Memories

Learning Goal: Memories in VHDL and on Gecko4Education EPFL-Edition.

Requirements: Quartus II Web Edition and ModelSim-Intel.

1 Introduction

In this lab, you will learn how to implement memories in VHDL and how to instantiate FPGA memory components. The components you build here will be reused in the project of designing a system with a full processor. You will implement three different memories: a **Register File**, a **ROM**, and a **RAM**.

The **ROM** and the **RAM** are *large* and *slow* memories external to the processor, whereas the **Register File** is a *small* and *fast* data storage for use within the processor itself.

2 32-bit Register File

The first memory component that you will implement for this lab is a **Register File**, which you will also use in the CPU that you will implement in the following weeks.

Figure 1 shows the entity of the **Register File**. This **Register File** has 32 registers of 32 bits. Its first register (i.e., the register at address 0) has a fixed value of 0.

Figure 1: Entity of the **Register File**.

2.1 Reading the Register File

For this **Register File**, the read process is *asynchronous*. The inputs **aa** and **ab** select which two registers to read. Their values are sent on the outputs **a** and **b**, respectively. The diagram in Figure 2 illustrates a typical read process.

2.2 Writing to the Register File

The write process is *synchronous*. Input **wren** enables writing **wrdata** in the register addressed by **aw**. Writing a value in the register at address 0 has no effect–its value is fixed to 0.

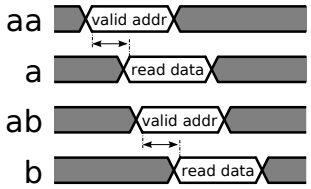


Figure 2: A read process of the **Register File**.

The diagram of Figure 3 illustrates a typical write process. It has essentially the same behaviour as flip-flops. The address, the data and **wren** are set during cycle 0. At the rising edge of the clock, the data is saved in the **Register File** at address **aw**. A new writing process can start during cycle 1.

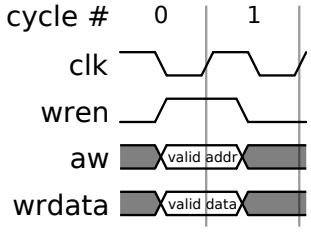


Figure 3: A write process of the **Register File**.

2.3 Exercise

Implement the **Register File** described in Section 2. For that, you can use an array of `std_logic_vector`. The following code gives an example of an array declaration.

```
architecture synth of register_file is
 type reg_type is array(0 to 31) of std_logic_vector(31 downto 0);
 signal reg: reg_type;
 ...
```

To read a value in an array, specify the index between parentheses exactly as you do when you select one bit in a `std_logic_vector`. The index can only be an integer, thus you may need the `to_integer(unsigned())` function that converts a `std_logic_vector` to an `integer`.

```
 data <= reg(to_integer(unsigned(address)));
```

- Open the provided **quartus** project.
- Implement the **Register File** in the file named `register_file.vhd`.
  - Do not forget that register 0 must have a fixed value of 0.
- To simulate the **Register File** using ModelSim, use the provided testbench named `tb_register_file.vhd`. Start with the given partial code (in the template) and complete it. You must instantiate your **Register File** component. This testbench template writes values in all the registers using a loop. Add the necessary code to read back and verify the written values.
- Simulate your **Register File** using your modified testbench.

3 Synchronous Memories

In this section, you will implement a simple system with *synchronous* memories. The FPGA on the Gecko4Education EPFL-Edition board has several *synchronous* memory blocks (SRAM) that we will use to create a **RAM** and a **ROM**. The schematic in Figure 4 illustrates this simple system.

