lab 1
Build an 8-bit ALU with Logisim

M. Briday

October 7, 2025

1 Objective

The objective of the lab is to build an ALU from basic logic elements, using the *logisim* tool.

This lab is split in 3 parts:

- build a very basic 1-bit full adder, using only logic gates;

- from this adder, construct a 8-bit basic adder;

- build an ALU that can add, substract and perform some basic logical operations, and outputs a status of the operation.

A sequential 8x8 hardware multiplier is eventually implemented using the adder and shift registers.

2 Logisim Evolution

2.1 Installation of Logisim Evolution

The original version of Logisim is no longer maintained, and several *forks* have appeared. We will use *Logisim Evolution*[1]. A pre-compiled version for GNU/Linux, MacOS or Windows can be found on `https://github.com/logisim-evolution/logisim-evolution/releases`.

2.2 Basic operation

Logisim is a logic system simulator. The basic help for the different elements is well documented in `Help -> User's guide`.

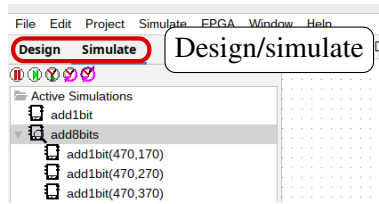To launch a simulation, you must first click on `'simulate'`, see Figure 5.

---

[1]`https://github.com/logisim-evolution/logisim-evolution`

Figure 1: Choice between design and simulation

3 Adder

3.1 1-bit adder

A full adder has in addition to its 2 inputs a and b, the management of the carry (both input and output), as in Fig. 2.
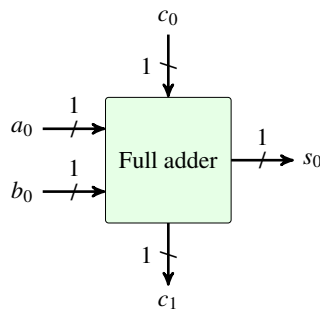

Figure 2: 1-bit full adder. c_i is the carry

 ▷ give the truth table of the 1-bit full adder;

 ▷ give the 2 equations of the outputs s_0 and c_1;

With logisim, draw the schematic of these 2 signals. You may use the operators in the *gates* section. The input/outputs should be defined in `wiring->Pin`.

You can define the external appearance of your model (figure 3).

Then, you can rename your current circuit, and add another one (figure 4).

 ▷ test your application with different inputs: in *simulate* mode, use the *hand* icon. You just need to click on the ports, as in Figure 5

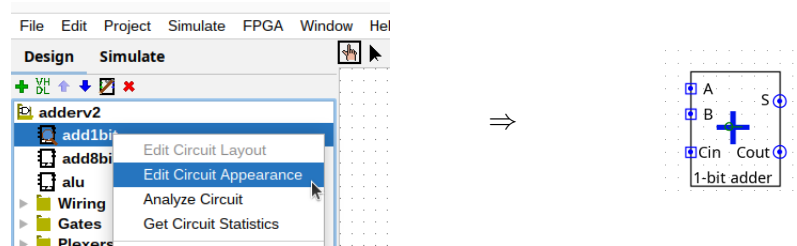Note that your test should consider the 1-bit adder as a black box!
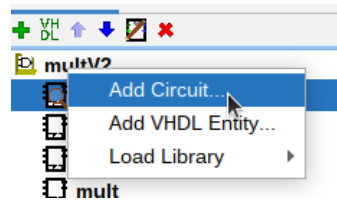
Figure 3: Define component appearance
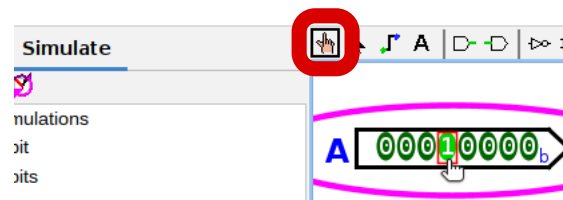

Figure 4: Define new components

Figure 5: Simulation: you have to click on the *Hand* icon before.

3.2   8-bit Ripple-Carry adder

Using this 1-bit adder component, we now create a 8-bit adder component, as in Fig. 6: the output carry c_{i+1} of the 1-bit adder A_i is connected to the input carry of A_{i+1}.

