Name(s): Aaron Wang and Ethan Little
CSE 20221 Logic Design
HW05 Adder/Subtractor Design

Preamble

1. Make sure you start from the Google Docs copy of this assignment, found in Google Drive.
2. Copy this assignment to your Google Drive folder and enter your answers in the boxes provided; **each empty box should be filled in with an answer**. You can either type your answers directly or scan handwritten work as long as it is legible. For solutions that require code, use a fixed-width font (Consolas preferred) no smaller than 10 points with proper indentation. Save your solutions as a single PDF file and upload them to Gradescope.
3. See the checklist at the end of the document for a list of what to turn in, and point totals.
4. Type your name(s) at the top of this document - you are encouraged to work in groups of 2.
5. Always double-check your final PDF prior to uploading to Gradescope. In past semesters, students occasionally encountered problems copying output from a terminal to a Google Doc.
6. When copying text (**preferred**) or taking a screenshot, please use black text on a white background, or white text on a black background, otherwise it is very difficult to read.

You will implement and simulate a Logisim-evolution model for a 3-bit adder/subtractor with overflow detection. A block diagram for the adder/subtractor is shown below:

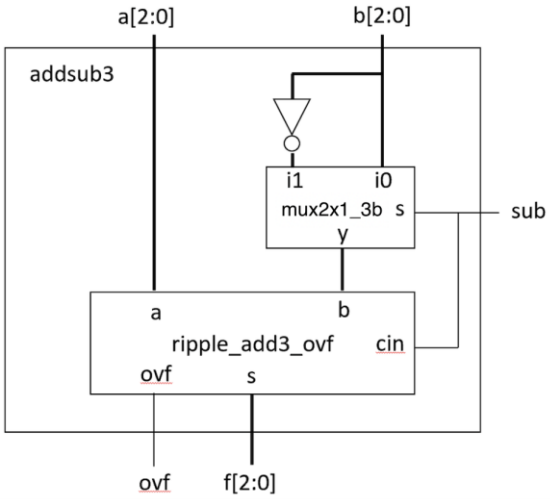


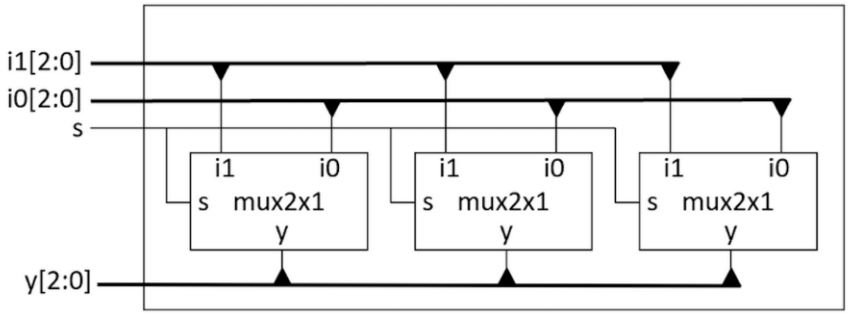
Figure 1: `addsub3` with subcircuits

The inputs and outputs of the `addsub3` circuit are as follows:

Port	***Direction***	***Description***
a[2:0]	input	first operand
b[2:0]	input	second operand
sub	input	add/subtract selector (0 = add; 1 = sub)
f[2:0]	output	sub == 0 → f = a + b sub == 1 → f = a - b
ovf	output	overflow detected (1) overflow not detected (0)

The main circuit, `addsub3`, contains these **subcircuits** (shown in Figure 1), which each contain subcircuits of their own:

- *ripple_add3_ovf:* 3-bit ripple-carry adder, composed of 3 1-bit full adders wired together, with additional circuitry connected to determine if an overflow occurred assuming `a`, `b` are interpreted in 2's complement form. Note that the overflow detection circuitry should be included in *ripple_add3_ovf*, not as a separate circuit.
- *mux2x1_3b*: 3-bit wide, 2:1 multiplexor (mux), implemented as 3, 1-bit wide 2:1 muxes wired together as shown in the diagram below. `y` is the output, `i1`, `i0` are inputs, and `s` is the select. (This diagram is *not* the Logisim-evolution schematic.)



`mux2x1_3b`

  - A subcircuit called `mux2x1` has been provided in the Homework directory on Google Drive (see the `mux2x1_lib.circ` file). You will **not implement your own** `mux2x1_3b`, since the goal here is to ensure you are familiar with using "external" circuit libraries.

