Computer Systems and -architecture

Project 5: 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: **Wednesday November 27, 2024, 22u00**
- Evaluation and feedback: **Wednesday December 4, 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. Build a circuit that implements a **12-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 12-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	12 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	12 bits	the address of the instruction in the instruction memory

2. Implement a partial datapath of 16-bit instructions and 12-bit data words and addresses by using your register file, a *data* RAM element (12-bit addresses, 12-bit words), your program counter with *instruction* RAM element (12-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).

3. Your datapath has to be able to read 16-instructions and act upon them. For this purpose you should construct a **decoder** circuit that reads instructions from the instruction memory and outputs the different values contained in the instruction:

name	in/out	width	meaning
Instruction	I	16 bits	The instruction that needs to be decoded
Operation code	O	3 bits	Operation code
Function code (if needed)	O	3 bits	Optional function code
rd	O	3 bits	register number for rd
rs	O	3 bits	register number for rs
rt	O	3 bits	register number for rt
immediate	O	12 bits	the immediate value, unsigned/sign extended to 12 bits

4. Your datapath is composed of different components and they all need to work together. A **controller** circuit will be needed to coordinate all these components. This circuit will have two inputs: operation code (3 bits) and function code (3 bits). The outputs will depend on how you decide to construct your datapath. Potential outputs are "Read memory", "Write memory", "ALU operation", etc.

5. The instructions have a consistent format. When designing your decoder and controller, try to see if there are any patterns in the instruction formats to make the circuit less complex:

   - 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.
     - For binary operations, registers are used as follows:
       `$rd := $rs` *`operation`* `$rt`
     - For unary operations (i.e., those that do not require a second operand), 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 datapath must be able to perform the load word (lw – reading from *data* RAM, op-code `1010`), and store word (sw – writing to *data* RAM, op-code `1011`) 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.

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`
`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`

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

- 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. You can also use the test-script.

6. 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_2425_zit1.py`. You need to install Python (`http://python.org/download/releases/2.7.3/`) to run `Test_2425_zit1.py`. Download `Test_2425_zit1.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 `CHECKMEM` 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. 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_2425_zit1.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 12 bits is the pc value at a given clock tick, the remaining 8 12-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.