

Name(s): Ethan Little and Aaron Wang
CSE 20221 Logic Design
HW06: Basic Calculator

Introduction

The purpose of this assignment is to design a 4-bit calculator that performs all of the albaCore arithmetic and logic functions, plus some additional functions. Inside the calculator are two main components: a combinational logic component called the *arithmetic/logic unit* (ALU), and a sequential logic component called an *accumulator* consisting of a 4-bit register to store a running total. Conceptually, this calculator has some similarities to the Thales Patent¹ mechanical calculator. This mechanical calculator has a set of internal wheels that maintain the accumulated sum, displayed as digits on the front panel, and some sliders for setting the next value to be added, also displayed as digits at the top of the front panel. The new value is added to the accumulated sum when the crank is turned.



More formally, we could say that the Thales Patent calculator performs the *register-transfer* operation

$$a \leftarrow a + b$$

where a is the accumulated total and b is the input value set by the sliders whenever the crank is turned. Your calculator will be capable of performing operations of the form

$$a \leftarrow a \text{ alu_op } b$$

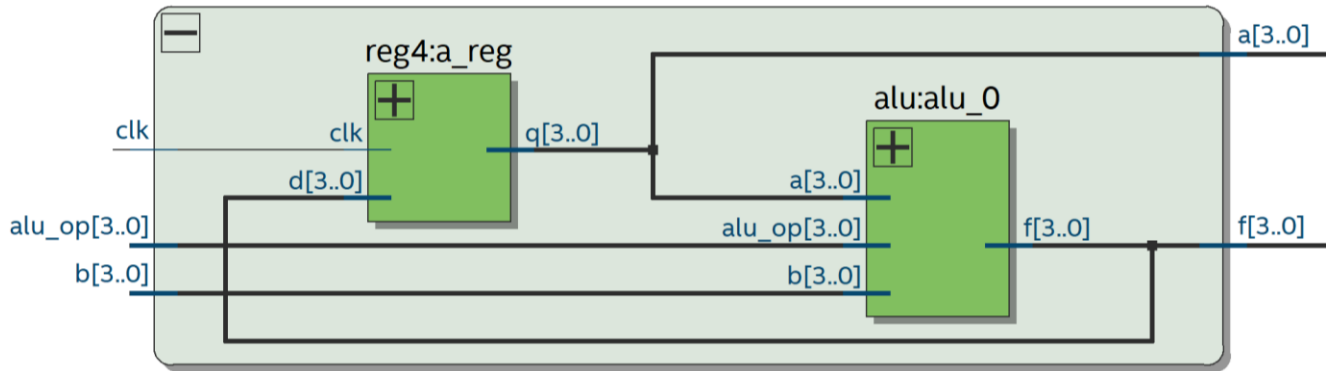
where a is the contents of the accumulator register, b is an input value set by switches in Logisim-evolution, and alu_op is one of the albaCore arithmetic/logic operations, such as add, sub, and, or, not, shift, or “bonus” operations². alu_op is also selected via switches in Logisim-evolution. Rather than triggering operations by turning a crank, however, your calculator will trigger operations on the rising edge of a clock pulse.

¹ If you are interested, here is a video showing how this machine works:

<https://www.youtube.com/watch?v=RSiMXBbVLQc>

²For better or worse, later we’ll see that albaCore is limited to a 3-bit alu_op and 8 total operations, but this could form the basis for an albaCore++.

Below is a block diagram of the **calculator**, with inputs and outputs described in the table and figure:



Important: *alu_op* and *b* have both input(s) **and** output(s) so we can see the state of each input on a Hex Digit Display. Make sure to connect all specified inputs and outputs!

