KTH Information and
Communication Technology

IS1200/IS1500

Lab 4 – Processor Design
2024-07-30

Introduction

Welcome to this lab! In this laboratory exercise, you will learn the fundamentals of processor design. After finishing this lab, you should be able to

1. Construct a simple single-cycle processor that executes an actual assembly program.
2. Explain how bit selection and instruction decoding work.
3. Explain the basic principles of the data path and the control unit.

Preparations

You must do the lab work in advance, except for the Surprise Assignment. The teachers will examine your skills and knowledge at the lab session.

We recommend that you book a lab session well in advance, and start preparing at least a week before the session.

You can ask for help anytime before the lab session. There are several ways to get help; see the course website for details and alternatives. We especially encourage you to post questions on KTH Canvas page.

During the lab session, the teachers work with examination and offer help. Make sure to clearly state that you want to *ask questions*, rather than being examined.

Sometimes, our teachers may have to refer help-seekers to other sources of information, so that other students can be examined.

Software

In this lab, we use the the open-source software **Logisim** or **Logisim Evolution**. You can download and install the software from http://cburch.com/logisim/ and https://github.com/logisim-evolution/logisim-evolution respectively. If you have not performed the LD-Lab, you must learn the basics of using this software before you do Lab 4.

To teach yourself how to use the Logisim tool, please go through Logisim's “Beginner's tutorial.” This tutorial can be found in the documentation section on Logisim's web pages. Note that you need to perform each of the steps using the software to really learn how to use the tool.

If you are taking course IS1200, you should already know the basics of digital design and the content of the LD-Lab (Logic Design), that is only examined for students taking IS1500. However, if you find Lab 4 a bit hard, we recommend that you perform at least parts of the LD-Lab to refresh your knowledge about digital design.

Resources and Reading Guidelines

- Lectures 9.
- Harris & Harris (2021) course book. Chapters 5.2.4 (about the ALU), and 7.1, 7.3 (about the single-cycle processor). The book is available online.
- The RISC-V Reference Sheet.
- The RISC-V Instruction Set Manual (Volume I: Unprivileged Specification)
- See the course web page *Literature and Resources* for links and details.

Examination

Examination is divided into three parts:

Part 1 – Assignments 1 and 2.

Part 2 – Assignments 3 and 4.

Part 3 – Assignments 5 and 6 (the surprise).

Examining one part takes between 5 to 15 minutes. Make sure to call the teacher immediately when you are done with an assignment.

Please state clearly that you want to *be examined*, rather than getting help.

The teacher checks that your solution behaves correctly, follows all coding requirements and is easily readable.

The teacher will also ask questions, to check that your knowledge of the design and implementation of your solution is deep, detailed, and complete. When you create your solution, remember to make detailed notes. With these notes, you can quickly refresh your memory during the examination, if some particular detail has slipped your mind.

You must be able to answer all questions. In the unlikely case that some of your answers are incorrect or incomplete, we follow a specific procedure for re-examination.

Assignments

Throughout all assignments, test your components carefully by trying different input data and checking the output against expected output.

Note: Before you start, download file `processor_riscv.circ` (logisim) `or` `processor_riscv_evolution.circ` (logisim evolution) from the course website[1]. This is the template project in which you should implement *all* your solutions. Open this file in Logisim and start your processor design journey!

Assignment 1: Arithmetic/Logical Unit (ALU)

The ALU is the heart of the processor and is the digital building block that performs the actual computations. The purpose of this assignment is to gain a deeper understanding of how an ALU can be designed by implementing a basic ALU. The ALU is very similar to the one presented at Lecture 9, with some minor differences/simplifications.

Task 1.1
Open circuit ALU in the `processor_riscv.circ` (logisim) or `processor_riscv_evolution.circ` (logisim evolution) project. Implement the ALU with the following properties:

- One 32-bit wide data input port named A.
- One 32-bit wide data input port named B.
- One 3-bit input port named F that determines the function of the ALU.
- One 32-bit data output port named Y that outputs the result of the selected function.
- One 1-bit output port named `Zero` that output value 1 if all bits of Y is zero else it outputs value 0.

Note that the input and output components and some other help components are already located in the **ALU** circuit.

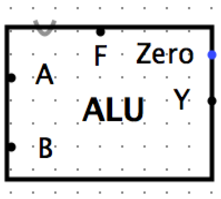
The ALU should implement the following functionality.

