Computer Systems and -architecture

Project 3: ALU

1 Ba INF 2020-2021

Brent van Bladel
brent.vanbladel@uantwerpen.be

Don't hesitate to contact the teaching assistant of this course. You can reach him in room M.G.305 or by e-mail.

Time Schedule

Projects are solved in pairs of two students. Projects build on each other, to converge into a unified whole at the end of the semester. During the semester, you will be evaluated three times. At these evaluation moments, you will present your solution of the past projects by giving a demo and answering some questions. You will immediately receive feedback, which you can use to improve your solution for the following evaluations.

For every project, you submit a small report of the project you made by filling in `verslag.html` completely. A report typically consists of 500 words and a number of drawings/screenshots. Put all your files in one tgz archive, as explained on the course's website, and submit your report to the exercises on Blackboard.

- Report deadline: **November 1, 2020, 23u55**
- Evaluation and feedback: **November 2 - November 8, 2020**

Project

Read section B.5 of Appendix B. You can only use the following Logisim libraries for this assignment: Base, Wiring, Gates, Plexers, Input/Output.

1. Build an arithmetic logic unit (ALU) for 16-bit two's complement data words. Use your 16-bit carry lookahead adder from the previous project. Use the Logisim `ALU_GroupXX.circ` file provided on the course page. Rename the file so that 'XX' is your group number. Open the file in Logisim. Do not change the 'main' circuit. Import your adder from the previous exercise by choosing from the menu 'Project' - 'Load Library' - 'Logisim Library...'. Then select your logisim file that includes the adder. Your adder will be loaded in an extra library, below 'Wiring', 'Gates', 'Plexers', etc. You can now reuse your adder. You will have to complete the '16-bit ALU' circuit in this assignment. You can only make use of the Logisim libraries 'Wiring', 'Gates' and 'Plexers' (not e.g., 'Arithmetic'). Implement the operations below, giving each operation a 4-bit binary code. Your ALU will execute the right operation according to a 4-bit operation input.

Your ALU should be able to perform the operations listed below. Make sure to test everything, including the different possible overflow cases!

- **generate 0** (Name: zero; ALU operation: `0000`).
  Example:

result	`0000000000000000`

- **NOT** (Name: not; ALU operation: `0001`).
  Example:

a	`0000000001001010`
result	`1111111110110101`

- **AND** (Name: and; ALU operation: `0010`).
  Example:

a	`0000000001001010`
b	`1111111111010101`
result	`0000000001000000`

- **OR** (Name: or; ALU operation: `0011`).
  Example:

a	`0000000001001010`
b	`1111111110010101`
result	`1111111111011111`

- **numeric addition (two's complement)** (Name: add; ALU operation: `0100`).
  Example:

a	`0000000001001010` (74)
b	`1111111110010101` (-107)
result	`1111111111011111` (-33)

  Mind overflow!

- **numeric subtraction (two's complement)** (Name: sub; ALU operation: `0101`).
  Example:

a	`0000000001001010` (74)
b	`0000000001101010` (106)
result	`1111111111100000` (-32)

  Mind overflow!

- **less than (two's complement)** (Name: lt; ALU operation: `0110`). Results in 1 if $a < b$, 0 if a $\geq$ b.
  Example:

a	`0000000001001010` (74)
b	`1111111110010101` (-107)
result	`0000000000000000` (false)

  Make sure that this operation can never produce an overflow error!

- **greater than (two's complement)** (Name: gt; ALU operation: `0111`). Results in 1 if $a > b$, 0 if a $\leq$ b.
  Example:

a	`0000000001001010` (74)
b	`1111111110010101` (-107)
result	`0000000000000001` (true)

  Make sure that this operation can never produce an overflow error!

- **equals** (Name: eq; ALU operation: `1000`). Results in 1 if $a = b$, 0 if a $\neq$ b.
  Example:

a	`0000000001001010` (74)
b	`1111111110010101` (-107)
result	`0000000000000000` (false)

Make sure that this operation can never produce an overflow error!

- **not equals** (Name: neq; ALU operation: `1001`). Results in 1 if $a \neq b$, 0 if $a = b$.
  Example:

a	`0000000001001010` (74)
b	`1111111110010101` (-107)
result	`0000000000000001` (true)

  Make sure that this operation can never produce an overflow error!
- **numeric inverse (two's complement)** (Name: inv; ALU operation: `1010`).
  Example:

a	`0000000001001010` (74)
result	`1111111110110110` (-74)