Pin	Direction	Description	Input/Output
clk	input	clock	None
alu_op[3:0]	input	ALU operation	In: 4-bit dip switch Out: Hex Digit Display named HEX4
a[3:0]	output	value in accumulator register a (note how it is also fed to the ALU input internally)	Out: Hex Digit Display named HEX3
b[3:0]	input	data input	In: 4-bit dip switch Out: Hex Digit Display named HEX2
f[3:0]	output	ALU output (transferred to a on rising clock edge)	Out: Hex Digit Display named HEX0

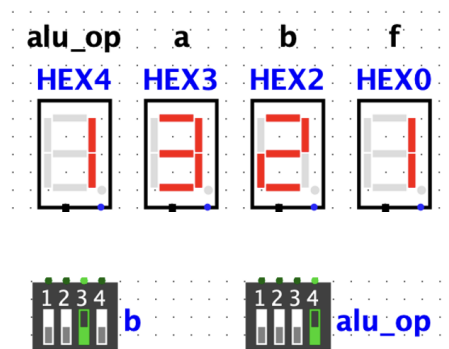


Figure 1: example showing inputs and outputs for $b = 2$, $a = 3$, operation is sub
The operations performed by the calculator are determined by *alu_op* as follows:

alu_op[3:0]	operation	
0	a + b	
1	a - b	
2	a & b	
3	a b	
4	~a	
5	a << b (logical left shift)	
6	a >> b (logical right shift)	
7	b	
8	a ? b	Choose your own adventure: ? is up to you, choose something not implemented in 1-7, 9. It can be something built into Logisim-evolution, or something (useful!) you create
9	-b	

Design and Implementation

Implement a complete design for the calculator in Logisim-evolution, following the bottom-up design practices outlined in HW05 – namely, design and test your ALU first, then the register, then the full system wired together.

