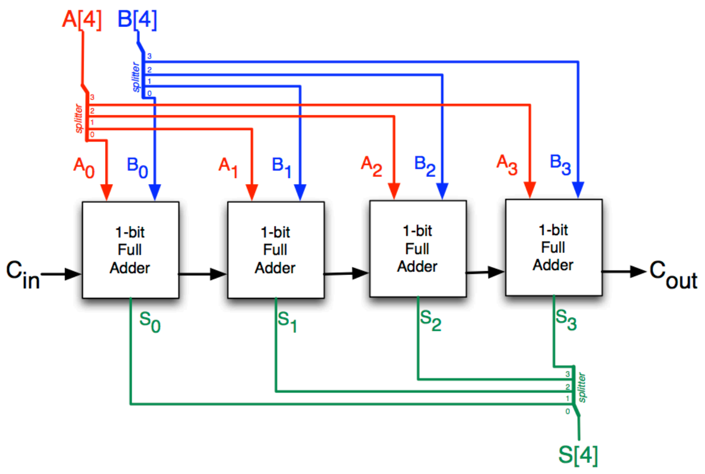
Add32:	C = A + B + Cin; V = overflow
Inputs:	A[32], B[32], Cin
Outputs:	C[32], V

The output `C` is computed by adding `A`, `B`, and `Cin`. `A`, `B`, and `C` are **signed two's complement numbers.** If overflow occurs, the output `V` should be asserted. In such cases, the output `C` should correspond to the value computed if all overflow errors are ignored. Once you complete your adder, be sure to test it!

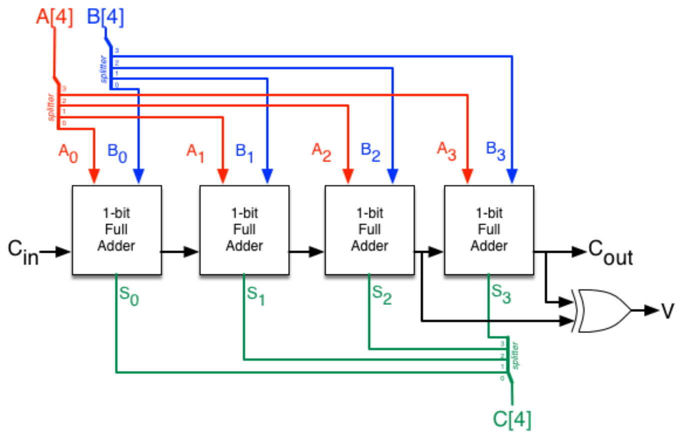
Hint: Use subcomponents to make wiring easier by building a 1-bit adder, then a 2-bit adder, then a 4-bit adder, and so on up to 32-bits.

Signed vs. Unsigned Adders

There actually isn't a huge difference between signed and unsigned adders. In fact the only difference between the two is the way that overflow is calculated. Overflow should always be calculated using the two most significant bits of a number. When building to a 32-bit adder, make sure to keep this in mind and do not bury your overflow bits within your adder. Below is a 4-bit unsigned adder, like the one you did in Lab 1...



...and here is a 4-bit two's complement signed adder.



Use this distinction when building your 32-bit signed adder. Note that the one-bit adder subcircuits in both the signed and unsigned 4-bit adders are identical (the one-bit adders are all **unsigned** adders).

Circuit 2: Shift32 (Logical)

Inputs:	B[32], Sa[5], Cin, SL
Outputs:	C[32]

Shift32:	C = (SL == 0) ? (B << Sa) | carrybits : carrybits | (B >>> Sa)
	Explanation: if SL == 0, shift left
	if SL == 1, shift right

You might remember during **Lab 2 (https://canvas.cornell.edu/courses/54972/assignments/522699)** implementing a LeftShift32. If you haven't already done so, we would recommend you merge over your LeftShift32 circuit to your ALU project as although we will be describing the Shift variant below, you will need to submit a finished alu.circ file with a subcircuit labeled LeftShift32.