- F = 000_2 should perform operation A + B.
- F = 001_2 should perform operation A - B.
- F = 010_2 should perform operation A AND B.
- F = 011_2 should perform operation A OR B.
- F = 101_2 should perform operation *set less than (SLT)*. That is, if A < B then the ALU should output value 1, else it should output value 0 (you may use the *comparator* component)

For all other values of F, the output is undefined (we do not care what the output is).

The layout will look like this when another circuit uses the ALU circuit:

1 We offer two different circuit files depending whether you use logisim or logisim evolution since formatting changed between them.



The layout for the component is already provided in the project.

Note: You are not allowed to use the Subtractor component in Logisim for this lab. However, you are allowed to use all the other pre-defined components.

Some advice:

- Input pins and output pins can be configured to be 32-bit. Note, however, that the most significant byte is displayed at the top and the least significant byte at the bottom. Take a look at Example 1 in circuit **Tests** (part of the project) and convince yourself of the bit structure.
- The ALU is very similar to the one in Harris & Harris in Section 5.2.4 and Lecture 9. Note however the new **Zero** output port, which must be defined using Logisim components of your choice.
- One option to perform the zero extension (for the SLT operation) is to nest two *Splitter* components to extract the sign bit, followed by a *Bit Extender* component, which can be found in the *Wire* library folder. As an alternative, use the **Bit Selector** component that can be found in the **Plexers** library folder.

Questions for Assignment 1

These questions are useful for preparing for the oral examination. However, the teacher may choose to ask questions that are not on this list at the examination.

- Explain how each of the ALU functions are defined. In particular, you need to be able to explain how subtraction works, including the use of two's complement.
- How did you implement the logic for the **Zero** output port? Did you consider any alternatives? Be prepared to explain your design choices.
- What is the purpose of the ALU? Why are several functions grouped together into one component?

Assignment 2: Register File

In this assignment, you will implement a register file with 2 read ports and 1 write port. The register file will then later be used to store the register values for the processor. This register file will only be able to handle the first eight registers (`0`, `ra`, `sp`, `gp`, `tp`, `t0`, `t1`, and `t2`). All registers are 32-bit. While this simplification forces us to use a rather limited (and non-standard way) of using the registers, we do this to avoid making the circuits too complex.

In circuit **RegisterFile**, an almost complete 3-bit addressed register file is provided.

Task 2.1

Register 0 (also called zero) has a special semantics. Implement the correct behavior for 0 in the circuit.

Task 2.2

For debugging purposes in later assignments, it is convenient to output the value for register t2. Add connections to the circuit so that register t2 is sent to output port t2.

Questions for Assignment 2

These questions are useful for preparing for the oral examination. However, the teacher may choose to ask questions that are not on this list at the examination.

- Explain if the read, write, or both operations are clocked (updated at the clock edge). Why is it implemented this way?
- Explain the semantics of reading from and writing to register 0, and how you implemented this behavior.
- How many bits of data can this register file store? If the address width were the same size as that of a complete 32-bit RISC-V processor, how many bits would the register file store?

Assignment 3: Control Unit

In this assignment you will implement a small control unit. You need to have a good understanding of both the control unit and the data path for the single-cycle processor, as presented at the lectures and in the course book. Note also that this control unit is a simplified version of the unit presented at the lectures and in the book; it cannot handle all instructions.

Circuit **Control Unit** is prepared with correct (but not optimal) implementation that can handle instructions **addi** and **add**.