We will have to use a 8-bits input pin (`Wiring->Pin`) and a splitter to change a 8-bit bus into 8 1-bits wires (`Wiring->Splitter`).

Notes:

- instead of long wires, you can use a *tunnel* (`Wiring->Tunnel`) to make a wire connection between 2 tunnels with the same name.

- you can use 8-bits input/output pins

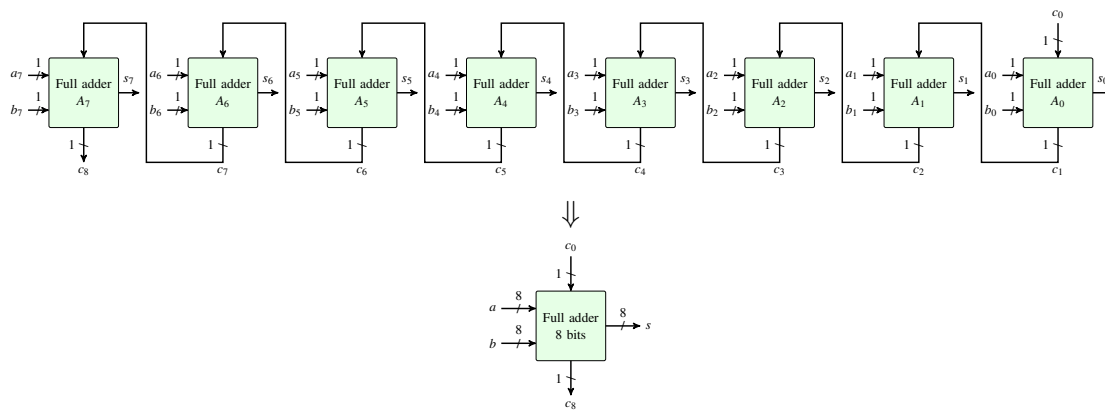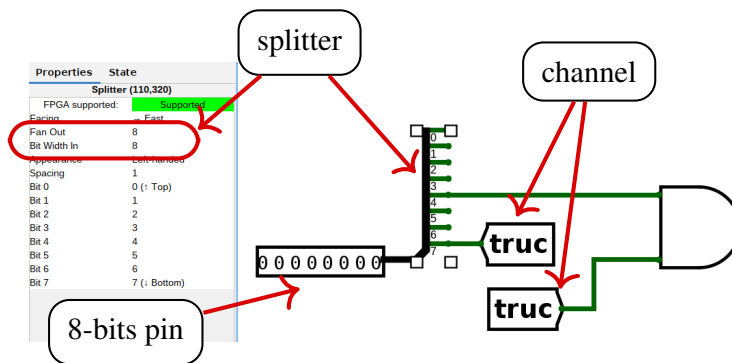- the *splitter* can be used to extract bits from a bus

3

Figure 6: 8-bit full adder. c_i is the carry

▷ implement the 8-bits adder;

▷ test your design

4 ALU

Most of the ALU is the adder, and we add some logic functions (AND, OR, ...). All these functions are performed simultaneously, and a multiplexer before the output selects the appropriate operation.

Notes:

- the substraction may use the adder and use the properties of the two's complement : -b=~b + 1.

- you can use multiplexers in `Plexers`

- you can use `Arithmetic->Shifter`. Note that the shift value should be a 3-bits bus only (that allows a 2^3-bits shift). You will need a splitter to isolate the 3 low bits of B.
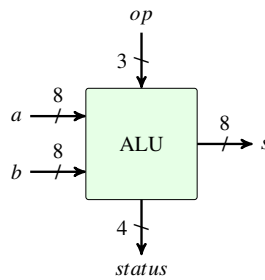
Figure 7: 8-bit ALU. op (input) and status (output) are defined in the text.

- you **cannot** use adders and other arithmetic operators, except one instance of the full adder you have just designed.

op is a 3-bits input signal that selects the operation to perform:

0. addition

1. substraction

2. and

3. or

4. not: ~a

5. xor

6. shift left[2] : a << b

7. shift right: a >> b

status is a 4-bits output signal that gives the common status of an ALU:

0. *N*egative

1. *Z*ero

2. o*V*erflow: the result is not correct with signed numbers, *i.e.* the sum of 2 positive numbers is a negative number.

3. *C*arry: the result is not correct with unsigned numbers: this is the 9th bit of an operation.

▷ Implement the different operations, with the output status.