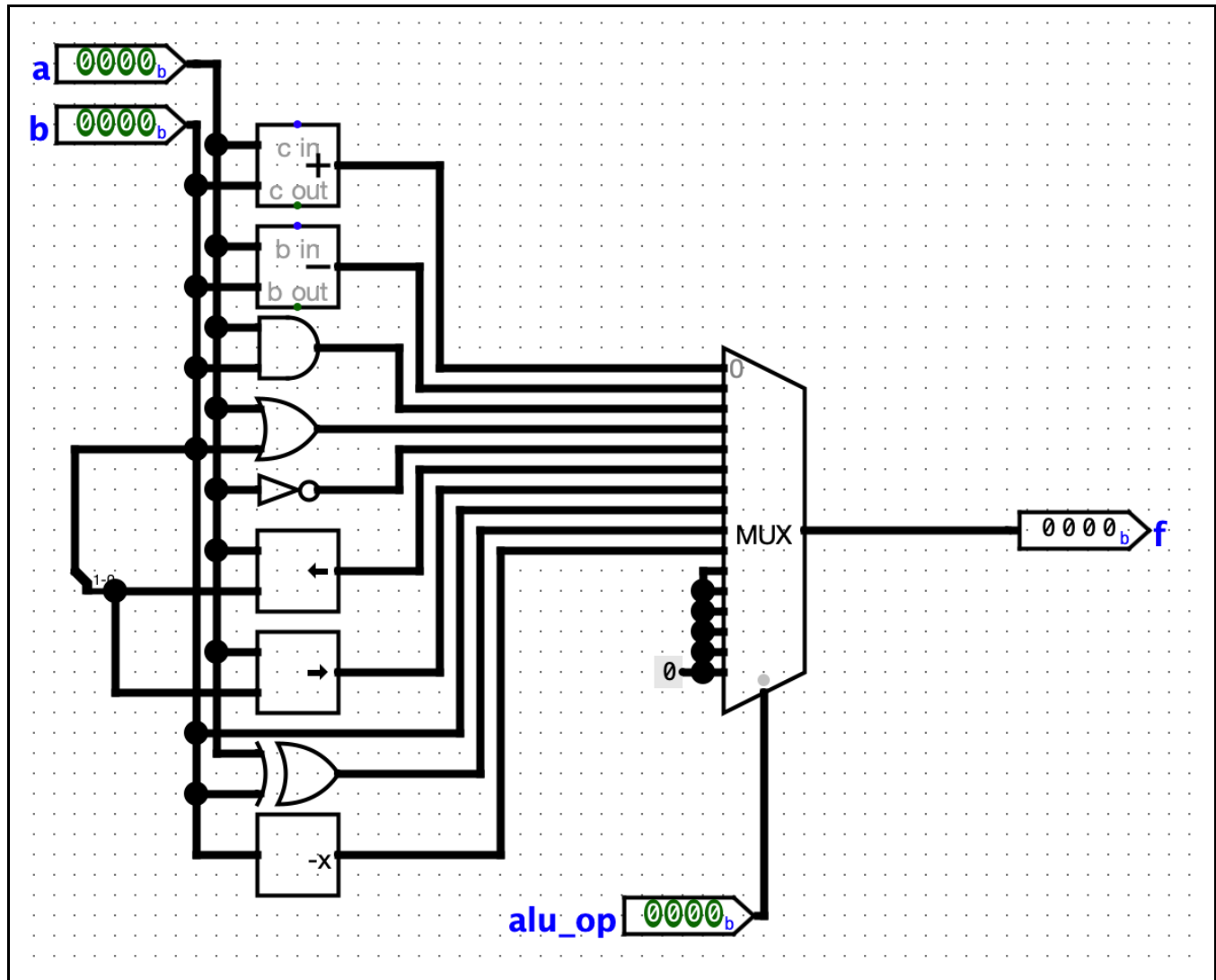
ALU

Design and test a 4-bit ALU with inputs a[3:0], b[3:0], and alu_op[3:0] and output f[3:0].

You do not need to implement each internal operation of the ALU “by hand.” Instead, build your ALU using the built-in Logisim-evolution gates (and, xor, etc.) and components (adder, subtractor, etc.).

(If you would like, you can use your addsub3 circuit from HW05 for the adder and subtractor.)

Logisim-evolution schematic for aLu



Define a test vector that tests all 10 ALU operations; only 1 test per ALU operation is required. The test case input values a, b are up to you.


Test vector for aLu

a[4]	b[4]	f[4]	alu_op[4]
1010	0101	1111	0000
1010	0101	0101	0001
1100	1010	1000	0010
1100	1010	1110	0011
1010	0000	0101	0100
1010	0001	0100	0101
1010	0001	0101	0110
000	1010	1010	0111
1100	1010	0110	1000
0000	0101	1011	1001

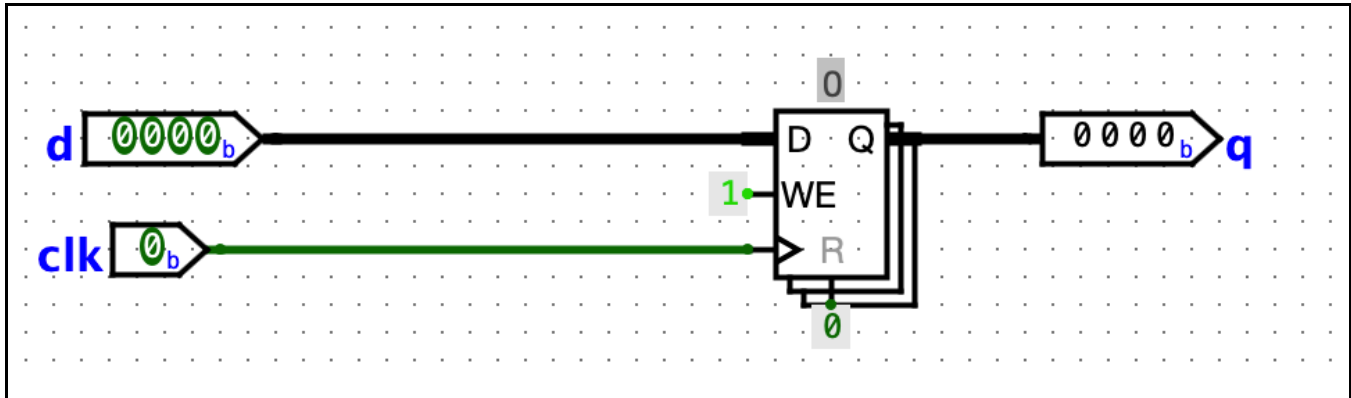
Screenshot showing test vector passing for aLu