Task 3.1

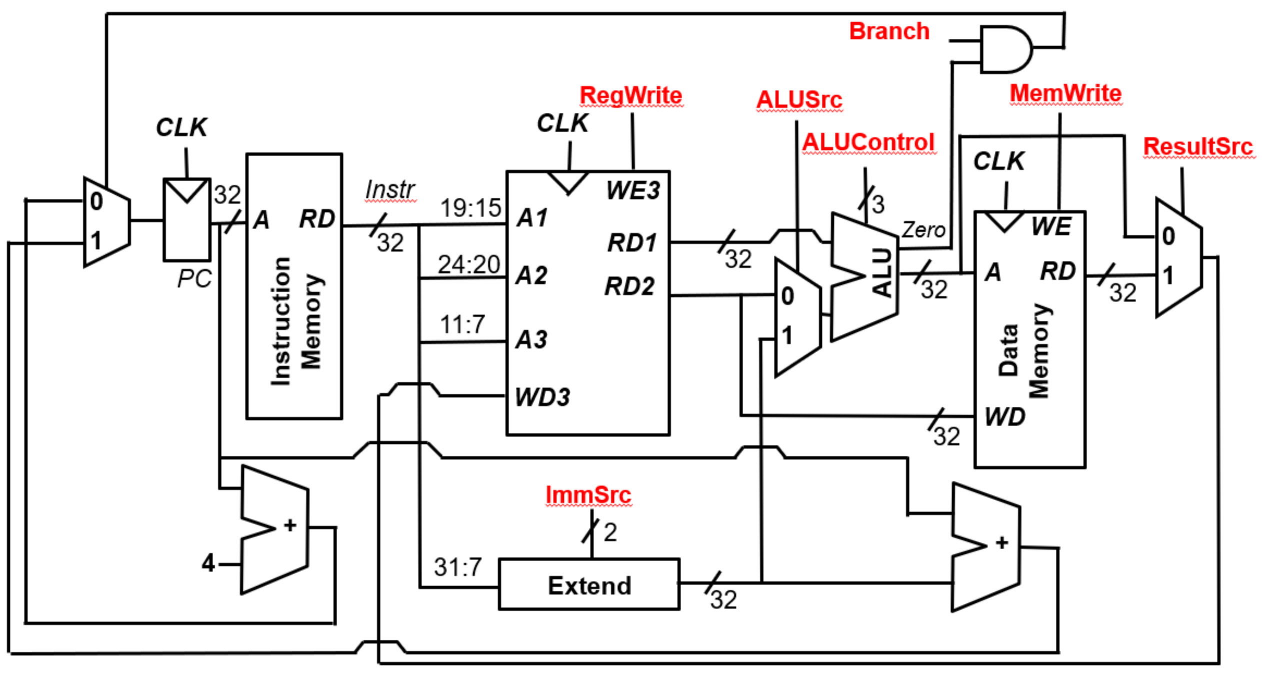
Add logic and components to circuit **ControlUnit** so that it can also handle the correct control signals for instruction **beq** (*hint:* use the ALU to test for the branching condition)**.**

Note: unlike the control signals in the slides (and in the literature), which have an **ImmSrc** signal, we can get away with not having it in our ControlUnit. Convince yourselves why we can get away with this in our design (*hint:* what is the maximal size our PC can address, and what implications does this have on reading the 12-bit immediate offset from branch instructions?).

Questions for Assignment 3

These questions are useful to prepare yourself for the oral examination. However, at the examination, the teacher may choose to ask questions that are not on this list.

- Explain how you have implemented the control signals for the **beq** instruction. Why is this a correct solution?
- Be preprared to explain:
  - What is the purpose and usage of the *ImmSrc* and *AluSrc* control signals (using the figure below)?
  - Why do we not need the *ImmSrc* signal in our simpler processor?



Assignment 4: Data path

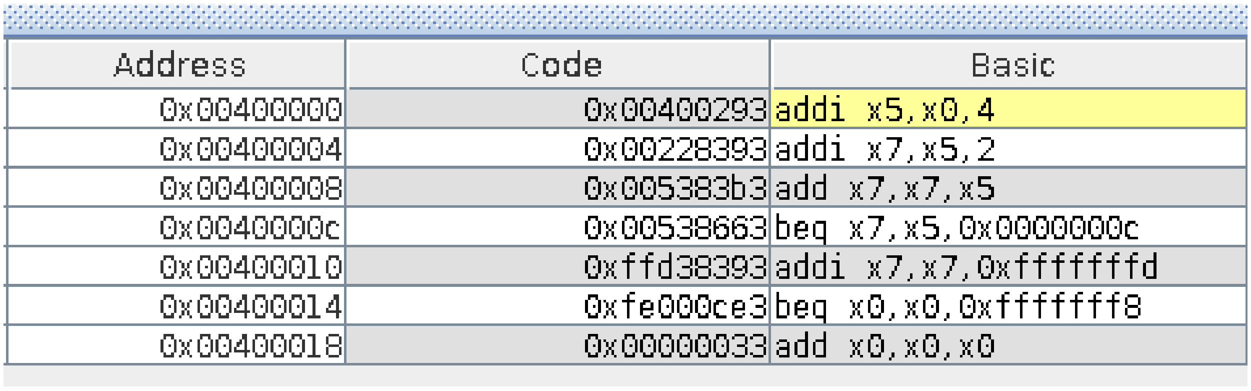
In this task, you should create a data path that can handle three RISC-V instructions correctly: instructions **addi**, **add**, and **beq**. This is the most demanding assignment in this lab, so follow the instructions below carefully. It is very important that you verify each of the instructions using the supplied test program.

