02132 Assignment 2: Hardware implementation in Chisel of a small CPU running the image erosion

Luca Pezzarossa and Jan Madsen

October 2, 2023

Preface

In this assignment, you are required to implement in hardware a small CPU (Central Processing Unit) and run the erosion step from the algorithm for the detection and the counting cells we presented in Assignment 1. The erosion step needs to be compiled by hand using an ISA (instruction set architecture) defined by you. This assignment aims to let you practice with compilation/assembly and hardware design. Thus, giving you a detailed understanding of how a CPU works and how software is run by the hardware.

The hardware CPU should be implemented using the Chisel language. Chisel (Constructing Hardware In a Scala Embedded Language) is an open-source hardware description language (HDL) used to describe digital circuits at the register-transfer level. Chisel is developed and supported by The University of California, Berkeley and it has been explored by several companies and institutions. For example, Google has used Chisel to develop tensor processing units for the edge. We selected Chisel instead of more common hardware languages like VHDL or Verilog since it is more friendly for developers with a software background, it is open-source, and the tool-chain is multi-platform. After all, designing hardware is a skill that is language independent.

As presented in the course introduction, this assignment is the most challenging of the three assignments of the course. The main reason being the fact that designing and testing hardware requires a completely different approach than the one used for software. In hardware, everything is parallel and debugging is mainly done graphically using waveforms that shows the value of all the signals of your circuit at any moment in time. This is a steep learning curve for beginners with only software developing experience, but it is also a very valuable learning experience for understanding what is really going on in the hardware when you run your code.

1 General information

This assignment should be carried out in **groups of two or three people** and you are free to keep your previous group or change it. In any case, you need to re-register the group in DTU-Learn. In addition to the designing tasks, you are also required to prepare a short report describing the approach used in its implementation (further details are provided in Section 5). All the material related to this assignment can be found on the DTU-Learn course page at the following location:

`DTU-Learn/Course content/Content/Assignments/Assignment 2`

The deadline for this assignment is **Sunday 5$^{\text{th}}$ November 2023 at 11:59**. By this date, you have to hand-in an electronic version of the short report in PDF format and the source files as ZIP archive using the assignment utility in the DTU-Learn course page at the following location:

`DTU-Learn/Course content/Assignments/Assignment 2`

Before the deadline, during the last laboratory session dedicated to this assignment held on **Wednesday 1$^{\text{st}}$ November 2023**, you are asked to demonstrate the functionality of your implementation to the teacher or to the TAs. More information regarding the DEMO session will follow as DTU-Learn announcements.

This document is divided into 5 sections and 1 appendix:

- Section 2 is a guide on how to install the Chisel tool and also provide pointers to online resources related to Chisel.
- Section 3 provides the general background needed for the assignment, describes the provided code and the overall CPU architecture.
- Section 4 presents the developing and testing tasks you need to carry out in this assignment.
- Section 5 discusses time-management, report requirements, and evaluation criteria.
- Appendix A provides an example of a small instruction-set architecture.

There are many ways to compile the erosion step and design an ISA and a CPU. Similarly to Assignment 1, we are not enforcing a particular approach (besides the already implemented modules and tests), rather we would like you to explore. For this reason, it may be difficult for the teachers to understand your implementation and thus, help us read/debug your design by taking into account the following hints:

- Make sure you understand the role of the ISA and the basic principles of hardware design.

- Structure your assembly code and hardware design such that it is clear what the different parts/modules are doing.
- Test your hardware modules individually with a dedicated tester before connecting everything together.
- Explain how the erosion step compiled in assembly works.
- Comment your hardware design (when needed).
- Deliver all the source code needed to test the implementation. Add a README file with instructions on how to run the test cases including the ones you developed.

Feel free to ask or send an e-mail to the course teacher if you have questions regarding practical matters and the assignment in general.

2 Setting up and working with Chisel

This section[1] is a guide on how to install the Chisel toolchain Windows, Linux, and Mac. In addition, we will also provide useful pointers to Chisel documentation and material online. We have tested the steps presented in the following at our best using our own computers. However, this does not guarantee that they will work flawlessly to any computer due to different flavours/configuration of your own machines. If you encounter problems that you cannot solve yourself with a Google search, contact a TA or the teacher during the laboratory session.

Chisel is just a library for Scala. And Scala is just a language that executes on the Java virtual machine (JVM) and uses the Java library. Therefore, you need to have Java OpenJDK 8 installed on your laptop. **It is important to use version 8 (also called 1.8) of Java, not a newer one. If you have a newer one installed you do not need to uninstall it, but you need to install also the required version.**

For working on the command line you should also install sbt, the Scala build tool. Please note that installing sbt will make the IntelliJ-build process a lot easier as well. A nice editor for Chisel/Scala is IntelliJ. In summary, the tools we need to install are:

- Java OpenJDK 8
- sbt
- IntelliJ
- GTKWave

[1] This section contains material derived from the open-source documentation developed by Martin Schoeberl and available here and here. Martin Schoeberl is an associate professor at DTU-Compute in the Embedded System Engineering section.

Install Chisel in Windows

1. Install OpenJDK 8 (HotSpot) from AdoptOpenJDK, which will froward you to the Adoptium website for download. **It is important to use version 8 (also called 1.8) of Java, not a newer one. It is also important that you enable the "Set JAVA_HOME" for installation.**
2. Install sbt.
3. Install GTKWave and put a link to the executable on the desktop. When installing GTKwaves for Windows, please download the binary version form the web-page.
4. Install IntelliJ (the free Community edition) and create a desktop shortcut.
5. Start IntelliJ to finish the setup.
   - Select the UI theme you prefer better in the Customize tab.
   - Select Install for Scala in the Plugin tab.
   - If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
     - On Project JDK select New.
     - Select JDK.
     - Select the path to your OpenJDK 8 installation, usually `C:\Program Files\AdoptOpenJDK\jdk-8.0.232.09-hotspot\`

Install Chisel in Linux

1. Install OpenJDK 8 **(it is important to use version 8 (also called 1.8) of Java, not a newer one)** with the terminal command:

   ```
   sudo apt install openjdk-8-jdk
   ```

2. Install sbt (if on Ubuntu, deb archive is preferable).
3. Install GTKWave from here or with the terminal command:

   ```
   sudo apt install gtkwave
   ```

4. Install IntelliJ (the free Community edition).
5. Start IntelliJ to finish the setup.
   - Select the UI theme you prefer better in the Customize tab.
   - Select Install for Scala in the Plugin tab.
   - If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
     - On Project JDK select New.
     - Select JDK.
     - Select the path to your OpenJDK 8 installation, usually `C:\Program Files\AdoptOpenJDK\jdk-8.0.232.09-hotspot\`

Install Chisel in macOS

1. Install OpenJDK 8 from AdoptOpenJDK. **It is important to use version 8 (also called 1.8) of Java, not a newer one.**
2. Install sbt with the terminal command: `brew install sbt`
3. Install GTKWave.
4. Install IntelliJ (the free Community edition).
5. Start IntelliJ to finish the setup.
   - Select the UI theme you prefer better in the Customize tab.
   - Select Install for Scala in the Plugin tab.
   - If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
     - On Project JDK select New.
     - Select JDK.
     - Select the path to your OpenJDK 8 installation, usually `C:\Program Files\AdoptOpenJDK\jdk-8.0.232.09-hotspot\`

Visualizing waveforms using GTKWaves

As previously mentioned, in hardware, everything is parallel and debugging is mainly done graphically using waveforms that shows the value of all the signals of your circuit at any moment in time. Waveforms are generated by the Chisel simulator as `.vcd` file and can be loaded and visualized using GTKWaves. Figure 1 shows the main parts of the GTKWaves user interface. You need to move the signals you are interested in from the 'Available signals' area into the 'Selected signals' area.

Note: In the 'Available signals' area, you may find a lot of generated signals named `_GEN_[...]`. Just ignore these signals.

Note: In GTKWave, you can just reload waves when you re-run a simulation (File -> Reload waveform) to update the waveform without the need to open a new file and set up GTKWave again.

Test your installation

Once you have installed all the tools you need as listed above, you can test your installation by running the Hello example included in the provided code. The Hello example loads a 20-by-20 pixel binary image and produces the inverted version of it (negative image).

1. Open IntelliJ and open the `build.sbt` file in the project folder. Open it as a project (remember to select the JDK 1.8 and not Java 11 if prompted).

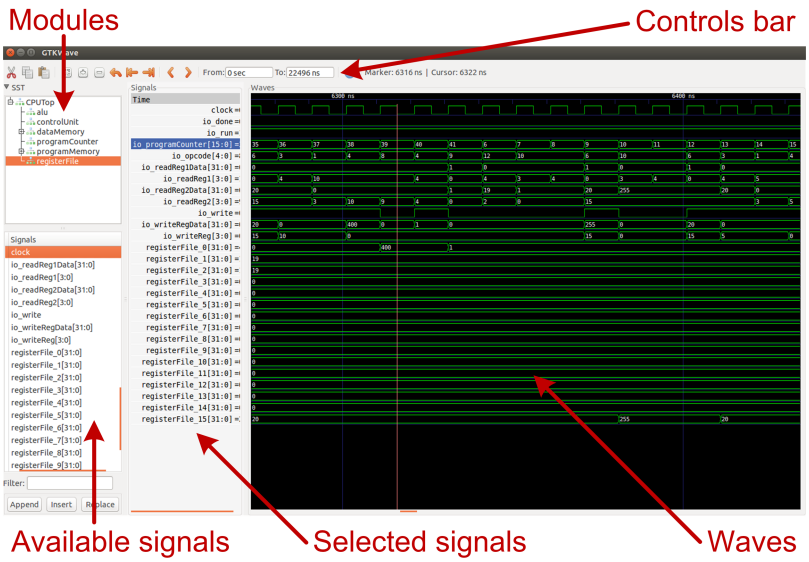

Figure 1: The main parts of the GTKWave user interface.

2. Explore the file structure and inspect the files
   `.../src/main/scala/Hello.scala`
   and `.../src/test/scala/HelloTester.scala`.

3. To run the simulation of the Hello module, you need to run in the IntelliJ terminal the command:

   ```
   sbt "test:runMain HelloTester"
   ```

   When the simulation is finished, you can examine the output in the terminal and see the original image and the inverted one. White pixels are represented by the symbol **o**, while black pixels are represented by the symbol *. Also, you can open the waveforms file `Hello.vcd` generated in the folder `.../generated/` by opening it with GTKWave.

If this works successfully, you are ready to start designing and testing hardware in Chisel.

Known issues and solutions

In the foll wing, we explain how to overcome some known issues with the the installation of the Chisel development environment. These solutions are based on the previous iterating of the course. Since tools are continuously evolving, the solutions might not work or new issues might appear. that have appeared during today's lab session.

Issue 1: Mac laptops with ARM-based processor

If you have a Mac laptop that uses the new ARM-based processors (e.g., M1 chip), you may not be able to install sbt with the provided instructions. Please try the following.

1. Install the package manager SDKMAN with:

   ```
   curl -s "https://get.sdkman.io" | bash
   ```

2. Close and reopen terminal or run the command:

   ```
   source "$HOME/.sdkman/bin/sdkman-init.sh"
   ```

3. Install Java 8 with:

   ```
   sdk install java 8.302.08.1-amzn
   ```

4. Install sbt with:

   ```
   sdk install sbt
   ```

5. Compile sbt with:

   ```
   sbt compile
   ```

Issue 2: Missing JAVA_HOME or wrong Java version

When running the tests in Chisel, the sbt commands expects to use Java 1.8. Sometime it is sufficient to change the Java version in IntelliJ by going to File → Project structure → Project SDK and selecting the right Java version. However, in some case this does not apply to the terminal integrated into IntelliJ. If this happens, try the following solution:

In Windows and Linux: Go into File → Settings → Tools → Terminal → Environment Variables (open it, do not write in the text box). Add an environment variable using the + button. The variable name is JAVA_HOME and the content is the path of the installation of Java 1.8

In MacOS: Go into IntelliJ IDEA → Preferences → Tools → Terminal → Environment Variables (open it, do not write in the text box). Add an environment variable using the + button. The variable name is JAVA_HOME and the content is the path of the installation of Java 1.8 You can have an hint of the path of the installation of Java 1.8 by looking at the sbt shell in IntelliJ. The path is printed on the first line (please just copy the path without the executable name).