Main Circuit: `addsub3`

- This should be the **last** circuit you create, it will incorporate the *ripple_add3_ovf* and *mux2x1_3b* subcircuits you created. This is called a "bottom up" approach, where we define each subcircuit and slowly build our way up to `addsub3`.

One of the main learning goals for this assignment is to use the "building blocks" we have seen in class (gates, muxes, etc.) to practice creating more complex systems in Logisim-evolution. Given this is our second Logisim-evolution assignment, suggestions for how to approach the assignment follow.
Note: no additional deliverables are on this page, but there are some hints included.

Suggested approach/order to complete this assignment:

1. See the deliverables checklist and/or scan the assignment so you know what subcircuits/circuits you are creating, how they are tested, and what to turn in.
2. Implement and *test* a 3-bit 2:1 mux, comprising 3 of the 1-bit 2:1 muxes
    a. Add gate and wires to implement the circuit, then "proofread" the circuit for correct logic, no missing/bad connections, etc.
    b. Write a test vector that tests all input combinations for the circuit
    c. Ensure your design works correctly with the Test Vector tool
3. Implement *and test* a 1-bit full adder
    a. Rather than wiring this from scratch, you are welcome to write a truth table and use the *Combinational Analysis* tool in Window → Combinational Analysis.
        i. Recall: there is a simpler expression than the tool gives for sum (using xor gates).
    b. "Proofread" the subcircuit for correct logic, no missing/bad connections, etc.
    c. Write a test vector that tests all input combinations for your subcircuit
    d. Ensure your design works correctly with the Test Vector tool
4. Implement and *test* a 3-bit ripple-carry adder, comprising 3 of your 1-bit full adders. Recommended: initially, do this without overflow detection.
    a. Repeat 2a-c for the 3-bit ripple-carry adder (again, see the notes or textbook as needed)
    b. For your test vector, you are not required to test all input combinations; use the given test cases (see table below)
5. Implement overflow detection by wiring the logic and creating another output in `ripple_add3_ovf`
    a. Think back to our discussion from L03, see page titled "Overflow: Summary" on how to test for overflow. You can do this with a *single gate*. Working through this is a good thought exercise, but the solution is well known; if you get stuck, talk to the instructor or TAs and we can guide you to the solution.
    b. Update your test vector to test the new output
6. Implement and *test* the complete 3-bit adder/subtractor (`addsub3`)
    a. Repeat 2a-c for the adder/subtractor
    b. This is a bit like saying "draw the rest of the owl", and it is probably the trickiest part of the assignment, so consider doing the following:
        i. Following the required module definition (below), start by adding *only* the `ripple_add3_ovf` module to `addsub3`
        ii. Your adder/subtractor is now just an adder, which you have already tested
            1. Ensure `a`, `b`, `f` are connected correctly
        iii. Add a `mux2x1_3b` module to your `addsub3` module
            1. Connect `b` and `sub` as shown on the first page
        iv. Now, create your test vector and ensure your design works in simulation for the given test cases

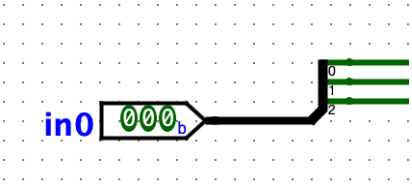
Multiplexor

Implement a 3-bit 2:1 multiplexor using the 1-bit 2:1 multiplexor subcircuit. Download the `mux2x1_lib.circ` subcircuit file to your computer and add it as a subcircuit. To add an existing subcircuit, click *Project* → *Load Library* → *Logisim-evolution Library* and navigate to select the .circ file you downloaded. You should now see the following at the bottom of your Design tab:

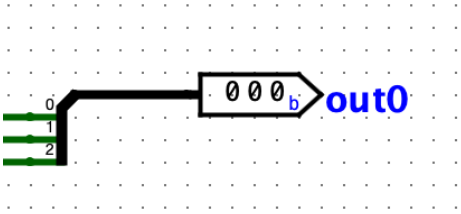
You can click `mux2x1` and then click anywhere in the schematic to include a copy of `mux2x1` in your schematic.

Now, wire your 3 2:1 multiplexors to create the 3-bit multiplexor. You will need two 3-bit input pins (name them `in0`, `in1`), and a 3-bit output pin (name it `out0` – out is reserved). This presents a dilemma: try connecting one of your input pins to any of the 2:1 multiplexors. You will see an orange line and text that says “Incompatible widths”.