Passed: 10 Failed: 0				
Status	a	b	f	alu_op
pass	1010	0101	1111	0000
pass	1010	0101	0101	0001
pass	1100	1010	1000	0010
pass	1100	1010	1110	0011
pass	1010	0000	0101	0100
pass	1010	0001	0100	0101
pass	1010	0001	0101	0110
pass	0000	1010	1010	0111
pass	1100	1010	0110	1000
pass	0000	0101	1011	1001

Accumulator Register




Design and test a 4-bit positive edge-triggered register with inputs clk, d[3:0], and output q[3:0]. Call your circuit reg4. You may use the built-in register in Logisim-evolution. The write enable (WE) and reset (R) of the built-in register will not be used and should not be connected to input pins; instead, connect them to an appropriate constant input to ensure the register is written on every rising clock edge, and never reset. You should *not* have a clock generator ( **Clock**) in the circuit, the clock is passed into the clk input pin.

Logisim-evolution schematic for reg4



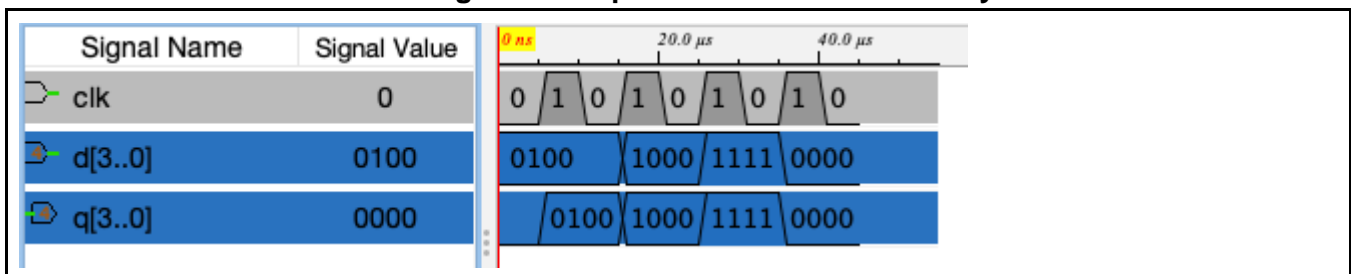
Rather than writing a test vector, test reg4 using a timing diagram. Show the register storing values 3, 26, 38, and then 27 on subsequent rising clock edges.

Here are suggested steps for how to do this efficiently (see also our notes from class on 10/10).

- Click the *Simulate* menu and then *Timing Diagram*,
- Click the *Timing diagram* tab in the window that pops up (next to *Options*),
- Click the Register signal in the *Signal Name* pane and press Delete on your keyboard to remove it (you can also do this by clicking *Add or Remove Signals*, selecting the Register entry, and clicking the left arrow),
- Toggle the play/pause button to pause the simulation ,
- Slow the simulation down to give you time to change the inputs: click the *Simulate* menu and then *Auto-tick Frequency* and select 0.5 Hz,
- Enable the clock to tick automatically: click the *Simulate* menu and then *Auto-tick Enabled* (or toggle this button: ); since clk is an input and there is no internal clock generator in reg4, a dialog will pop up asking you to select a clock; select the clk input and click OK,
- Set up your first input for d using the poke tool, and then toggle the play/pause button  to begin the simulation,
- Pause after each falling edge, change d, and unpauses to load the new value into the register,
- Finally, pause the simulation so you can take a screenshot.
- Tip: drag the red vertical line all the way to the left in the timing diagram window and use the CTRL+R (or CMD+R on a Mac) keyboard shortcut to reset the timing diagram. (Dragging the red line can prevent some graphical artifacts.)