Online Chisel documentation and material

The main reference book for learning Chisel is:

- *Martin Schoeberl, Digital Design with Chisel, 2nd edition, 2019* (free, open-access book available here). The book source is available here.

In addition, you can take a look at the following online documentation and material:

- The official Chisel website. Click Chisel3 for documentation.
- A collection of FAQ from another course using Chisel at DTU.
- The Chisel Cookbook: large FAQ and introduction to Chisel.

3 Background and specifications

Background

You still work on the cell detection and counting algorithm at the fictive company Bioware System and, after having implemented the algorithm in software using C, you are given the task to investigate if it is feasible and convenient to develop a small CPU to run the algorithm.

To carry out this task you have to:

1. Design your instructions set architecture (ISA).
2. Compile by hand the erosion step of the algorithm, which is a computation-intensive step.
3. Design your small CPU and implement it in hardware using Chisel.
4. Run the compiled code on your CPU in simulation.

Designing an instruction set, compile code by hand, and designing and implementing a CPU are very complex tasks. Thus, for this assignment, we have introduced some simplifications and made some assumptions.

The main simplification consists of the reduced size of the binary image on which we run the erosion step, which is reduced to 20-by-20 pixels. This is due to the fact that simulating hardware takes a long time. Performing the erosion steps of an image of 950-by-950 pixel is functionally equivalent to doing it for a smaller one. Figure 2 shows how the pixels are mapped in the 20-by-20 binary image. For this assignment, we also assume that back pixels correspond to the value 0 and white pixels correspond to the value 255.

In the following, we describe the CPU specifications and the provided code/infrastructure.

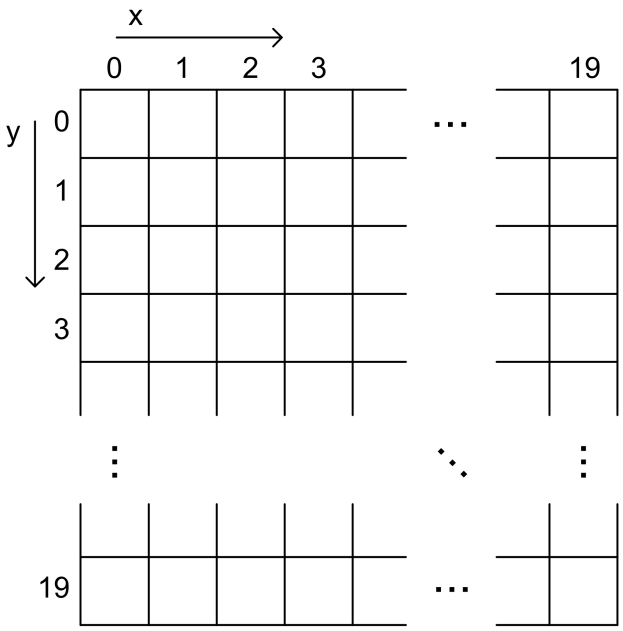

Figure 2: The pixel mapping of the 20-by-20 binary image.

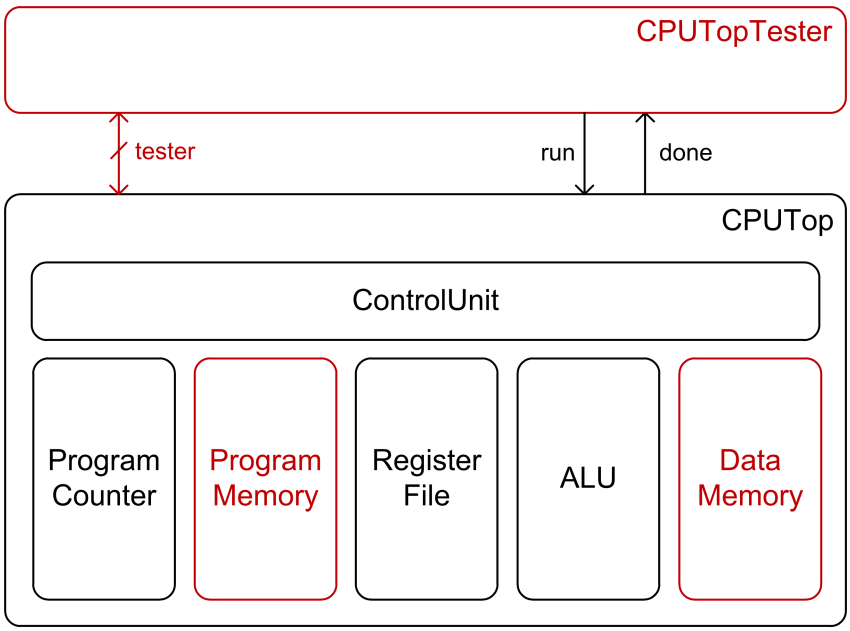

Figure 3: An overview of the full CPU system and the CPU tester.

Full system overview

Figure 3 shows the full CPU system and of the CPU tester. The names of the modules match the names of the files/classes of the provided code. The modules outlined in red are provided to you already implemented. To start, it is a good idea not to modify them. When you get more confident in Chisel you are of course free to modify them if you need.

