Computer Systems and -architecture

Project 5: 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 2, 2022, 22u00**
- Evaluation and feedback: **December 6, 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. Build a circuit that implements a **16-bit program counter (PC)**. Use the Logisim `SD_GroupXX.circ` file provided on the course page. Rename the file so that 'XX' is your group number. Use your 16-bit register. By default, the PC is increased each clock cycle, and the next instruction is read from memory. In case of a relative branch, the PC is increased, and then the branch value is added as a 2's complement value (e.g., if the PC has value 10 and the next cycle there is a branch of value 5, then the next PC value is 16, not 15). In case of an absolute branch (or jump) the PC is directly set to the branch value. You should have the following inputs and outputs:

name	in/out	width	meaning
branch relative?	I	1 bits	branch to relative value?
branch absolute?	I	1 bits	branch to absolute value? (cannot be 1 if branch relative? is 1)
branch value	I	16 bits	the value that is used in case of a branch
C	I	1 bit	clock input
reset	I	1 bit	if set, the PC is reset to 0
instruction address	O	16 bits	the address of the instruction in the instruction memory

2. Implement a partial datapath of 16-bit instructions and 16-bit data words and addresses by using your register file, a *data* RAM element (16-bit addresses, 16-bit words), your program counter with *instruction* RAM element (16-bit addresses, 16-bit words), and your own ALU. Implement your datapath in the "main" circuit in `SD_GroupXX.circ`. You will have to modify your register file first so that it has an output for every register, to connect to the outputs in the "main" circuit (this has to be done for debugging purposes).

   - The datapath must be able to perform so-called register operations. These are the operations you implemented in your ALU. This time, operands are read from, and the result is stored into registers. The relevant registers are selected by specifying the rs, rt and rd index inputs in your register file. The operation code (op-code) is the same as the ALU code. The registers are used as follows:

     `$rd := $rs` *`operation`* `$rt`

     For unary operations (i.e., not, inv, sla, sra, inc, dec), the registers are used as follows (rt is unused):

     `$rd :=` *`operation`* `$rs`

     For the zero instruction, the registers are used as follows:

     `$rd := 0`

     For the copy instruction, the registers are used as follows:

     `$rd := $rs`

     The 16-bit instructions for the unary register operations are formatted as follows:

     - `15-13` : 001 (instruction code)
     - `12-9` : operation code
     - `8-6` : rd
     - `5-3` : rs
     - `2-0` : 000 (unused)

     Example: To invert the value of register 6, and put the result in register 3, the following instruction is loaded:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	1	1	1	1	0	0	0	0
unary reg op			*op code*				*rd*			*rs*			*unused*		

     The 16-bit instructions for the binary register operations are formatted as follows:

     - `15-13` : 010 (instruction code)
     - `12-9` : operation code
     - `8-6` : rd
     - `5-3` : rs
     - `2-0` : rt

     Example: To add the values of register 1 and register 2, and put the result in register 5, the following instruction is loaded:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	0
binary reg op			*op code*				*rd*			*rs*			*rt*		

- The datapath must be able to perform the load word (lw – reading from *data* RAM), and store word (sw – writing to *data* RAM) operations. These are memory instructions, and similarly to the MIPS lw/sw instructions, a constant can be used to denote an offset. The meaning of these instructions is as follows:

  `lw: $rd := MEM[$rs + offset]`

  `sw: MEM[$rs + offset] := $rd`

  Load word loads the contents of memory address $rs + offset into $rd. Store word stores the register value $rd into memory address $rs + offset. The offset is often used for loading an array of values from memory.

  The 16-bit instructions for the memory operations are formatted as follows:

  - `15-13` : 011 (instruction code)
  - `12` : operation code
  - `11-9` : rd
  - `8-6` : rs
  - `5-0` : signed immediate

  Example: To store the value of register 3 in memory, 4 address spaces beyond the address stored in register 2:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	0	1	0	0	0	0	1	0	0
memory op			*op code*	*rd*			*rs*			*signed immediate*					

- The datapath must be able to perform immediate operations. The meaning of these instructions is as follows:

  The ori (*"or immediate"*) performs the or operation with the unsigned 8-bit immediate value and the value of register rd, and then places the result back in register rd. `ori: $rd := $rd | immediate`

  The lui (*"load upper immediate"*) shifts the unsigned 8-bit immediate value eight times to the left, and then places the result in register rd. `lui: $rd := immediate << 8`

  The addi (*"add immediate"*) performs the add operation with the unsigned 8-bit immediate value and the value of register rd, and then places the result back in register rd. `addi: $rd := $rd + immediate`

  The subi (*"sub immediate"*) performs the sub operation with the unsigned 8-bit immediate value and the value of register rd, and then places the result back in register rd. `subi: $rd := $rd - immediate`

  The 16-bit instructions for the immediate operations are formatted as follows:

  - `15-13` : 100 (instruction code)
  - `12-11` : operation code
  - `10-8` : rd
  - `7-0` : signed immediate

  Example: To add 8 to the value of register 3:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0
immediate op			*op code*		*rd*			*signed immediate*							

All instructions you need to implement, their name, assembler instruction and description are collected in the following table:

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

[1] R-type instruction.

[2] Integer division.

- You can try out your datapath by editing your RAM-elements. You can do this by right-clicking them and selecting edit contents or save/load image.