The output `C` is computed by shifting `B` to the left `Sa` bits if SL == 0, and filling the vacated bits on the right with `carrybits`, which is just `Sa` copies of `Cin`. Otherwise, the output `C` is computed by shifting `B` to the right `Sa` bits if SL == 1, and filling the vacated bits on the left with `carrybits`. The shift amount `Sa` can be anything from 0 to 31, encoded as an **unsigned** integer.

When the specification denotes a pin as `B[32]`, the pin should be named "`B`" and be 32 bits wide.

Once you are confident in your circuit's correctness, think about now how you would use this circuit to handle both right shift arithmetic in addition to right shift logical and left shift. Do **not** create two different circuits to handle them. (You will not be making direct modifications in the circuit itself)

Note: You should implement this shift behavior (for both the logical and arithmetic variants of right shift) using your already made LeftShift circuit. The way to do so is a challenge we leave up to you to figure out. **WARNING:** *You will be **penalized** if you implement Shift32 without using LeftShift32.*

Hint: Shifting a value by a constant amount, either left or right, is simply a matter of adding, removing, and renaming the wires, and so requires no gates at all (this can be done with a splitter).

Circuit 3: ALU32

ALU32:	(C, V) = f_{Op}(A, B, Sa)
Inputs:	A[32], B[32], Op[4], Sa[5]
Outputs:	C[32], V

The `C` and `V` outputs are computed according to the value of the `Op` input based on the table of operations below. For the `add` and `subtract` operations, the `V` output should be set to `1` if and only if the `C` output could be incorrect due to a numerical overflow occurring during computation. The value `C` output in the presence of overflow should correspond to the value computed if all overflow errors are ignored.

Op	name	C	meaning	V
0000	and	C = A & B	*C gets the result of a bitwise **and** operation on inputs A and B*	V = 0
0001	or	C = A | B	*C gets the result of a bitwise **or** operation on inputs A and B*	V = 0
101x	shift left logical	C = B << Sa	*C gets the result of a shift left logical operation on input B*	V = 0
1101	xor	C = A ^ B	*C gets the result of a bitwise **xor** operation on inputs A and B*	V = 0
1111	nand	C = ~(A & B)	*C gets the result of a bitwise **nand** operation on inputs A and B*	V = 0
010x	shift right logical	C = B >>> Sa	*C gets the result of a shift right logical operation on input B*	V = 0
011x	shift right arithmetic	C = B >> Sa	*C gets the result of a shift right arithmetic operation on input B*	V = 0
1000	ne	C = (A != B) ? 000...0001 : 000...0000	*if A != B, C gets the value 000...0001, else 000...0000*	V = 0
1001	eq	C = (A == B) ? 000...0001 : 000...0000	*if A = B, C gets the value 000...0001, else 000...0000*	V = 0
1110	lt	C = (A $<$ 0) ? 000...0001 : 000...0000	*if A $<$ 0, C gets the value 000...0001, else 000...0000*	V = 0
1100	ge	C = (A $\geq$ 0) ? 000...0001 : 000...0000	*if A $\geq$ 0, C gets the value 000...0001, else 000...0000*	V = 0
0010	subtract	C = A - B	*C gets the result of A - B*	V = overflow
0011	add	C = A + B	*C gets the result of A + B*	V = overflow

An `x` in the opcode indicates that the circuit should not depend on the value of that bit when determining the appropriate operation. For example, your circuit should perform a shift left logical when the opcode is either `1011` or `1010`. All opcodes that are not covered in the table can have undefined behavior and will not be tested.

The expression `(test) ? 1 : 0` has a value of `1` if `test` is true, and 0 otherwise. In both cases, the upper 31 bits of the output are zero. **Note the difference between logical right shift (which fills with zero bits), and arithmetic right shift (which fills the new bits with copies of the sign bit of `B`)**. The logical and (`&`), or (`|`), xor (`^`), nor, and complement (`~`) operators are all bit-wise operations. Also note that le and gt compare A to **0**, not B.