Figure 4 shows the folder/file structure of the provided code. The folder `.../src/main/scala/` contains the hardware modules written in Chisel. The folder `.../src/test/scala/` contains the testers written in Scala/Chisel/Java. The folder `.../generated/` contains the results of the simulation. In this folder, you can find the waveform file (.vcd) which you can use to debug your hardware implementation. To run the simulation of a module, you need to run in the IntelliJ terminal the command `sbt "test:runMain <TESTER_NAME>"` where you specify the tester name you want to use. For example, to run the full CPU tester (file: CPUTopTester.scala), just run the following command:

```
sbt "test:runMain CPUTopTester"
```

When the simulation is finished, you can examine the output in the terminal (created using 'prints' in the tester file) and the waveforms file `CPUTop.vcd` generated in the folder `.../generated/` by opening it with GTKWave.

```
<project root>
├── generated
├── src
│   ├── main
│   │   └── scala
│   │       ├── ALU.scala
│   │       ├── ControlUnit.scala
│   │       ├── CPUTop.scala
│   │       ├── DataMemory.scala
│   │       ├── Hello.scala
│   │       ├── ProgramCounter.scala
│   │       ├── ProgramMemory.scala
│   │       └── RegisterFile.scala
│   └── test
│       └── scala
│           ├── CPUTopTester.scala
│           ├── HelloTester.scala
│           ├── Images.scala
│           ├── ProgramCounterTester.scala
│           └── Programs.scala
└── ...others...
```

Figure 4: The folder/file structure of the provided code.

Other files and folders besides the ones listed are present and generated, but they are not of interest for the scope of this assignment. Out of curiosity, please

note that a file with extension `.v` is generated in the folder `.../generated/`. This is a Verilog file that describes your hardware design and can be used to produce a real chip.

In the following, we describe the purpose and functionality of each module.

CPUTop module

This is the top level of your CPU. In this module, all the sub-module are connected together to build the CPU. It is good practise to not describe any major functionality in this module, but just use it to connect other modules. Of course, multiplexers, small boolean expressions, concatenation and splitting of signals can be used when connecting modules together.

The interface of this module consists of a bunch of `tester` inputs and outputs, a `run` boolean input and a `done` boolean output. The `tester` inputs and outputs are used by the CPUTopTester module to load the content of the program memory with your program and the content of the data memory with the binary image to be processed. Moreover, these are also used to dump the content of the data memory in order to retrieve and show the processed image. **Do not touch these signals unless you are confident in Chisel and you want to modify the memory structure and the CPUTopTester.**

The input signal `run` interacts with the CPU you are developing. When `run` is true (1) the processor execute the code. When `run` is false (0) the processor pauses execution. In other words, this means the program counter stops being updated when `run` is false.

The output signal `done` is driven by your CPU and becomes true when the execution of the program is finished. Also, when the signal `done` becomes true the program counter needs to stops being updated (end of execution – no more instructions are executed). This means that you need to include the instruction `END` in your instruction set to indicate the end of the program (as shown in the example ISA in Appendix A).

CPUTopTester module

This module is the tester of your CPU and it does not describe hardware. It contains sequential Scala, Java, and Chisel code that interacts with the CPUTop module in order to load and dump the memories and to print/show the images in the terminal.

The CPUTopTester module performs the following operation in sequence:

1. **Load the data memory with image data:** During this phase, the tester loads a 20-by-20 pixels image in the data memory (DataMemory module). The test images are stored in the file `.../src/test/scala/Images.scala`.

   We provide the following four test images (also shown in Figure 5):

   - **blackImage:** An image where all the pixels are back.

- **whiteImage:** An image where all the pixels are white.
- **cellsImage:** An image with 3 white spots (cells) to be eroded.
- **borderCellsImage:** An image with 3 white spots (cells) touching the image edges to test erosion functionality in the image edge.

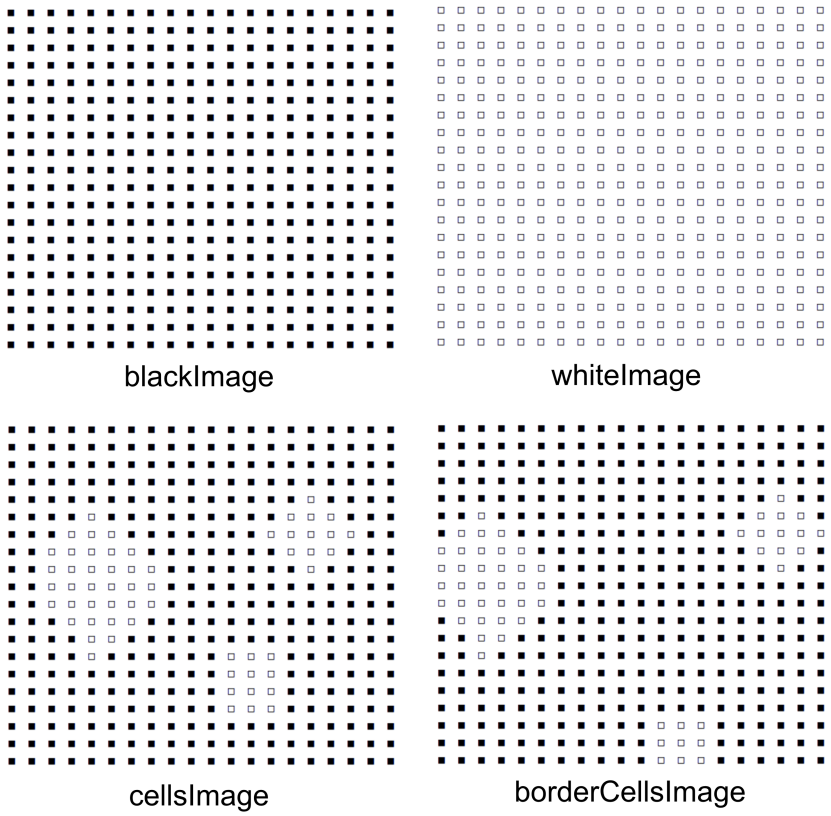

Figure 5: The four 20-by-20 test images provided.

You can select which image to load by changing the following line of code in the CPUTopTester:

```
var image = Images.cellsImage
```

You are free to add your own image to erode in the file `.../src/test/scala/Images.scala`.

The selected image is loaded in the data memory from address 0 to address 399. Each 32 bits entry of the data memory is used to store one pixel. Clearly, storing one pixel in a 32-bit word is wasting memory space, but it greatly simplifies the design of the CPU.

