Computer Systems and -architecture

Project 6: Full Datapath

1 Ba INF 2024-2025

Kasper Engelen
kasper.engelen@uantwerpen.be

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` or `zip` archive, as explained on the course's website, and submit your report to the exercises on Blackboard. Links to external files (e.g., dropbox, onedrive) are **not accepted**!

- Report deadline: **Monday December 16, 2024, 22u00**
- Evaluation and feedback: **Friday December 20, 2024**

Project

Read sections 4.1, 4.2, 4.3 and 4.4 of Chapter 4. You can use all Logisim libraries for this assignment.

1. In the previous assignment, we used the ALU operations as instructions and added two additional instructions (`lw` and `sw`). Next to these instructions, in this assignment we also support **branch and jump instructions**.

   We introduce a number of new instructions for `jump` and `branch`. Because you should be able to branch, you will have to connect your **program counter** to your datapath so that it can jump to a given address instead of just the next instruction.

   Implement the instructions described in the table below ("imm" stands for "immediate", "uns" stands for "unsigned" and "sig" stands for "signed, two's complement"). You already have implemented the R-type instructions and the lw/sw instructions in the previous assignment.

2. Once done, your datapath can correctly execute a program written in machine language, as the behaviour of arithmetic, branching and memory operations is now fully implemented! You can use the script `Test_2425_zit1.py` as follows (note the `-f` flag to denote the simulation of a full datapath):

Binary (16-bits)	Name	Assembly	Semantics
<opcode> <Rd> <Rs> <Rt> <Func>			
000 ddd 000 000 0000	zero	zero rd	rd = 0
001 ddd sss ttt 0000	add	add rd rs rt	rd = rs + rt
001 ddd sss ttt 0001	sub	sub rd rs rt	rd = rs - rt
001 ddd sss ttt 0010	and	and rd rs rt	rd = rs & rt (bitwise)
001 ddd sss ttt 0011	or	or rd rs rt	rd = rs | rt (bitwise)
001 ddd sss ttt 0100	lt	lt rd rs rt	rd = (rs < rt)
001 ddd sss ttt 0101	gt	gt rd rs rt	rd = (rs > rt)
001 ddd sss ttt 0110	eq	eq rd rs rt	rd = (rs == rt)
001 ddd sss ttt 0111	neq	neq rd rs rt	rd = (rs != rt)
010 ddd sss 000 0000	not	not rd rs	rd = !rs (bitwise)
010 ddd sss 000 0001	inv	inv rd rs	rd = -rs
010 ddd sss 000 0010	sll	sll rd rs	rd = (rs << 1)
010 ddd sss 000 0011	srl	srl rd rs	rd = (rs >> 1)
010 ddd sss 000 0100	sla	sla rd rs	rd = rs * 2
010 ddd sss 000 0101	sra	sra rd rs	rd = rs // 2 (integer division)
011 ddd iii iii iii 0	ldi	ldi rd imm (signed)	rd = imm
011 ddd iii iii iii 1	lui	lui rd imm (unsigned)	rd = imm << 3
100 ddd iii iii iii 0	addi	addi rd imm (unsigned)	rd = rd + imm
100 ddd iii iii iii 1	subi	subi rd imm (unsigned)	rd = rd - imm

101 ddd iii iii iii 0	brnz	brnz rd imm (signed)	if rd != 0 then
			pc = pc + 1 + imm
			else
			else pc = pc + 1

101 ddd iii iii iii 1	jr	jr rd imm (unsigned)	pc = rd + imm
110 iii iii iii iii 0	j	j addr (unsigned)	pc = addr
110 iii iii iii iii 1	jal	jal addr (unsigned)	r7 = pc + 1; pc = addr
111 ddd sss iii iii 0	lw	lw rd rs imm (unsigned)	rd = MEM[rs + imm]
111 ddd sss iii iii 1	sw	sw rd rs imm (unsigned)	MEM[rs + imm] = rd

```
python Test_2425_zit1.py -f -t <test-file> -c <circ-file>
```

You can use labels for branching and jumping in your tests. When testing the full datapath, you can only perform checks at the end of the program. (This is because of branching: it would not make sense to check a register value in the middle of a loop, as it can have a different value in a different iteration of the loop.)