3. Run the test files for your Datapath. Do this *during* the development of your Datapath, not afterwards! A test file is given for each type of instruction; run it on your circuit using the program `Test.py`. You need to install Python (`http://python.org/download/releases/2.7.3/`) 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 `SD_GroupXX.circ` project. The program takes a file containing small assembler programs that are datapath tests as input, and a `SD_GroupXX.circ` logisim file. It runs all datapath tests and reports test errors and failures. For this assignment you will have to do the following:

   - The tests are written in a MIPS-style assembler language. See for example this simple test:

```
LOADMEM
zero r0 # 0 first, a zero instruction (see troubleshooting section)
lw r1 r0 5 # 1 loads a[0] into r1
lw r2 r0 6 # 2 loads a[1] into r2
add r3 r1 r2 # 3 put a[0]+a[1] into r3
DATAMEM # 4
10 # 5
-1 # 6
CHECKMEM
r1:  10
r2:  -1
```

```
r3:  9
pc:  4
END
```

This test involves four instructions: a zero instruction, two times loading and an addition (below `LOADMEM`). When running the test, the `LOADMEM` part is assembled into binary strings (in this case four strings). Then, the `DATAMEM` adds a `0`-instruction that will cause the simulator to halt here. Subsequently, a data part is provided with two numbers (below `DATAMEM`). In total, we have now seven binary strings: the first four are instructions, then a STOP-instruction, then two data strings. These are loaded into *both* your instruction RAM and your data RAM. So you will have to write your tests as if your architecture was a stored-program architecture: where the program and the data are in the same memory element. This means that you should be aware that you can reference and alter your program instructions!
In this case, the register r0 will be set to 0. Next, the word in memory address 5 (which is 10) will be loaded into r1, and the word in memory address 6 (which is -1) will be loaded into r2. Then, r1 and r2 are added, and the result is stored in r3, so r3 should contain the value 9. The actual tests are written below `CHECKMEM`: here we check whether r3 contains the value 9 and whether the program counter has value 3. You can check the value of the pc and any registers (unfortunately not of memory contents - you will have to load them into a register to check them). The check is performed after the last instruction (in this case, the addition). The test is ended by the `END`-line.
You can add multiple test programs to the same file, by simply starting a new `LOADMEM` after the `END`. You can also perform checks at a specific point in your code, by adding a `DATAMEM` block at that point. To test the lw-instruction more thoroughly we can alter the above test as follows:

```
LOADMEM
zero r0 # 0 first, a zero instruction (see troubleshooting section)
lw r1 r0 5 # 1 loads a[0] into r1
CHECKMEM
r1:  10
LOADMEM
lw r2 r0 6 # 2 loads a[1] into r2
CHECKMEM
r2:  -1
LOADMEM
add r3 r1 r2 # 3 put a[0]+a[1] into r3
DATAMEM # 4
10 # 5
-1 # 6
CHECKMEM
r3:  9
pc:  4
END
```

Your goal is to add significant tests to `Test_GroupXX.txt`. Write *a lot* of tests. Implement all operations in your simple datapath and create for each multiple tests in your test file.

- All files must be in the same directory. The program must be executed from the console as follows:
  `python Test.py -s -i TestFile.txt -c SD_GroupXX.circ`
  with `TestFile.txt` as the file containing your datapath tests and `SD_GroupXX.circ` as your logisim file (change XX to your group number). Note the "-s" flag, indicating the tests are for the simple datapath. Some lines will be 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 the expected result did not match what you have specified in your test. If you have failure or error tests, there will be some information about this failure/error in the output.
- If your tests fail while you expect them to be successful, try the following:
  (a) Double-check your solution to make sure that there is no error in your datapath. Make sure that you have connected your register file to the register outputs correctly!
  (b) The script has generated some file(s) named `TestFile.textX`, which contain the compiled hexadecimal version of each test program you wrote in `TestFile.txt`. In Logisim, use these generated `TestFile.textX` files to load them in your RAM-element (right-click your RAM-element, select "load image"). Check in Logisim whether the outcome is correct. If this is the case but the corresponding test failed when executed with the script, it means that there is something wrong with the execution of the script, so continue with step c.

(c) Check the corresponding `TestFile.reportX` file. The file is a printout of the simulation in Logisim of the corresponding test. It can be read line per line as follows: the first 16 bits is the pc value at a given clock tick, the remaining 8 16-bit numbers are the register values (r0 to r7) at this clock tick. Since your pc should increase every clock tick, the first column should also increase by one on each line. If this is not the case, and some pc values appear twice below each other, this probably means that values are only read at the rising edge. In this case, please check your registers (i.e., D-Flipflops used in your register) in Logisim and make sure they are all edge-triggered on the falling edge.

(d) If you still have not found the error, you might have hit a bug in the Logisim simulator that only occurs on some platforms. This bug can be resolved as follows: try starting every test program with `zero r1` (don't forget to also update address values as the whole program shifts by one...). For some reason, the Logisim simulator does not seem to be able to handle certain calculations on the first clock tick on some platforms.

(e) If you experience "random" bugs where in some test runs your test fails, but in others your test passes, then the following should resolve this: in Logisim, disable the option Project→Options→simulation tab→Add Noise to component delays.

(f) If all fails, contact me.