  Mind overflow!
- **shift left logical (two's complement)** (Name: sll; ALU operation: `1011`).
  Example:

a	`0000000010101010`
result	`0000000101010100`

  Make sure that this operation can never produce an overflow error!
- **shift right logical (two's complement)** (Name: srl; ALU operation: `1100`).
  Example:

a	`0000000010101010`
result	`0000000001010101`

  Make sure that this operation can never produce an overflow error!
- **shift left arithmetic (two's complement)** (Name: sla; ALU operation: `1101`). This implements "times two".
  Example:

a	`0000000000101010` (42)
result	`0000000001010100` (84)

  Mind overflow!
- **shift right arithmetic (two's complement)** (Name: sra; ALU operation: `1110`). This implements "divide by two" (integer division - test behaviour in Python).
  Example:

a	`0000000000101010` (42)
result	`0000000000010101` (21)

  Make sure that this operation can never produce an overflow error!
- **no operation** (Name: noop; ALU operation: `1111`).
  Example:

a	`0000000001001010`
result	`0000000001001010`

2. Design a 16-bit ALU that has the following interface:
   - 4-bit input: operation code
   - 16-bit input: operand a
   - 16-bit input: operand b
   - 16-bit output: result of the ALU calculation
   - 1-bit output: true in case of error/overflow

3. Run the test files for your ALU. Do this *during* the development of your ALU, not afterwards! A test file is given for each operation; run it on your circuit using the program `Test.py`. You need to install Python (`http://python.org/`) to run `Test.py`. Download `Test.py`, `tests.zip` (from the course page) and `logisim-generic-2.7.1.jar` (`http://sourceforge.net/projects/circuit/files/2.7.x/2.7.1/`) and save in the same folder as your adapted `ALU_GroupXX.circ` project. The program takes a file containing ALU tests as input, and a `ALU_GroupXX.circ` logisim file. It runs all ALU tests and reports test failures. For this assignment you will have to do the following:

   - Test your ALU using the provided set of test files.
     - Every test file contains 1 test per line. Example of a simple test:
       `add 1 2 3`
       It should be read as follows: we want to test the 'add'-operation, with operand '1' and '2' (in decimal notation), and the expected outcome of the ALU should be '3' (also in decimal notation). This expected outcome is generally called the "oracle", as it predicts the outcome of the test.

     - You see that for each test, you have to provide the operation you want to test (you can choose between zero, or, and, add, sub, lt, gt, eq, neq, not, inv, sla, sra, sll, srl, noop), values for operands a and b, and an oracle for each test. This oracle will be compared to the actual outcome of your ALU for this operation and with inputs a and b in `ALU_GroupXX.circ`. If the operation only uses the first operand (e.g., inv), you will still have to provide two operands, but the second one should be ignored by your ALU. So another valid test would be:
       `inv 1 0 -1`
       Where the second operand of value '0' will be ignored. This will test whether the numeric inverse of 1 is -1.

     - You can also provide binary values for your tests, so if you want to test the or-operation, you can write the following for example:
       `or 000010101010 000001010010 000011111010`

     - You can also test for overflow by adding a '1' to your line of code. For example, this would be a valid test:
       `add 20000 20000 0 1`
       Adding 20000 to 20000 would indeed generate an overflow for our 16-bit ALU as 40000 cannot be represented in a 2's complement 16-bit notation. This test has an added '1' at the end of the line, denoting that this test *should* generate an overflow. In this case, the test program will not compare results (therefore the result is simply '0' here). However, if your ALU does not generate an overflow, this test will fail!

     - Bear in mind that you can create test cases that are wrong, e.g.:
       `gt 100 101 1`
       This would be wrong, as 100 is not greater than 101, so the oracle must be 0 instead of 1. This test would produce a failure for a correct circuit, so double-check your test cases! Instead, a correct test would be:
       `gt 100 101 0`

   - All files must be in the same directory. The program must be executed from the

console as follows:
`python Test.py -a -i op_test.txt -c ALU_GroupXX.circ`
with `op_test.txt` as the file containing your ALU tests and `ALU_GroupXX.circ` as your logisim file (change XX to your group number). You can use the `-h` option to show the usage of the script. Try to execute the command before starting to implement the circuit.

- As you see, some lines are outputted to the console, ending with a line denoting how many tests were executed (depending on how many test lines there are in the test file) and how many of them failed or produced an error.
- If not successful, tests can be 'errors' or 'failures'. An error means that some of the resulting signals were 'Error' signals or 'don't care' signals ('E' or 'x', or a red/blue signal line in logisim). A failure means that either the expected result did not match what you have specified in your test, or the expected error value did not match. If you have failure or error tests, there will be some information about this failure/error in the output.

- If the script does not work as expected (e.g., error messages seem to be wrong), let me know as soon as possible! This script is meant to aid you in your projects, and should not slow you down!

4. To prepare for the next lab session, read sections B.7, B.8 and B.10 of Appendix B.