Pixels are stored row by row. This means that the pixel at coordinates (x, y) is stored at the address $x + 20 * y$. For example, pixel (0,0) is stored at address 0, pixel (7,14) is stored at address 287, and pixel (19,19) is stored at address 399. These are the addresses that your assembly program should use to access the pixel of the input image to be eroded.

2. **Load the program memory with instructions:** During this phase, the tester loads your program in the program memory (ProgramMemory module). You can provide the program in the file `.../src/test/scala/Programs.scala`. For testing purposes, you can put multiple programs in this file and select which one to load by changing the following line of code in the CPUTopTester:

   ```
   val program = Programs.program1
   ```

   The program consists of a sequence of 32 bits instructions and it is loaded in the program memory starting at address 0.

3. **Run the simulation of the CPU:** During this phase, the tester sets the `run` signal of the CPU to true (1) enabling the CPU to execute the code. This phase continues until the output signal `done` driven by your CPU and becomes true (1) signalling the end of the program execution or if a maximum of 20000 clock cycles have been simulated. Then the signal signal `run` is cleared to false (0).

   To to simulate more that 20000 clock cycles, change the following line of code in the CPUTopTester:

   ```
   var maxInstructions = 20000
   ```

4. **Dump the data memory content:** During this phase, the tester reads the data memory from address 0 to 799 to retrieve the original image and the processed one. The original image is expected to be stored from address 0 to 399 as explained earlier. The processed image is expected to be stored by your program from address 400 to 799. This means that the processed pixel at coordinates (x, y) is stored at the address $x+20*y+400$.

5. **Show original image:** During this phase, a graphical representation of the original image is printed in the terminal. Use this to ensure that your program did not affect the original image stored in memory. You can see the graphical representation of the input image in Figure 6.

6. **Show processed image:** During this phase, a graphical representation of the processed image is printed in the terminal. Use this to ensure that your program works as expected. You can see the graphical representation of a processed (eroded) image in Figure 6.

To run the CPUTopTester in order to debug and test your CPU, run the following command in the IntelliJ terminal:

```
sbt "test:runMain CPUTopTester"
```

When you look at the waveforms in the generated file `CPUTop.vcd` in the folder `.../generated/`, the part you are interested is when the signal `run` is true (1). This is the interval of time when your CPU is running. The waveforms where the signal `run` is false (0) shows the loading and dumping procedures of the program and data memories performed by the tester and should be ignored.

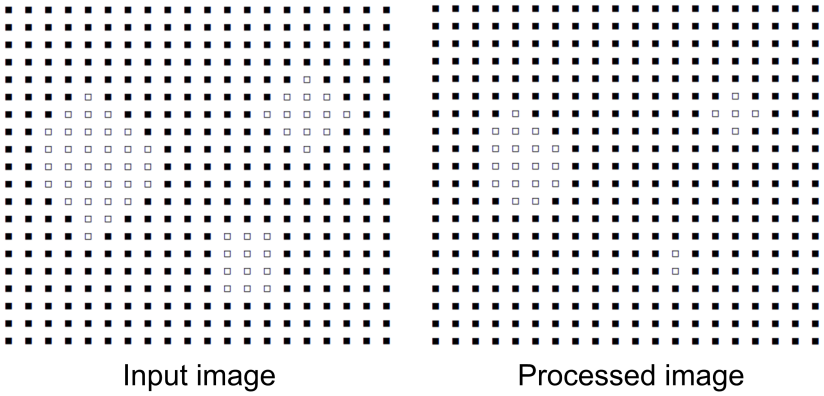

Figure 6: Original and processed images as represented in the terminal.

ProgramMemory module

The ProgramMemory module is the memory where the instructions of your program are stored and it is provided to you already implemented. The ProgramMemory has a size 65536 words of 32 bits and it is a read-only memory; hence, your CPU cannot write to this memory, but only read. Figure 7 shows the interface of the memory. When providing an address in the `address` input port, the corresponding 32 bits instruction stored at the required location is immediately provided in the `instructionRead` output port as shown in the timing diagram in Figure 8. The address size is 16 bits; thus, ranging from 0 to 65535.

DataMemory module

The DataMemory module is also provided to you already implemented and it is the data memory of the CPU where the input image, the processed images, and other data are stored. The DataMemory has a size 65536 words of 32 bits and the CPU can both write and read to this memory. Figure 9 shows the interface of the memory. When the `writeEnable` input is false (0) the memory can be read in the same way as the ProgramMemory. When the `writeEnable` input is true (1), the 32 bits data provided to the `dataWrite` input is written in the specified `address`. Figure 10 shows the timing diagram for a read and for a write operation.

ControlUnit, ProgramCounter, RegisterFile, ALU modules

The ControlUnit, ProgramCounter, RegisterFile, and ALU (Arithmetic and Logic Unit) modules need to be implemented by you according to the functionalities offered by your ISA and on how you plan to connect them together to build your CPU. Section 4 provides more information with regards to the development and testing of these modules.

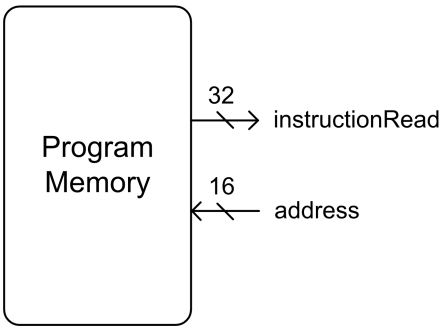

Figure 7: Interface of the ProgramMemory module.

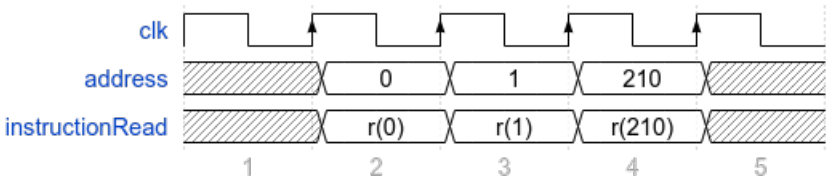

Figure 8: Timing diagram of the ProgramMemory module. In cycle 2, 3, and 4, reading operations to address 0, 1, and 210 are carried out.