Notes & Hints

Getting started: Design your circuits on paper before building them in Logisim. Design, build, and test an adder/subtractor unit for use in your ALU. Repeat the same steps for circuit 2: design, then build and test the left shifter circuit first. Next, design, build, and test a left/right shifting unit to be used in the ALU. Think of the left/right shifter as miniature ALUs: it will have its own opcode-like control input of your choice that selects between its different sub-operations. Then design a comparator unit that can perform the four comparison operations by processing the output of the adder/subtractor or other subcomponents. Finally, design, build, and test the complete ALU for circuit 3. The overall idea is to compute several potentially needed values of the output `C` using the pieces you have already built and then to select the appropriate one using a multiplexer.

Decoding logic: Once you have all of your subcomponents, you need to combine these so that your circuit can compute several values in parallel, but it will ultimately select only one for output. Your decoding logic can often be simplified if you note that you only need the output of a sub-component to be correct (i.e. for it to receive the correct inputs) if you know ultimately that it will be selected for output. In short, try to find the cases where you really don't care about the inputs to, or outputs from, a sub-component.

Don't duplicate components: Your ALU should use your adder and shifter as components. Your ALU should also use a **single component** of each for the entire design. As in class, your ALU should only use a single 32-bit adder component to implement both addition and subtraction. Similarly, your ALU should use **only a single** 32-bit shifter component to implement **all of the shift operations**. Do not change your original shifting and adding components, instead you can achieve full functionality by using these subcircuits only once in your full ALU. For instance, right shifting can be accomplished by transforming the inputs and outputs to your left shifter. ****IMPORTANT**You will be penalized if your final ALU circuit uses more than one adder or shifter**.

On specifications: It is important that your implementation of the three circuits described above adhere to the specification in this document. Adhering to specification is important in most all design processes, and it will also have a more immediate effect on your grade for this project. Automated grading will expect that the three circuits above (and their inputs and outputs) are named **exactly** as specified (case-sensitive!) and behave exactly as specified. Also recall that when the specification denotes a pin as `A[32]`, the pin should be named "`A`" and be 32 bits wide. The pin should not be named "`A[32]`".

On circuit design: Be sure to follow the logisim design guide (https://canvas.cornell.edu/courses/54972/pages/logisim-design-guidelines) to avoid losing points!

On bringing components together: Check out the textbook to understand how an ALU works.

Test Vectors

Extensively testing your circuit is important to ensure correctness.

While you obviously can't test every possible input, it is feasible in Logisim to test up to several thousand input tuples. You can even write a program to do so (in Python, Java, Bash, etc.) if you would like to, although we don't require it. You should strive to choose inputs strategically and include enough of them to test the *entire* functionality of your ALU. Some of your tuples should be written by hand to test corner cases (e.g. adding combinations of zero, +1, -1, max_int, min_int, etc.). Some might be generated systematically (e.g. testing every possible shift amount, and every possible `Op`), and others might be generated randomly (to catch cases you didn't think of). **Testing is an important part of this project and you will be graded on both your random and edge cases.**

For this project, you should create three ASCII text test vector files, one for each of the **LeftShifter**, **Adder**, and **ALU** circuits. **A brief comment at the beginning of each file should indicate how the tests were chosen/generated and what they are testing. Each edge case should also include an explanation of why they are an edge case. In addition, please clearly group your random and edge cases together and comment where each type begins (e.g. "#Random cases start here").** You can create a comment by starting a line with a pound sign (#).

