Computer Systems and -architecture

Project 6: Full Datapath

1 Ba INF 2022-2023

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: **December 20, 2022, 22u00**
- Evaluation and feedback: **December 23, 2022**

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 memory and immediate instructions. Next to these instructions, in this assignment we also support branch and jump instructions.

   We introduce a number of new instructions, including 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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	name	instruction	description
000			0000				rd			000			000			zero[1]	`zero rd`	$rd := 0
001			0001				rd			rs			000			not[1]	`not rd rs`	$rd := !$rs
001			1010				rd			rs			000			inv[1]	`inv rd rs`	$rd := -$rs
001			1011				rd			rs			000			sll[1]	`sll rd rs`	$rd := $rs << 2
001			1100				rd			rs			000			srl[1]	`srl rd rs`	$rd := $rs >> 2
001			1101				rd			rs			000			sla[1]	`sla rd rs`	$rd := $rs * 2
001			1110				rd			rs			000			sra[1,2]	`sra rd rs`	$rd := $rs / 2
001			0100				rd			rs			000			inc[1]	`inc rd rs`	$rd := $rs + 1
001			0101				rd			rs			000			dec[1]	`dec rd rs`	$rd := $rs - 1
001			1111				rd			rs			000			cp[1]	`cp rd rs`	$rd := $rs
010			0010				rd			rs			rt			and[1]	`and rd rs rt`	$rd := $rs & $rt
010			0011				rd			rs			rt			or[1]	`or rd rs rt`	$rd := $rs | $rt
010			0100				rd			rs			rt			add[1]	`add rd rs rt`	$rd := $rs + $rt
010			0101				rd			rs			rt			sub[1]	`sub rd rs rt`	$rd := $rs - $rt
010			0110				rd			rs			rt			lt[1]	`lt rd rs rt`	$rd := $rs < $rt ? 1 : 0
010			0111				rd			rs			rt			gt[1]	`gt rd rs rt`	$rd := $rs > $rt ? 1 : 0
010			1000				rd			rs			rt			eq[1]	`eq rd rs rt`	$rd := $rs = $rt ? 1 : 0
010			1001				rd			rs			rt			neq[1]	`neq rd rs rt`	$rd := $rs != $rt ? 1 : 0
011			0	rd			rs			signed imm						lw	`lw rd rs imm`	$rd := MEM[$rs+imm]
011			1	rd			rs			signed imm						sw	`sw rd rs imm`	MEM[$rs+imm] := $rd
100			00		rd		unsigned imm									ori	`ori rd imm`	$rd := $rd | imm
100			01		rd		unsigned imm									lui	`lui rd imm`	$rd := imm << 8
100			10		rd		unsigned imm									addi	`addi rd imm`	$rd := $rd + imm
100			11		rd		unsigned imm									subi	`subi rd imm`	$rd := $rd - imm
101			0	rd			rs			immediate (signed)						beq	`beq rd rs imm`	$rd == $rs ? $pc := $pc + 1 + imm
101			1	rd			rs			immediate (signed)						blt	`blt rd rs imm`	$rd < $rs ? $pc := $pc + 1 + imm
110			00		rd		immediate (signed)									jr	`jr rd imm`	$pc := $rd + imm
111			0	target address												j	`j imm`	$pc := addr
111			1	target address												jal[1]	`jal imm`	$r7 := $pc + 1; $pc := addr

[1] Register r7 will be reserved for the return address of the `jal` instruction.

- In order to get all control lines right, you will have to add a **Control Unit** circuit to your datapath.
  - Input is the instruction (16 bits).
  - Outputs are the ALU OP-code as well as all control lines for i.e. the program counter, instruction and data memory, multiplexers and the register file. Choose your control lines wisely: this can make the implementation a lot easier!

  More information on the implementation of a control unit can be found in Section 4.4 of *Computer organization and design.*
- Similarly you can create an **Immediate** circuit (this is different from the book's datapath):
  - Input is the instruction (16 bits).
  - Output is the immediate value (16 bits), depending on the instruction this will be a 6, 8, or 12-bit value that is unsigned/sign extended/shifted to 16 bits.
- 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.py` as follows (note the `-f` flag to denote the simulation of a full datapath:

  ```
  python Test.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.)
- To prepare for the next lab session, read section 4.9 of Chapter 4.