Task 4.1: Updating the program counter and fetching instructions

1. Go to the **Datapath** circuit. All components in this assignment should be added to this circuit.
2. The circuit already contains the register for the PC counter and the connection to the instruction memory. Add logic for incrementing the PC with value 4 (next instruction). Note that the address width of the program counter is 8 bits, so that we do not need to allocate a too big ROM memory. Test the circuit with the provided clock.
3. The code memory should be implemented using the read-only memory (ROM) component in Logisim (under library director Memory). This component is already provided in the **Datapath** circuit. The address bit width is 8 bits and the data bit width 32 bits. Note also that we have added a shift component before the ROM address input (shift right by 2) so that the PC address corresponds to extracting one word (4 bytes). Initiate it and test that you can fetch instructions after each other. To see that it works correctly, it is good to add an output port, that you can for observing output during testing.

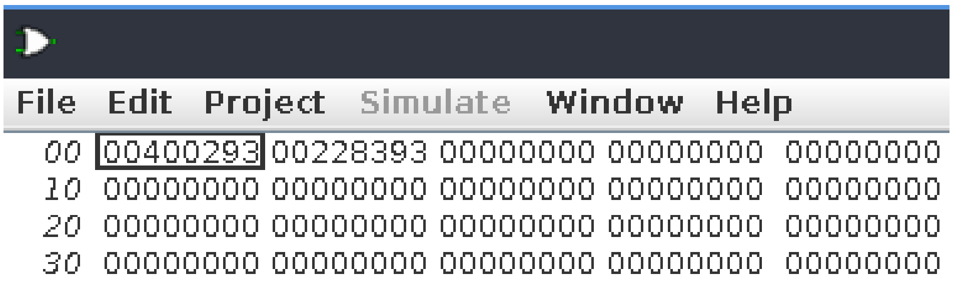
Task 4.2: Implementing the addi instruction.

1. The first thing to do is to create a test program that we can use to see that the operation works. Download the program `test.S` from the course web page. Open the program in the RARS simulator and go to the execute view (shown below). In the **Code** column, we have the machine code for the instructions. Run the program and convince yourself of its behavior.



Address	Code	Basic
0x00400000	0x00400293	addi x5,x0,4
0x00400004	0x00228393	addi x7,x5,2
0x00400008	0x005383b3	add x7,x7,x5
0x0040000c	0x00538663	beq x7,x5,0x0000000c
0x00400010	0xffd38393	addi x7,x7,0xfffffffd
0x00400014	0xfe000ce3	beq x0,x0,0xfffffff8
0x00400018	0x00000033	add x0,x0,x0

2. Enter the machine code for the two first instructions in the ROM memory (the two first **addi** instructions). Before you add the instructions, right click on the component and select “Clear contents”. After that you have added the instructions, the edit view should then look like this:



```
00 00400293 00228393 00000000 00000000 00000000
10 00000000 00000000 00000000 00000000 00000000
20 00000000 00000000 00000000 00000000 00000000
30 00000000 00000000 00000000 00000000 00000000
```

NOTE: you might need to save the content to a file and then load it again to be able to update the memory in the component.

3. Add the **RegisterFile** and the **ALU** components. See the course book and lecture 9 for details about how you implement an immediate instruction. Note that the address width is 3 bits for the **RegisterFile**, which means that you should only use the 3 least significant bits of the RISC-V addresses after decoding. To select bits, use the **Splitter** (or, if you prefer, the **Bit Selector)** component (can be found in the Plexer library folder). In the circuit **Tests**, there are two examples that shows how the Splitter component can be used. You probably need to play with these examples to understand how it works.
4. Attach the data path to the relevant control signals of the control unit. Do not forget to feed the instruction into the control unit.
5. Test that the execution sequence of the 2 instructions works as expected. You may need to add a few output components to see what happens in the circuit.

Task 4.3: Implementing the add instruction (R-type).

1. Open file `test.s` again in RARS and enter the three first instructions to the ROM component (including the third instruction that is an **add**-instruction).
2. Implement the necessary logic for the **add** instruction in the data path. Note that you need to add one (or more) multiplexors. Connect these multiplexor(s) to the right control signals.
3. Test that the sequence of two **addi** and one **add** instruction work correctly.