The box below demonstrates the format of a Logisim test vector.

```
#First line labels input and output pins, labeled with [bit width] if >1
#Each line after is an individual test with each column representing a pin
#Numbers can be in decimal, binary, hexadecimal, or even octal if you want
#Logisim determines the base of your input value as follows:
#if starts with 0x or 0o: hex or octal respectively
#if length of input value == width of input/output pin: binary
#else: decimal

#Choose whichever base(s) is/are most comfortable for you.
#They will all be interpreted in binary by Logisim and you can assume this
#will always work. The purpose of this is that, for instance, binary might be
#easier for you if you are making tests for OR whereas decimal might be easier
#if you're making tests for ADD. We assume that, unless you are Prof. Bracy,
#your brain defaults to 7+6=13, not 0b111+0b110=0b1101, for instance.

#In this example, the columns are aligned neatly for readability.
#Your test vector does NOT have to be aligned like this. You can separate each
#value with a single whitespace. We won't be grading your test on its looks.

B[32]                             Sa[5]   Cin   C[32]
34                                2       0     136                               #Decimal
325948595                         15      0     -900104192                        #Negative Decimal
00000000000000000000000000000001  00100   1     00000000000000000000000000011111  #Binary
0x00000005                        0x01    0x0   0x0000000A                        #Hexadecimal
0o17777777777                     0o1     0o0   0o37777777776                     #Octal
00011111001010100010110111110001  0o10    0x0   707653888                         #Everything together!
```

Testing FAQ's

Q: How do we test our circuits?

The Cornell version of Logisim adds a special "Test Vector" feature to Logisim for automated testing of circuits. The documentation for this is accessed from within Logisim: select Help->User's Guide from the toolbar. On the left pane of the help window that appears, look for and select the item labeled "Test Vectors".

Q: To write a test case for my test vector, I need to know what the correct result for a certain operation is. How could I ever do this?

All of the ALU's operations are clearly defined arithmetic operations. The results can easily be computed by hand. Even better, implementations of these arithmetic operations are available in every major programming language.

Q: We can just use Logisim's logging feature to generate a test vector, right?

Let's get this straight. To verify the correctness of *your ALU,* you are going to log the output of *your ALU* for a few inputs, and then you are going to verify that *your ALU* gives the same output when given those same inputs? This is basically asking "does your ALU produce the same outputs as your ALU?"

The first rule of tautology club is the first rule of tautology club (http://xkcd.com/703/) .

Q: How many test cases do we need? Is x number of tests enough?

We aren't looking for a specific number of tests, but that your tests should convince you your circuit behaves as intended. That means you need to write both random and edge cases for each op code. The number of edge cases should vary depending on the op code but try to think of at least 3-4 cases that can cause unexpected behaviour (eg. overflow). Roughly 100 random cases split evenly between the op codes is definitely enough, but you are welcome to generate more for your own testing purposes.

Design Documentation

Your final submission will include a design doc explaining the following:

- Diagrams showing the components of your ALU (adder, shifter etc.) and connections between them. Explain any functionality that we might find confusing.
  - Note that this, and all diagrams, can simply be Logisim screenshots *if your circuit is neat and easy to read.* Please use the Logisim text tool or an image editor to add labels/annotations if necessary.
  - If a diagram contains many components, a couple of sentences should suffice for the description of its functionality. You definitely do not need to write paragraphs explaining how everything works.
- Explanations of your control logic. What exactly does this mux do? What effect does changing this bit have? Include a brief description of what the values of any control signal mean, and a truth table showing the value that signal takes for each possible opcode.
- Any design decisions or tradeoffs you made, if you feel they are relevant.

Note that the design doc does *not* have to be long or overly detailed, nor does it have to be in LaTeX. **Here is an example format** **(https://canvas.cornell.edu/courses/54972/files/8369046/download?wrap=1)** (**Latex source** **(https://canvas.cornell.edu/courses/54972/files/8369235/download?wrap=1)** **(https://canvas.cornell.edu/courses/54972/files/8369235/download?download_frd=1)**). You don't have to follow it.

Students often ask how detailed a design doc should be. To give you an idea, good design docs tend to be around 10 pages in length with full-sized screenshots. That being said, reaching this page number is not indicative of the quality of an individual design doc. Shorter ones can be good if filled with concise yet comprehensive descriptions and longer ones can be bad if missing key information.