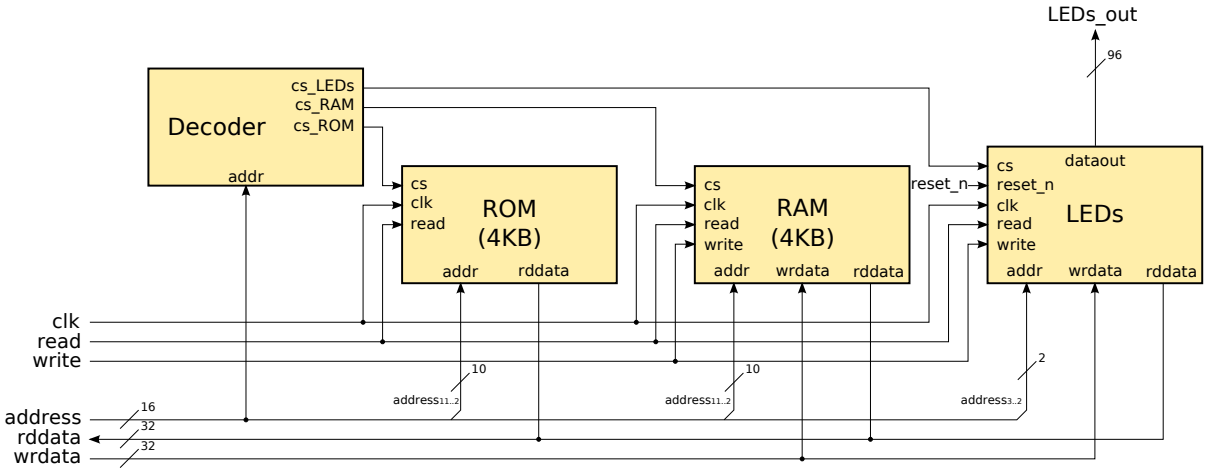
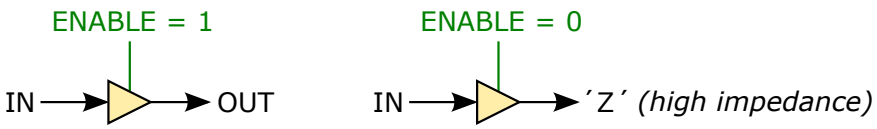


Figure 4: A schema of a system with synchronous memories and a LEDs module.

The **rddata**, **wrdata**, **read** and **write** signals of this system are shared by all the slave modules (i.e., the memories and the LEDs). The read and write processes are initiated with the **read** and **write** signals, respectively. Only one slave module can send data on the **rddata** bus at a time. Therefore, the slave modules must have tri-state buffers on their outputs.

> Tri-state buffers are necessary when multiple signal outputs (drivers) are connected to a common bus. Their role is to allow only one driver to connect to the bus at a time. Tri-state buffers are controlled by *enable* signals. When the enable signal is active, the tri-state buffer acts as a simple buffer: it passes the input signal to the output. When the enable signal is inactive, the tri-state buffer acts like an open switch (input is disconnected from the output). In VHDL terminology, disconnected output is represented by the high impedance type called `Z`.
>
> 
>

The selection of the module is done by the **Decoder**, which looks at the global address and activates the **cs** signal (*chip select*) of the corresponding slave module.

The SRAM blocks that we will use on the FPGA have specific read and write timings. In order to be compatible with this SRAM interface, the bus of the system has been designed and its protocol is described in the following subsections.

3.1 Read Process

The read process of an SRAM is *synchronous* and has a latency of one cycle. The timing diagram in Figure 5 illustrates a typical read process on the bus. During cycle 0, a valid address is provided and the **read** signal is set to 1. The **Decoder** raises the **cs** signal, which allows the selected module to send the read data during the next cycle. At the rising edge of the clock, the selected module must save the address, and the **read** and **cs** signals. During cycle 1, the registered address selects the corresponding

line in the memory to be outputted on **rddata**. The saved values of **cs** and **read** enable the tri-state buffer's output. Another read process may start during cycle 1.

In a *Synchronous* read-memory, when read address is provided at cycle n, the data will be *valid* for reading at cycle $n + m$, where $m \geq 1$ is the latency of read. Contrarily, in an asynchronous read process, when address is provided, data is *immediately* (i.e., in the same cycle) valid. Asynchronous memories are only used when the memory is very small (e.g., a register file), because the address decoding is cheap enough (in terms of critical path delay) can be done within a fraction of a cycle, With larger memories, the address decoding circuit is usually broken down into smaller stages connected by registers. Therefore, more cycles are required to to perform the read operation. In fact, the larger the memories are, the bigger m becomes, because more stages are address decoding are required.

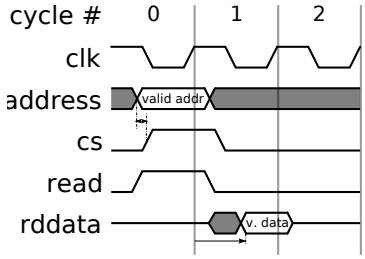


Figure 5: A synchronous read process on the system bus.

3.2 Write Process

The write process of the SRAMs is *synchronous* and has no latency cycles. The diagram in Figure 6 illustrates a typical write process on the bus. It is very similar to a write in the **Register File**. The **address**, **wrdata** and **write** signals are set during cycle 0. The **Decoder** provides the **cs** signal. At the rising edge of the clock, the data is saved by the module. A new writing process can start during cycle 1.