To remedy this, add a splitter from the Wiring menu in the Design tab. You can modify the *fan-out* of the splitter to define how many bits the signal will be split into (3, to split to the 3 individual muxes), and modify the *Bit Width In* to define how many bits the signal source is (3, from the input pin).



You can do the same on the output side, but make it face *West* instead of the default *East*.

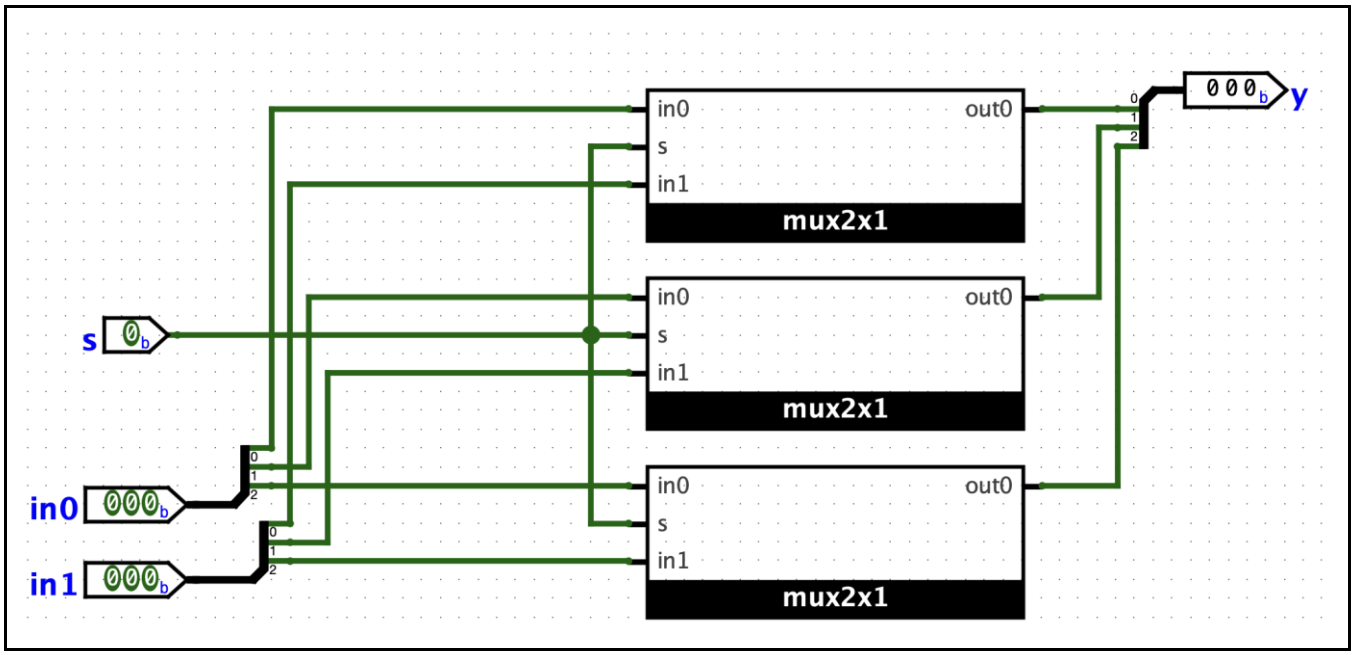


See Additional Features → Splitters in the User’s Guide for more information on how to use splitters.

Take a screenshot of your 3-bit 2:1 multiplexor schematic and paste it in the box below.

***mux2x1_3b* schematic**

(no .circ file required until `addsub3` is complete)



Testing the 3-bit 2:1 multiplexor

Note: you are not required to turn in test vector output for the 1-bit multiplexor, nor are you required to submit an individual .circ file.

Pick a representative set of inputs to ensure your multiplexor can select either input, and passes the correct input to the output. Create a test vector to test these cases – follow the format in the test vector from HW04, which is like a truth table, though not all combinations of inputs need be included. You can specify multi-bit inputs and outputs by changing the column header to NAME[N] where N is the number of bits, e.g., x[4] is a 4-bit input named x. For readability, use tabs (or multiples spaces) to space columns.