Task 4.4: Implementing the beq instruction.

1. Add the logic for the **beq**-instruction. Connect the relevant control signals.

2. Implement the rest of the test program `test.S` into the ROM. Initialize the PC to 0x14 and check that the jump is performed.

Questions for Assignment 4

These questions are useful to prepare yourself for the oral examination. However, at the examination, the teacher may choose to ask questions that are not on this list.

- Explain how the bit selection works. Which instructions are using what logic and why?
- Explain how the **beq** instruction implemented, how the address is calculated, and how the control unit controls the signals.

Assignment 5: Factorial Function in Assembler

In this task, you should develop a small RISC-V assembly program and execute it on your new processor.

Task 5.1

Open the RARS simulator and implement an assembly program that computes the factorial number **n!**, where **n** is a input parameter. The program has the following requirements:

- Only three instructions are allowed to be used: **add**, **addi**, and **beq**.
- Only registers with numbers 0 to 7 are allowed to be used. That is, you are allowed to use registers `0, ra, sp, gp, tp, t0, t1, and t2.`
- The input value should be given in `t0`. That is, the first instruction should assign `t0` to value `n`.
- The final result should be stored in `t2.`
- At the end of the program, implement a "stop" loop, i.e., a loop that that jumps to itself forever.

Note that you do not have a **mul** operation available. Hence, you should simply implement the mul operation by performing a loop with **add** instructions. This is not the fastest implementation, but it is good enough for testing your processor!

Task 5.2

In the RARS simulator, test that the program works for at least 0!, 3!, and 8!.

Task 5.3

Move the machine code of the complete program into your processor's ROM memory. Test that your processor gives the correct result for at least 0!, 3!, and 8!

Some advice: You can easily move your machine code from the RARS simulator into the ROM memory in Logisim.

1. In RARS, assemble the program.
2. Choose "Dump Memory..." from the File menu in RARS
3. In the "Dump Memory To File" dialog window, set "Dump Format" to "Hexadecimal Text".
4. Click "Dump To File..." and save the file on your system as a .txt file such as, `riscv1.txt`.
5. Use a plain text editor, such as *Notepad* or *Gedit*, to open the file (`riscv1.txt`), to add a new line at the very beginning of the file with the following contents:
```
v2.0 raw
```

6. Save the file, then load it into Logisim (“Load Image”)

On the next page, there is a graphical overview of the procedure.

Questions for Assignment 5

These questions are useful to prepare yourself for the oral examination. However, at the examination, the teacher may choose to ask questions that are not on this list.

- Show and explain how the factorial function works for arbitrary input value **n** (the teaching assistant will give you the value that you should test). Be prepared so that you know how to change the input value easily.
- Explain how you implemented unconditional jumps in your program.

Optional: Another way of compiling and transfering your code

As an alternative to dumping the memory from the RARS simulator, you can actually use the very same RISC-V compilers to perform the compilation and extraction, although it might be more complex. To do this, you need to:

1) Download the time4riscv.zip file from the Canvas homepage and extract it into a directory (this is the same file used in the labs for the board)
2) Remove all files from the directory except `boot.S`, `Makefile`, and `dtekv-script.lds`
3) Edit the boot.S and remove everything from it (that is, make it empty). The reason for this is that your processor only supports the **addi**, **add**, and **beq** instructions, where-as the `boot.S` code has some other instructions.
4) Finally, create a new file with containing the factorial code, and compile everything by writing: `make`
5) If all went smoothly, you should now have a main.bin file, which is the raw file containing your machine code. Next, use a program to dump the hex file, for example by invoking:
```
hexdump -ve '1/4 "%.4x "' main.bin
```
6) You can take the the dumped content and copy+paste it into the ROM in your logisim processor.

Assignment 6: Surprise assignment

You will get a surprise assignment at the lab session. We will hand out the surprise assignment directly at the beginning of the lab, presupposed that you have finished all the other assignments. If you have not received the surprise assignment, please ask the teaching assistant for the assignment paper.

Revision history

2024-07-30: The lab was now ported to RISC-V.

2016-08-24: Added a requirement to Assignment 1: the Subtractor component must not be used.

2016-01-14: First published version for Fall, 2016.

2017-01-17: Added instructions on how to move machine code from MARS into Logisim.