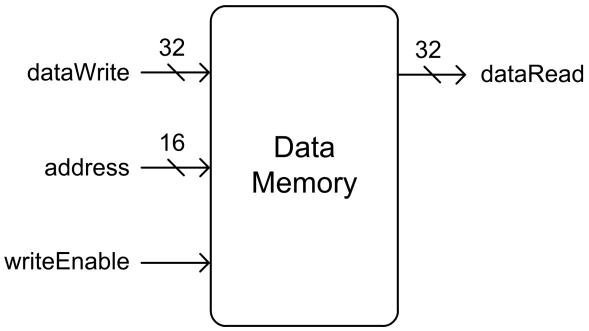

Figure 9: Interface of the DataMemory module.

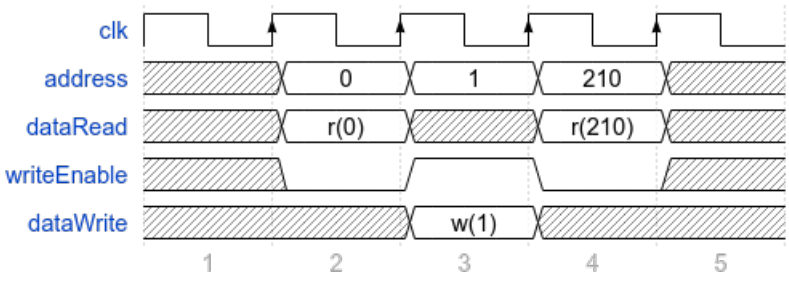

Figure 10: Timing diagram of the DataMemory module for read and write operations. In cycle 2 and 4, reading operations to address 0 and 210 are carried out. In cycle 3, a writing operation to address 1 is carried out.

Notes on the CPU development

In the following, we provide a list of notes to take into account during the development of the CPU. As usual, we do not enforce any detailed implementation and you are very welcome to explore the solution you like better. Of course, you need to describe, justify, and evaluate the design decision you make.

- The CPU is a single clock CPU; thus, it is not pipelined. This means that registers are only used in the resister file and in the program counter.
- The CPU should have data width of 32 bits, instruction width of 32 bits, and address width of 16 bits.
- In principle, you can develop a CPU that only works with unsigned numbers, since you do not need negative numbers in the erosion step. It is up to you to decide what number types your CPU supports.

4 Development and testing tasks

In this section, we list and describe the tasks you need to perform for the development and testing of your assembly program and CPU. The tasks are ordered according to a top-down development process. However, during development, it might be necessary that you go back and revise some of the decisions you made earlier. For example, if you used an instruction that is too difficult to implement in hardware or you if find a better way to implement certain functionality when you start implementing. This iterative approach is very common during development.

Task 1: Preparation

In this task, you should carefully read the assignment text and have a clear idea of what you need to do. Also, we recommend that you understand the basic concepts of a CPU architecture and role of the ISA (from the lecture).

In addition, you should set up your Chisel environment (see Section 2) and carry out a quick review of the Chisel syntax and of the provided code.

Task 2: Implement the ProgramCounter module

The program counter (ProgramCounter module) is a simple 16 bits counter that counts from 0 to maximum $2^{16} - 1$ or to the size of your program. In this task, you need to implement and test the ProgramCounter module in Chisel.

Since this is the first block you implement in Chisel, we will provide to you the file with a predefined interface and the block diagram of the counter as shown

in Figure 11. The behaviour is as shown in the following table, the symbol `X` means that it could be 0 or 1 indifferently (don't care):

`run`	`stop`	`jump`	**Operation**
`0`	`X`	`X`	`programCounterNext = programCounter`
`X`	`1`	`X`	`programCounterNext = programCounter`
`1`	`0`	`0`	`programCounterNext = programCounter + 1`
`1`	`0`	`1`	`programCounterNext = programCounterJump`

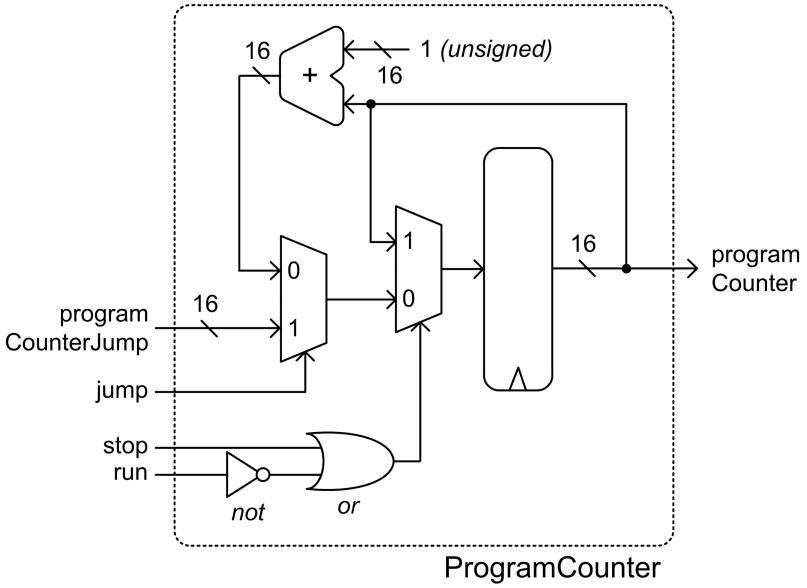

Figure 11: Diagram of the program counter.

The input `run` enables the counter to run and it is connected to the `run` input of the CPUTop. The input `stop` stops the counter when the program reaches and 'END' instruction. The input `jump` loads the value provided in the input `programCounterJump`, and it can be used by the jumps/branch instructions.

Test the ProgramCounter module by running in the IntelliJ terminal the command:

```
sbt "test:runMain ProgramCounterTester"
```

When the simulation is finished, you can examine the waveforms file `Hello.vcd` generated in the folder `.../generated/` by opening it with GTKWave and make sure the mule works as expected. For the other modules, you need to implement your own tester.