***mux2x1_3b* Test Vector (copy the text file contents here)**

```
in1[3] in0[3] s   y[3]
000    111    1   000
010    101    1   010
100    001    1   100
111    000    1   111
111    000    0   000
001    100    0   100
010    101    0   101
000    111    0   111
```

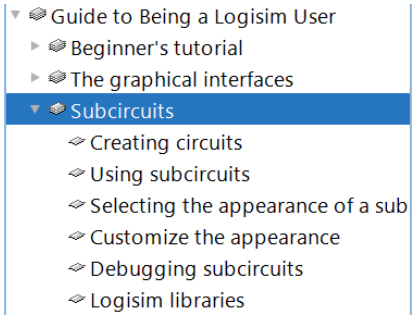
mux2x1_3b Test Vector window showing all tests passed

Passed: 8 Failed: 0

Status	in1	in0	s	y
pass	000	111	1	000
pass	010	101	1	010
pass	100	001	1	100
pass	111	000	1	111
pass	111	000	0	000
pass	001	100	0	100
pass	010	101	0	101
pass	000	111	0	111

1-bit Full Adder

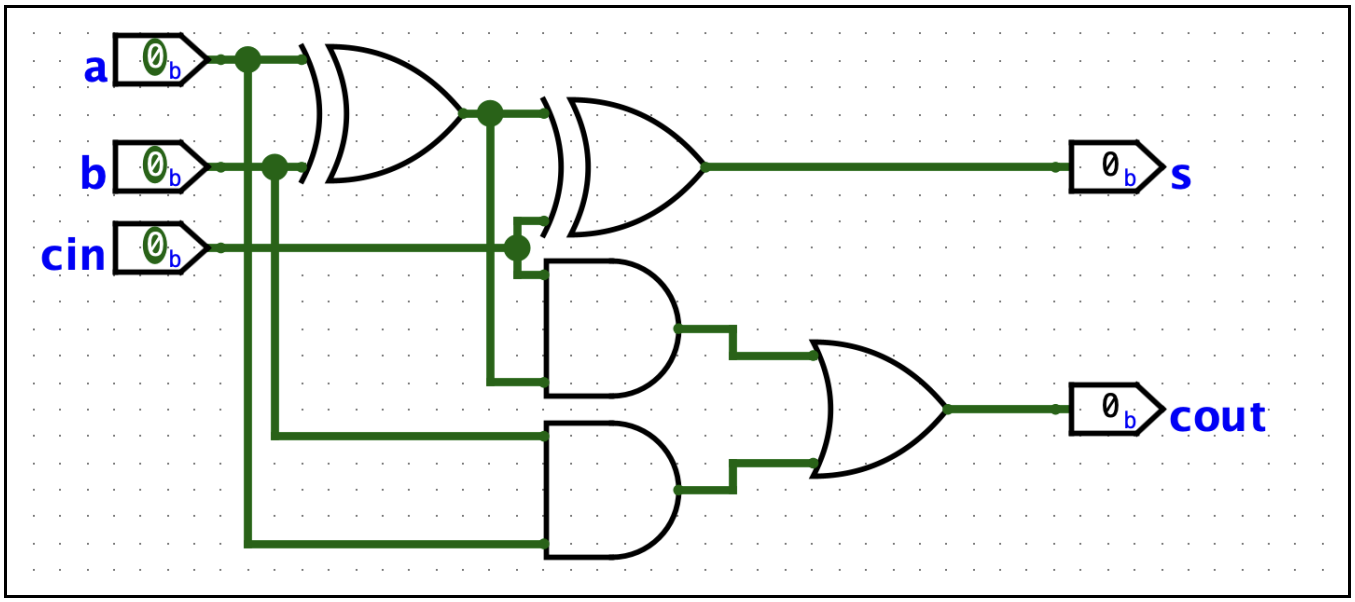
Implement a 1-bit full adder as a new circuit, which you will later use as a subcircuit. In short, click the + in the Design tab, give the circuit a name (`add1`), and define the logic in the schematic. You can then use it as a subcircuit as we did previously. See the "Subcircuits" section in the Logisim-evolution documentation for additional details and an example for how to create a new circuit and use the subcircuit.



Take a screenshot of your 1-bit adder schematic and paste it in the box below.

Note: you are not required to turn in test vector output for the 1-bit adder, nor are you required to submit an individual .circ file.

add1 schematic



3-bit Ripple Carry Adder

First, complete the table below to define a set of test cases and determine expected results in binary and *signed* decimal. The signed decimal should indicate what the sum is. For example, 101 + 101 (-3 + -3) gives the *sum* 010, which is interpreted as 2. This is a good way to double-check your understanding of overflow is correct (we got negative + negative = positive, which indicates overflow).