4.1 Basic extension: arithmetic operator

The *arithmetic shift right* perform a shift with a *sign extension*: If the value is negative, then the bits added at left are 1 instead of 0.

▷ Add Arithmetic shift right operation (instead of not)

5 One step further… a multiplier!

The multiplier works in the same way as in the decimal way learnt at school, with the calculation of partial products:

---

[2]Most modern processor implements a multi-bit shift, but the Microchip AVR for instance does not!

```
   456          1001  (B)
X 123        X 0101  (A)
------       -------
  1368         1001
  912.         0000.
456..          1001..
------         0000...
56088          -------
               0101101
```

We implement here an hardware multiplier (Fig. 8) as a sequential operator which adds either B or 0 to the product (a partial product) and performs shifts at each step. If A and B require n bits, then the product is a $2n$ bits value.
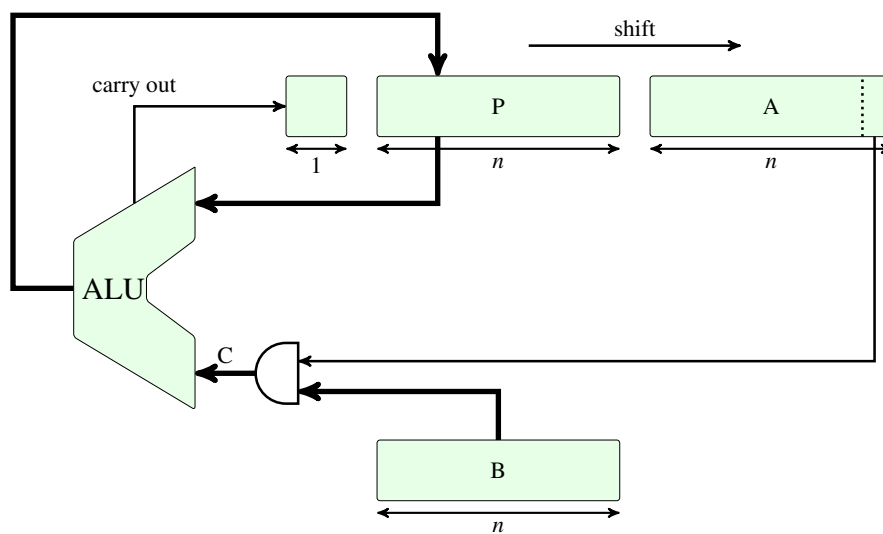

Figure 8: Multiplier

- P is set initially to 0

- for each step (n steps):

  - if the lsb of A is 1, then B is added to P. The sum is placed back in P.

  - P and A are shifted right, with the carry-out of the sum being moved into the MSB of P, the LSB of P moved to the MSB of A, and the LSB of A shifted out.

After n steps, the products appears in register P and A, with A holding the low order bits.

The circuit is sequential and needs to perform shifts and adds one after the other. We can use for the registers A, P, B and the carry-out a shift register as defined in Fig. 10:

- If load is set when there is a rising edge on clk, then S is copied into P

- If shift is set when there is a rising edge on clk, then the content of P is shifted: new input bit on the left blue wire (MSB), output bit on the red wire at right (LSB)

- At each time, the value of the register can be read (signal P).

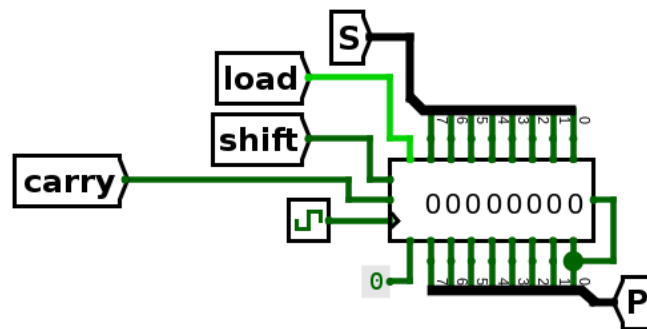- The value of the register can be updated during simulation using the 'hand' tool and a click on a value.

Figure 9: Shift register

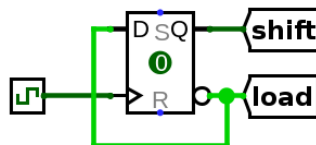To generate the clocks load and shift, we use a simple frequency divider with a flip-flop:

Figure 10: clocks shift and load generation using a flip-flop