README

In addition to a design document, you are required to submit a short README. The README includes the following:

- Your name and NetID
- An estimate of length of the critical path of the complete ALU
- An estimate of the number of gates required to implement the ALU (including gates needed for subcomponents)

Critical Path

In synchronous logic, the critical path is the slowest logic path in the circuit. We have assumed that the operation of the ALU completes in one clock cycle. In order to determine how long the clock cycle is, you need to figure out which path in your circuit is the longest path for the input signals to propagate through. This particular path is called the critical path. The amount of time for the input signals to propagate through the critical path is the minimum length of one clock cycle. The reciprocal of the clock period gives the maximum frequency of the input clock signal. You may express your critical path in terms of the number of gates in the path. To determine the critical path you should use the following simplifying assumptions:

- Standard AND, OR, NOR, NAND, XOR, and XNOR gates have a gate delay of one unit
- NOT gates are ignored

- Multi-input and multi-bit gates both have the same gate delay as their variants
- A mux has a gate delay of 2 regardless of size (you can derive this formula on your own by looking at how a mux is built out of basic gates!)

Gate Count

In microprocessor design, gate count refers to the number of transistor switches, or gates, that are needed to implement a design. Even with today's process technology, gate counts remain one of the most important overall factors in the end price of a chip. Designs with fewer gates will typically cost less, and for this reason, gate count remains a commonly used metric in the industry. To determine the gate count you should use the following assumptions:

- Standard AND, OR, NOR, NAND, XOR and XNOR gates count as one gate
- NOT gates are ignored
- Multi-input gates count as a single gate
- An n-bit gate counts as n gates (the multi-bit gates Logisim provides aren't actually real; they are just a convenient shorthand for using a gate for each bit)
- A mux counts as (# of data bits)*(# of inputs + 1) gates (again, you can see this by looking at what it's made of)

Tentative Point Distribution

The following is subject to change but in the past the point distribution has been:

- Circuit: 70
- Documentation: 10
- Testing: 20

Submission Checklist

Circuit

- Follows every item in **Logisim Design Guidelines (https://canvas.cornell.edu/courses/54972/pages/logisim-design-guidelines)**
- Does not have any forbidden components (e.g. constants, Logisim Adder/Shifter)
- Add32, LeftShift32, and Shift32 subcircuits should appear **only once** in the entire submission
- Any operations that can use the same circuitry (e.g. EQ and NEQ) should use the same circuitry

Note: Any circuitry that mimics the function of a forbidden or duplicated component will still be penalized.

- Does not use more components than necessary
- No overcomplicated circuitry
- No circuitry with the same functions as a Logisim component (e.g. building your own MUX)
- Add32, LeftShift32, Shift32, and ALU32 subcircuits follow specifications exactly
- Should be named exactly as shown (case sensitive)
- Should have all the pins listed, and only the pins listed, also named correctly (case sensitive, without bit widths e.g. "[32]")
- Is readable, using abstractions (subcircuits) appropriately and tunnels sparingly

Documentation

- Includes README with gate count and critical path
- Design document adequately describes subcircuits and the logic used to control them

Testing

- Includes comments that explain how tests were chosen
- Edge cases include explanations on why they are edge cases
- Covers all operations
- Covers both corner and random cases

Academic Integrity. As one of the most widely studied architectures, RISC-V has a wealth of information available on the web and in textbooks. You may consult any RISC-V documentation available to you in order to learn about the instruction set, what each instruction does, how an ALU works, etc. However, we expect your design to be entirely your own, and your submission should cite any significant sources of information you used. If you are unsure if it is okay to borrow from some other source, just ask the TAs. If you are hesitant to ask the TAs or to cite the source, then it is probably not okay. Plagiarism in any form will not be tolerated. It is also your responsibility to make sure your sources match the material we describe here (warning: the top Google hit for "RISC reference" contains several inaccuracies).

(https://canvas.cornell.edu/courses/54972/assignments/522717)

P2 (https://canvas.cornell.edu/courses/54972/assignments/522716)

(https://canvas.cornell.edu/courses/54972/assignments/522717)

‹
(https://canvas.cornell.edu/courses/54972/modules/items/2052961)

›
(https://canvas.cornell.edu/courses/54972/modules/items/2052963)