COMSM1302 Lab Sheet 2

This lab sheet covers content from week 2 lectures, focusing on boolean arithmetic and relevant circuit implementation. On successful completion of the lab, students should be able to:

1. Create subcircuits within Logisim and use them as components within other circuit designs.
2. Assemble circuits that can handle multiple bit inputs and outputs, using splitters.
3. Design circuits that can carry out arithmetic operations, in particular addition and subtraction.
4. Use multiplexers and demultiplexers within a circuit design to make computational decisions based on relevant inputs.

You'll need the Logisim software to implement gates and circuits, and may need to install Java first if running Logisim on your own computer. Details on how to download and run Logisim can be found on the unit page and if you need help with using Logisim, try referring to the Logisim user guide.

You will also need to borrow a NAND board to try physical implementations, which you can find details of on the unit page. If you get stuck using NAND boards, try watching the NAND board explanation or have a look through the GitHub documentation.

Half adder

Assemble a **1-bit half adder** circuit on Logisim, using any gates with two or fewer inputs[1]. The desired behaviour is defined by the given truth table.

A	B	C_out	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Then, build a **4-bit incrementer** on Logisim, using your half adder as a subcircuit[2]. The circuit should take input from a 4-bit input pin, add 1 (from a constant 1-bit pin), and output the result through a 4-bit output pin. A *splitter* can be used to separate a multi-bit signal into 1-bit signals, or vice versa - these can be found in the wiring library, along with constant pins.

Hint: You'll need more than one copy of your half adder subcircuit...

Full adder

Assemble a **1-bit full adder** circuit on Logisim, using any gates with two or fewer inputs. The desired behaviour is defined by the given truth table.

C_in	A	B	C_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Then, build a **4-bit adder** on Logisim, using your full adder as a subcircuit. The circuit should take inputs from two 4-bit input pins, add them, and output the result through a 4-bit output pin. It should also include a 1-bit input pin and a 1-bit output pin, to account for the carry.

Hint: You'll need more than one copy of your full adder subcircuit...

Challenge: Can you adapt your 4-bit adder, so it can subtract as well as add?

Hint: You'll need a multiplexer and an understanding of 2's complement!

[1] Remember: NOT gates only have one input, but you should use two input versions of any other logic gates.
[2] To use your half adder as a subcircuit, single-click on the circuit name then click where you'd like to place it.

NAND adders

Assemble circuits on Logisim only using NAND gates with two inputs - one should implement a **1-bit half adder** and one should implement a **1-bit full adder**.

Validate your solutions by assembling each of your designs on a NAND board, being careful to sensibly plan where you will connect wires first. As an *optional task*, try chaining your full adder NAND board with another student's - each NAND board will need to be connected to the same power supply, using a USB splitter.

Plexers

For each of the following definitions, find an equivalent Boolean expression using $\wedge$, $\vee$, and $\neg$. You may assume that any signals mentioned are 1-bit.

Hint: Truth tables might be a useful place to start...

- A **2-to-1 multiplexer** is a decision-making circuit that allows one of two input signals to pass through to a single output, with the choice controlled by a selection input.
- A **1-to-2 demultiplexer** is a decision-making circuit that allows a single input signal to be sent to one of two outputs, with the choice controlled by a selection input.

Validate each expression by assembling a logically equivalent circuit on Logisim, using logic gates with two or fewer inputs, and checking every possible combination of input values produces the output value you'd expect.

Challenge: Can you assemble similar circuits on Logisim, one that implements a **4-to-1 multiplexer** and one that implements a **1-to-4 demultiplexer**?

Arithmetic Logic Unit

Build a **Hack ALU** on Logisim, that includes 6 pre-built multiplexers, with the following 1-bit control bits:

1. **zx**: if zx == 0 then x = x, else x = 0
2. **nx**: if nx == 0 then x = x, else x = ¬x
3. **zy**: if zy == 0 then y = y, else y = 0
4. **ny**: if ny == 0 then y = y, else y = ¬y
5. **f**: if f == 0 then out = x&y, else out = x+y
6. **no**: if no == 0 then out = out, else out = ¬out

The signals x, y, and *out* should be 4-bits and you should use your 4-bit adder as a subcircuit within this design. Test your design is correct using the given table, which lists **some** combinations of values of the control bits.

Then, connect 1-bit outputs to your ALU design that report the following comparisons:

1. **zr**: if out == 0 then zr = 1, else zr = 0
2. **ng**: if out < 0 then ng = 1, else ng = 0

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Challenge: Can you explain why each series of calculations produces the output defined in this table?

Number representations

Convert each of the following numbers into **binary, octal, decimal, and hexadecimal** - excluding each number's original base. You may assume that 2's complement representation is used, where appropriate.

- 10101101_2
- 01110101_2
- 11011100_2
- 705_8
- 312_8
- 051_8
- 123_{10}
- -3_{10}
- -84_{10}
- 92_{16}
- $7f_{16}$
- da_{16}

Challenge: For each binary number given or calculated, determine its decimal value if it were interpreted using signed magnitude representation, and again if it were interpreted using 1's complement representation.

Extra challenge: For each binary number given or calculated, determine its decimal value if it were interpreted as a floating-point number. You may assume 1 sign bit, 2 exponent bits, and 5 mantissa bits are used, where any numbers with less than 8 bits are padded with 0s and any numbers with more than 8 bits ignore the overflow.

Hint: Remember that it's stored using a biased exponent (the bias for a 2-bit exponent is 1) and normalised mantissa!