Screenshot of timing diagram for reg4 showing the register storing values 4, 8, 15, and then 0 on subsequent rising clock edges

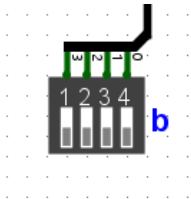
Make sure to include both the Signal Name pane and the waveform in your screenshot



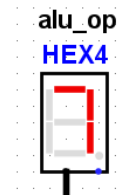
Calculator

⛔ Stop and check yourself. Did you thoroughly test the ALU and register designs? Read through the rest of the assignment to ensure you understand what you need to complete, and how the ALU and register, together, can be used to accumulate a calculated value. If you have any questions, now is the time to ask!

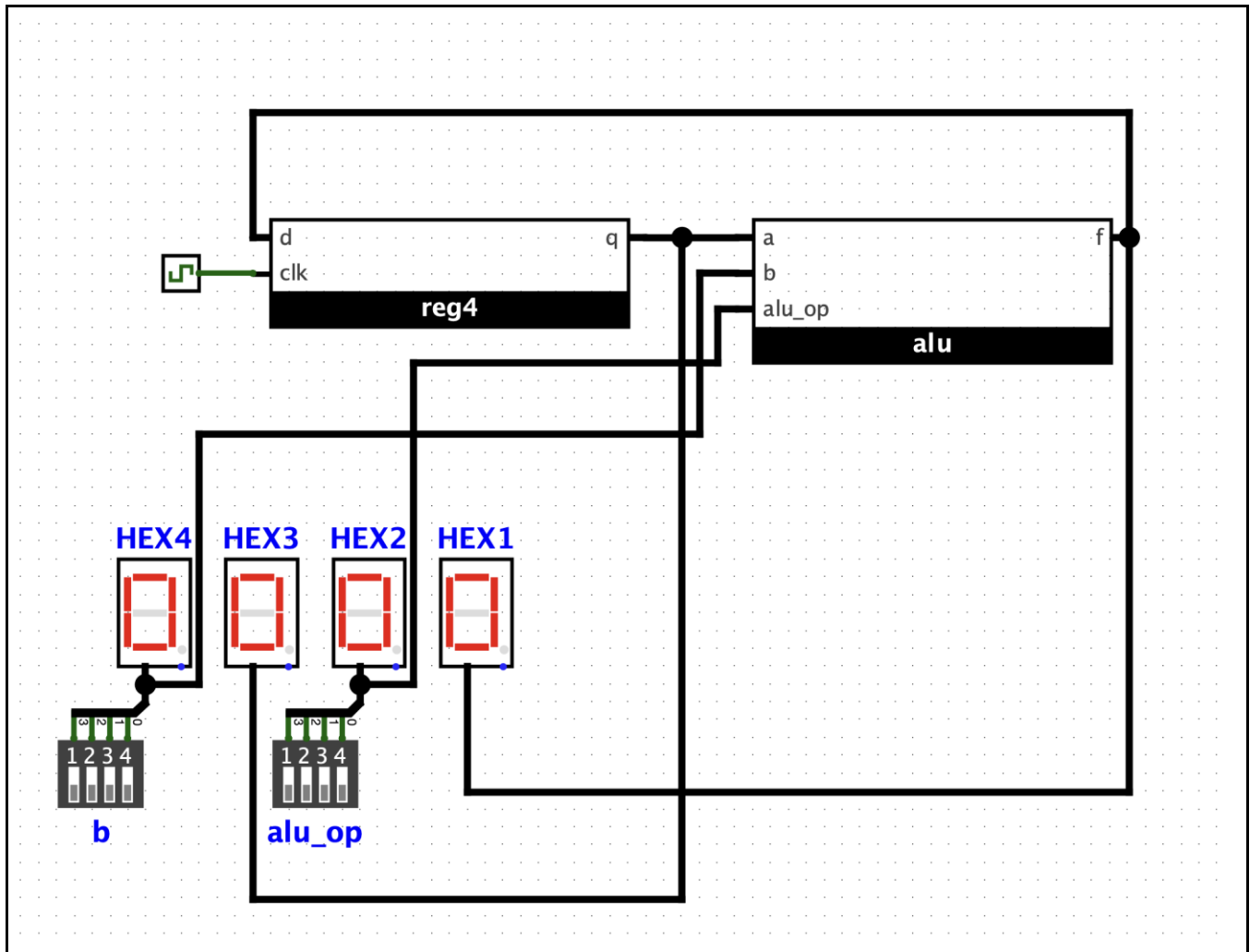
Design and test the calculator consisting of the ALU and accumulator register wired together and with pins as shown in the block diagram and tables above (see pages 2-3). Use dip switches for the `b` and `alu_op` inputs. An N-bit dip switch is numbered 1 to N from left to right, but it is natural for us to read the binary from right (least significant bit) to left (most significant bit). Wire the dip switches using splitters to account for this, as shown below with `b` as an example:



Add four **Hex Digit Display** circuits to display the outputs for `alu_op`, `a`, `b`, and `f`. Label the hex displays HEX0, etc. as in the table above. Add a text label above each to indicate what it is displaying (see the example, below). The leftmost pin on the bottom of each Hex Digit Display expects a 4-bit input (recall: a nibble is 4 bits) which will be displayed in hexadecimal, for example, when `alu_op` is 0b0111, 7 will be displayed:



Logisim-evolution schematic for calculator



Upload your complete .circ file to one group member's dropbox and note the location in the space below:

/escnfs/home/awang27/esc-courses/fa24-cse-20221.01/dropbox/HW06

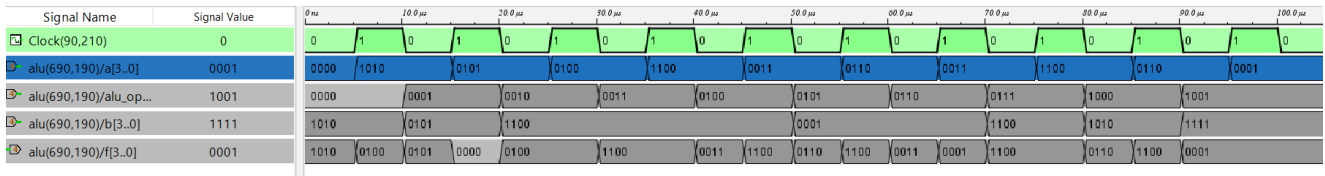
Use the Timing Diagram to test all 10 ALU operations. **The first test** should load a constant value from the b input into the accumulator register a. Tests for the remaining operations are up to you; summarize in the table below to help you make a plan for creating the timing diagram and to help the graders follow your work. Note in each case that a should get the value of f following the *next* rising clock edge.

Screenshot of timing diagram for calculator, and list of expected inputs/outputs

Expected inputs/outputs for your chosen test cases:

alu_op[3:0]	operation	a	b	f
-------------	-----------	---	---	---

0000	a+b	0000	1010	1010
0001	a-b	1010	0101	0101
0010	a & b	0101	1100	0100
0011	a b	0100	1100	1100
0100	~a	1100	1100	0011
0101	a << b	0011	0001	0110
0110	a >> b	0110	0001	0011
0111	b	0011	1100	1100
1000	a xor b	1100	1010	0110
1001	-b	0110	1111	0001



Deliverable Checklist**ALU (12 points)**

	Deliverable	Points
1a.	alu schematic	8
1b.	alu test vector	2
1c.	alu test vector passing	2

Accumulator (13 points)

	Deliverable	Points
2a.	reg4 schematic	5
2b.	reg4 timing diagram	8

Calculator (30 points)

	Deliverable	Points
3a.	calculator schematic and .circ file in dropbox, with path	15
3b.	calculator timing diagram	15