Note: When you run the testers (including the program counter tester), you will see 0 test passed. This has no relevance for you, since no tests are performed. To

verify your design (e.g., your program counter) you need to open the waveform file in GTKWave and check that the outputs are as expected. For the program counter, check the ProgramCounterTester.scala file to know what it is expected (count for 5 cycles, hold for 10 cycles, jump to 30, count for 5 cycles).

Task 3: Define the ISA and compile the program

In this step, you need to define your instruction set architecture and compile the erosion step by hand. You can get inspired by the MIPS ISA and/or by the simple ISA provided in Appendix A.

You can compile your own erosion step, or compile the erosion step algorithm provided as pseudo-code in Listing 1.

Listing 1: Pseudo-code for the erosion step.

```
for (x from 0 to 19) {
  for (y from 0 to 19) {
    //Processin border pixel
    if (x==0) { out_image(x,y)=0; continue; }
    if (y==0) { out_image(x,y)=0; continue; }
    if (x==19) { out_image(x,y)=0; continue; }
    if (y==19) { out_image(x,y)=0; continue; }
    //Processing inner pixel
    if (in_image(x,y)==0) {
      //Black pixel
      out_image(x,y)=0;
    } else {
      //White pixel, cheking neighboring pixels
      if (in_image(x-1,y)==0 or
          in_image(x+1,y)==0 or
          in_image(x,y-1)==0 or
          in_image(x,y+1)==0) {
        //Erode
        out_image(x,y)=0;
      } else {
        //Do not erode
        out_image(x,y)=255;
      }
    }
  }
}
```

In any case, your algorithm needs to take into account the border condition of the image. We recommend you verify the compiled program by hand on a very small image (e.g., 4-by-4 pixels). Your CPU implementation needs at least to support the instructions you use in your compiled program.

Task 4: Encode the instructions

Once you have your ISA, you need to encode the instructions into the 32 bits words that will get stored in the program memory. This means, that you need

to decide how the 32-bit instruction is divided into fields, and what each field represents.

Task 5: Draft the CPU architecture

Now that you have the ISA and the instruction encoding, you need to draft the block diagram of the CPU architecture by connecting together the modules presented in Figure 3. Depending on your instruction set you might need 0 or more multiplexers.

In other words, you are now defining how your components should be connected. **Please show your block diagram to the teacher or the TAs before continuing with the implementation - it might save you hours of trouble!**

Task 6: Implement the ALU module

Most CPUs contain a single module which is able to perform a number of arithmetic and logic functions, called Arithmetic and Logic Unit (ALU) and shown in Figure 12.

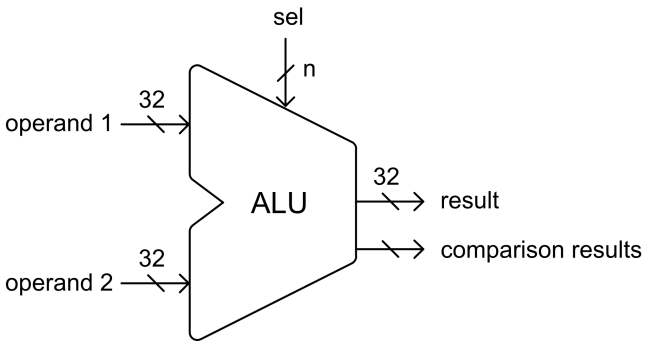

Figure 12: Interface of the ALU module.

The ALU must support all the arithmetic and logic operations you need in your instructions set. The value of the `sel` input determines the function applied to the two inputs according (add, subtract, and, or, pass through, etc.). Depending on your CPU design, the ALU may also contain comparison operations (equal, less than, greater than, etc.) which results is provided by additional outputs of the ALU. These outputs are used, for example, to decide whether to jump (branch) if two registers are equal (`BEQ - BRANCH IF EQUAL`).

Implement the ALU module, prepare a tester, and test the module in simulation.

Task 7: Implement the RegisterFile module

The register file (RegisterFile module) consists in a set of registers used to temporarily store the data on which the CPU is currently operating upon. This allows the CPU to save time when doing calculations since it only needs to fetch the data from the data memory once (in a real CPU this takes many clock cycles when compared to reading the value from a register).

Figure 13 shows the interface of a generic register file containing 16 registers of size 32 bits. This is the expected functionality:

- `aSel` selects between 16 different registers and outputs the contents of the selected register to `a`.
- `bSel` selects between the same 16 registers (as `aSel` does) and outputs the contents of the selected register to `b`.
- If `writeEnable` is true (1), the value provided in the input `writeData` is stored at the register indicated by `writeSel`.

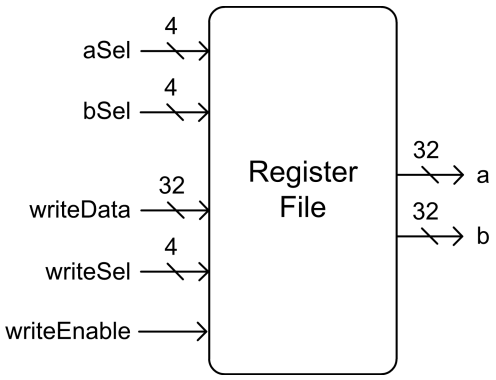

Figure 13: Interface of a generic RegisterFile module.

We recommend you use no less than 8 registers and not more than 32. It is up to you decide what configuration suits your design better. Implement and test this module.

Task 8: Implement the ControlUnit module

From the draft of the CPU architecture you developed in Task 5, you should be able to distinguish control signals and data signals. Control signals are the signals, which configure a particular component, such as for instance the `sel` signal of the ALU, the signals that control multiplexers, or the write enable for the data memory. These are the signals your ControlUnit module needs to drive. Depending on the opcode of the instructions the ControlUnit module should drive all the control signals in order to configure the datapath for the required operation. In summary, the ControlUnit module is just a decoder

which receives the instruction opcode in input and drives all the control signals of your CPU. Figure 14 shows the interface of a generic ControlUnit module. Implement and test this module.

