Computer Systems and -architecture

Project 3

1 Ba INF 2014-2015

Bart Meyers
bart.meyers@uantwerpen.be

Don't hesitate to contact the teaching assistant of this course. You can reach him in room M.G.3.17 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 a tgz archive, as explained on the course's website, and submit your report to the exercises on Blackboard.

- Report deadline: **November, 12 2014, 23u55**
- Evaluation and feedback: **November, 14 2014**

Project

Read section C.5 of Appendix C. 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 12-bit two's complement data words. To do this, create a circuit that implements a 1-bit ALU. Combine them to obtain a 12-bit ALU. 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 '12-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** (0000).
  Always returns:

result	000000000000

- **NOT** (0001).
  Example:

a	010010101010
result	101101010101

- **AND** (0010).
  Example:

a	010010101010
b	100101010010
result	000000000010

- **OR** (0011).
  Example:

a	010010101010
b	100101010010
result	110111111010

- **numeric subtraction (two's complement)** (0100).
  Example:

a	010010101010 (1194)
b	111101010010 (-174)
result	010101011000 (1368)

  Mind overflow!

- **numeric addition (two's complement)** (0101). Ripple carry addition suffices.
  Example:

a	010010101010 (1194)
b	111101010010 (-174)
result	001111111100 (1020)

  Mind overflow!

- **shift left** (0110).
  Example:

a	010010101010
result	100101010100

- **shift right** (0111).
  Example:

a	010010101010
result	001001010101

- **numeric inverse (two's complement)** (1000).
  Example:

a	010010101010 (1194)
result	101101010110 (-1194)

  Mind overflow!

- **equals** (1001). Results in 1 if $a = b$, 0 if a $\neq$ b.
  Example:

a	`010010101010` (1194)
b	`111101010010` (-174)
result	`000000000000` (false)

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

- **less than (two's complement)** (`1010`). Results in 1 if $a < b$, 0 if a $\geq$ b.
  Example:

a	`010010101010` (1194)
b	`111101010010` (-174)
result	`000000000000` (false)

  Make sure that this operation can never produce an overflow error!
- **greater than (two's complement)** (`1011`). Results in 1 if $a > b$, 0 if a $\leq$ b.
  Example:

a	`010010101010` (1194)
b	`111101010010` (-174)
result	`000000000001` (true)

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

2. Design a 12-bit ALU that has the following interface:
   - 4-bit input: operation code
   - 12-bit input: operand a
   - 12-bit input: operand b
   - 12-bit output: result of the ALU calculation
   - 1-bit output: true if the result equals zero
   - 1-bit output: true in case of error/overflow
3. Create and run a test file for your ALU. You will do this by creating a file with tests, and running it on your circuit using the program `Test.py`. You need to install Python (`http://python.org/`) to run `Test.py`. Download `Test.py`, `Test_GroupXX.txt` (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 (you already have created `ALU_GroupXX.circ` in the previous assignment). 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:
   - Create your ALU test file by adding lines to `Test_GroupXX.txt`.
     - It already contains 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, not, and, or, sub, add, sl, sr, inv, eq, lt, gt), 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., not, inv), you will still have to provide two operands, but the second one will be ignored

(by your ALU). So another valid test (you can add it as a new line to the test file) 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 000000110011 000011001100 000011111111`

- You can also test for overflow by adding a '1' to your line of code. For example, this would be a valid test:
`add 2000 2000 0 1`
Adding 2000 to 2000 would indeed generate an overflow for our 12-bit ALU as 4000 cannot be represented in a 2's complement 12-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 1000 1001 1`
This would be wrong, as 1000 is not greater than 1001, 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 1000 1001 0`

- Your goal is to add significant tests to `Test_GroupXX.txt`. Significant tests are tests that also explore the borderline cases, dealing with e.g., overflow. Write a lot of tests!

- All files must be in the same directory. The program must be executed from the console as follows:
`python Test.py -a Test_GroupXX.txt ALU_GroupXX.circ`
with `Test_GroupXX.txt` as the file containing your ALU tests and `ALU_GroupXX.circ` as your logisim file (change XX to your group number). Try to execute the command before starting to implement the circuit. You should see the following message:

```
all done: Test_GroupXX.txt
starting tests...

testing Test_GroupXX.test --> Test_GroupXX.report
-- Test on line 1 error
add 1 2 3
Operation 0101 ('add') with operands 0000 0000 0001 (1)
and 0000 0000 0010 (2), result is xxxx xxxx xxxx, error code is x

1 tests done, 1 errors, 0 failures
```

  - 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 you have added to your file) and how many of them failed or produced an error (you should have

0 here).

- 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.
- On Windows 8, there can be a permission problem when trying to execute the program. If you experience this, try the following:
  (a) start the register editor (press Windows button, then type "regedit" followed by enter);
  (b) browse to path "`HKEY_LOCAL_MACHINE/SOFTWARE/JavaSoft`";
  (c) right click on the JavaSoft folder, select permissions;
  (d) select the "Users" group (or create if it does not exist), and allow "Full Control" and "Read" permissions.

- 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 C.7, C.8 and C.10 of Appendix C.