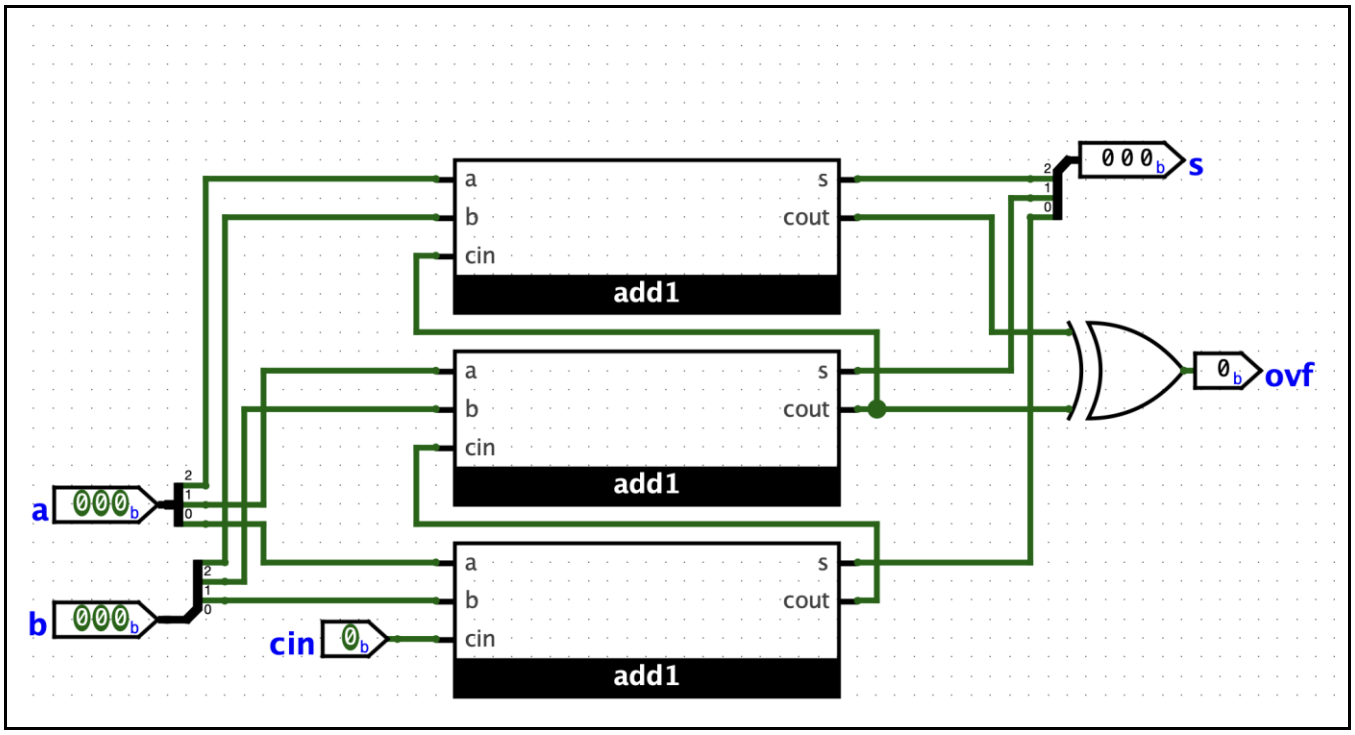
Table of test cases:

a bin	a dec	b bin	b dec	sum bin	sum dec	overflow?
010	2	111	-1	001	1	no
011	3	010	2	101	-3	yes
001	1	010	2	011	3	no
100	-4	110	-2	010	2	yes
100	-4	100	-4	000	0	yes
010	2	110	-2	000	0	no

Next, implement a 3-bit ripple carry adder as a new circuit that incorporates 3 copies of your 1-bit full adder subcircuit. Take a screenshot of your 3-bit ripple carry adder schematic and paste it in the box below.

ripple_add3_ovf schematic

(no .circ file required until `addsub3` is complete)



Test Vector for ripple-carry adder with overflow detection

Test your circuit with all 6 cases from the table above, where the carry-in to the least significant bit is 0 for all cases. For example, the first row is testing 010 + 111 → 2 + (-1). Again, see the example from HW04 for how to write a test vector.