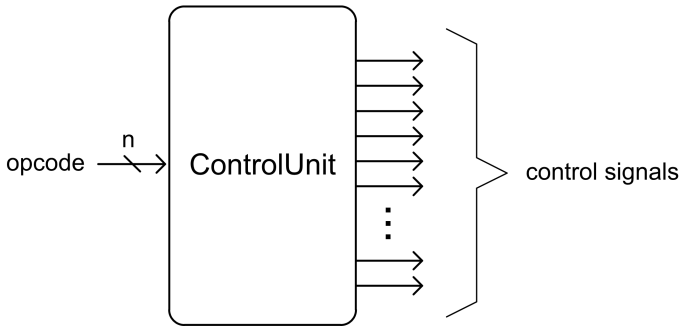

Figure 14: Interface of a generic ControlUnit module.

Task 9: Put everything together in the CPUTop module

In this task, use the modules you implemented and connect everything together in the CPUTop module according to the draft of the CPU architecture you developed in Task 5. You may need to use multiplexers and minimal boolean logic circuits (and, or, not) when connecting your components together.

Task 10: Run, test, and debug your program

At this point, your full CPU is ready to run your program. Enter the program you want to run in the file `.../src/test/scala/Programs.scala` and start the simulation by running the following command:

```
sbt "test:runMain CPUTopTester"
```

When the simulation is finished, you can examine the output in the terminal (created using 'prints' in the tester file) and the waveforms file `CPUTop.vcd` generated in the folder `.../generated/` by opening it with GTKWave.

We recommend you start by using small programs just to test that all the instructions are working properly and then you move to the erosion program you implemented and encoded in Task 3 and 4.

Task 11: Evaluate the implementation

Analyze the performance of the erosion step you compiled by hand in terms of clock cycles and execution time (assume a CPU frequency of 64 MHz, 500 MHz,

1 GHz and 2 GHz) and compare it to the PC implementation of Assignment 1. In addition, try to estimate the size of your CPU. Consider the following:

- What implementation is faster?
- Is it fair to compare your CPU with the one of your PC? What is different?
- How many clock cycles would it take to run the erosion for a full 950-by-950 pixels image (estimation)?
- What is the size of your CPU (estimation)? How may bits of registers, multiplexers of n bits, gates, etc.?

Task 12: Prepare the report

This is the task where you finalize the report. Use the report requirements available in Section 5.

5 Time management, report requirements, and evaluation

Similarly to Assignment 1, in order for you to check if you are on track or behind, we provide the following plan with reference to the tasks.

Date	**Expected task completion**
4$^{\text{th}}$ October	Completed: T1, T2, - Work on: T1, T2, T3, T4
11$^{\text{th}}$ October	Completed: T3, T4 - Work on: T5, T6, T7, T8, T9
25$^{\text{th}}$ October	Completed: T5, T6, T7, T8, T9 - Work on: T9, T10
1$^{\text{st}}$ November	Completed: T9, T10 - Work on: T11, T12
5$^{\text{th}}$ November	Completed: T11, T12

The short report should not be longer than 10 pages (everything included) and a mandatory template for the report is available in DTU-Learn. Similar to Assignment 1, write "directly-to-the-point", without re-explaining and presenting the basic ideas already presented in this document. Use the report to explain, justify, and validate your design decisions. Do not include the full code in the report, but you can include some code snippets if these are relevant to explain certain aspects of the implementation.

Your work will be evaluated according to the following task-based criteria on a scale of 100 points.

Achievement	**Points**
Task 2: Implement the ProgramCounter module	5
Task 3: Define the ISA and compile the program	10
Task 4: Encode the instructions	5
Task 5: Draft the CPU architecture	15
Task 6: Implement the ALU module	5
Task 7: Implement the RegisterFile module	5
Task 8: Implement the ControlUnit module	5
Task 9: Put everything together in the CPUTop module	15
Task 10: Run, test, and debug your program	25
Task 11: Evaluate the implementation	5
Report	5
Total	100

Appendix A:
Example of a small instruction-set architecture

This appendix presents a small ISA you can use to get inspired to implement your own instruction set. The ISA is listed in Table 5 and it is divided into 3 sections: arithmetic and logic instructions, data transfer instructions, control and flow instructions.

The arithmetic and logic instructions are used to perform operations between the content of the register file. The data transfer instructions are used to move data from memory into the registers and vice-versa. The control and flow instruction are used to control the execution of the program by performing 'jumps' between instructions if certain conditions are met.

Each instruction consists of an OPCODE and a list of operands. The OPCODE specifies the operation to be executed using the operands. Here we use `R1`, `R2`, and `R3` as operands, in general they could be replace by any register `Rn`

Table 1: The complete instruction-set architecture for the simulator.

Instruction	**Syntax (example)**	**Meaning (example)**
Arithmetic and logic instructions		
Addition	`ADD R1, R2, R3;`	`R1 = R2 + R3`
Subtraction	`SUB R1, R2, R3;`	`R1 = R2 - R3`
Multiplication	`MULT R1, R2, R3;`	`R1 = R2 * R3`
Immediate Add.	`ADDI R1, R2, 4;`	`R1 = R2 + 4`
Immediate Sub.	`SUBI R1, R2, 5;`	`R1 = R2 - 5`
Bitwise OR	`OR R1, R2, R3;`	`R1 = R2 or R3`
Bitwise AND	`AND R1, R2, R3;`	`R1 = R2 and R3`
Bitwise NOT	`NOT R1, R2;`	`R1 = not(R2)`
Data transfer instructions		
Load immediate	`LI R1, 6;`	`R1 = 6`
Load data	`LD R1, R2;`	`R1 = memory(R2)`
Store data	`SD R1, R2;`	`memory(R2) = R1`
Control and flow instructions		
Jump	`JR 7;`	`goto inst. 7`
Jump if equal	`JEQ 8, R2, R3;`	`if(R2==R3) goto inst. 8`
Jump if less than	`JLT 9, R2, R3;`	`if(R2<R3) goto inst. 9`
Jump if greater than	`JLT 10, R2, R3;`	`if(R2>R3) goto inst. 10`
No operation	`NOP;`	`do nothing`
End execution	`END;`	`terminates execution`