ripple_add3_ovf Test Vector (copy the file contents here)

```
a[3]  b[3]  cin   s[3]  ovf
010   111   0     001   0
011   010   0     101   1
001   010   0     011   0
100   110   0     010   1
100   100   0     000   1
010   110   0     000   0
```

ripple_add3_ovf Test Vector window showing all tests passed

Passed: 6 Failed: 0

Status	a	b	cin	s	ovf
pass	010	111	0	001	0
pass	011	010	0	101	1
pass	001	010	0	011	0
pass	100	110	0	010	1
pass	100	100	0	000	1
pass	010	110	0	000	0

Complete Adder/Subtractor

Implement the complete adder/subtractor, `addsub3`. For consistency, define your circuit's inputs and outputs as on the first page of this document: `a`, `b` are 3-bit inputs (the operands); `sub` is a single bit input (0 to select addition; 1 to select subtraction); `f` is a 3-bit output (the `sum`); `ovf` is a 1-bit output (1 if overflow is detected, 0 otherwise).

Test Vector for complete adder/subtractor

Test it with the same cases as done earlier with the ripple-carry adder with overflow detection, except that the addition of negative numbers from before are now represented as subtraction operations. For example, the first row should now test 010 - 001 → 2 - 1. Write a test vector to test 5 of the 6 test cases – do not test the test case in row 5 (-4 + -4 cannot be expressed as -4 - 4, since there is no 4 in a 3-bit 2's complement representation).

addsub3 Test Vector (copy the file contents here)

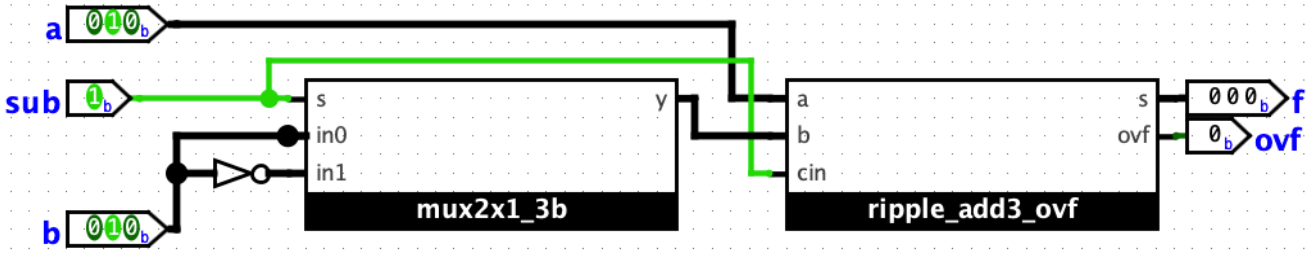
```
a[3]  b[3]  sub   f[3]  ovf
010   001   1     001   0
011   010   0     101   1
001   010   0     011   0
100   010   1     010   1
010   010   1     000   0
```

addsub3 Test Vector window showing all tests passed

Status	a	b	sub	f	ovf
pass	010	001	1	001	0
pass	011	010	0	101	1
pass	001	010	0	011	0
pass	100	010	1	010	1
pass	010	010	1	000	0

Take a screenshot of your 3-bit ripple carry adder schematic and paste it in the box below.

addsub3 schematic



Finally, copy the completed .circ file for addsub3 to one group member's dropbox and note the path in the space below. This circuit should contain *all* subcircuits above. Make sure all subcircuits you created are in this one .circ project, **not** references to subcircuits in other .circ projects.

dropbox full path

```
/escnfs/home/awang27/esc-courses/fa24-cse-20221.01/dropbox/HW05
```

Deliverable Checklist

Multiplexor (10 points)

	Deliverable	Points
1a.	mux2x1_3b schematic	5
1b.	mux2x1_3b Test Vector (file contents)	3
1c.	mux2x1_3b Test Vector (screenshot showing passing)	2

1-bit Full Adder (5 points)

	Deliverable	Points
2a.	add1 Schematic	5

3-bit Ripple Carry Adder (15 points)

	Deliverable	Points
3a.	Filled table showing test cases	2
3b.	ripple_add3_ovf schematic	8
3c.	ripple_add3_ovf Test Vector (file contents)	3
3d.	ripple_add3_ovf Test Vector (screenshot showing passing)	2

Complete Adder/Subtractor, addsub3 (20 points)

	Deliverable	Points
4a.	addsub3 Test Vector (file contents)	5
4b.	addsub3 Test Vector (screenshot showing passing)	5
4c.	addsub3 schematic and path in dropbox with **complete** .circ file	10