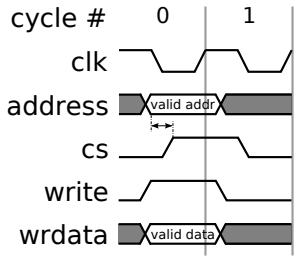
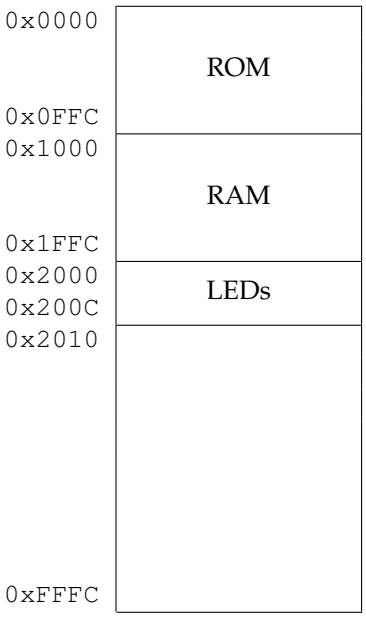


Figure 6: A synchronous write process on the system bus.

3.3 Decoder

The **Decoder** activates one of the slave modules (i.e., the **RAM**, the **ROM** or the **LEDs**) at a time, depending on the **address** value and the address space definition (refer to the next figure). For example, based on the address space below, the address `0x10F0` selects the RAM. In this case, the **Decoder** will activate the **cs_RAM** signal.

Remember that at most one **cs** signal can be activated at a time.



3.4 ROM and RAM Description

The *synchronous* **ROM** and **RAM** (both with read latency of 1 cycle) have each a size of 4KiB. The *synchronous* ROM (*Read Only* Memory) is a non-volatile memory. Our ROM.vhd file can read from ROM_Block.vhd, whose content will be initialized using Intel's memory block generation tool in Quartus. The *synchronous* RAM (Random Access Memory) is a volatile memory. In VHDL, you can represent this memory using an *array*. You can read from it and write to it using different processes depending on which signals in **cs**, **read** and **write** are set. These memories are word aligned (one word equals 4B), therefore, the 2 least significant bits of the address are ignored. There is also a tri-state buffer on their **rddata** output.

The file ROM_block.vhd implements a synchronous ROM with 1-cycle read and write latency. However, this module can not be readily connected to the other components depicted in Figure 4 and its data output `q` needs to passed through a tristate buffer. rest of the system. To do so, instantiate `ROM_BlocK` in the `ROM` (found under ROM.vhd) and enrich its interface with a chip select `cs` and read enable `read`.

3.5 LEDs Description

The **LEDs** module interfaces with the LEDs of the Gecko4Education EPFL-Edition. The output **out_LEDs** is directly connected to the 96 LEDs. The brightness of the LEDs is controlled by an 8-bit register, accessed only when the address is 11. The brightness scales linearly with the value in the 8-bit register and, at reset, it is initialized to the maximum possible value (255).

The **LEDs** module is word aligned, therefore, the 2 least significant bits of the address are ignored (you can assume they are always 0). Writing to **LEDs** modifies the value of one of its three 32-bit internal registers, which are connected to the **out_LEDs** output. The address specifies which of these registers is selected (see Table 1).

The read process has also one cycle latency; this ensures compatibility among the different components connected to the same bus. A read returns the current value of the corresponding register, that is the value currently displayed on the LEDs. Like the other modules, there must be a tri-state buffer on the **rddata** output.

3.6 Exercise

- Open the Quartus project.

Table 1: Addresses of the LEDs module.

Address	LEDs	Brightness
00	31..0	-
01	63..32	-
10	95..64	-
11	-	7..0

- The system structure is already defined in the project's `bdf` file (`GECKO.bdf`).
- Implement the decoder in the file named `decoder.vhd`.
- Use the LEDs file provided in the template.
- Implement the **RAM** in the file named `RAM.vhd`.
- Implement the **ROM** using the **ROM_Block** provided in the template. Please refer to section 3.6.1 for step-by-step guidelines on how this provided **ROM_Block** file **was** generated using the Intel memory block generation tool. The **ROM** will not be evaluated during this lab but it will be needed (and graded) during the next lab.
- When every component is implemented, use the `tb_memories.vhd` testbench to simulate them.
- The testbench can not verify the correctness of the access to the **ROM** during the simulation. You will have to verify it manually. Do not forget to include all the **vhdl** and **testbench** files in your modelsim project.

3.6.1 Synchronous ROM

For the **ROM**, we will use the Intel's memory block generation tool. This allows us to initialize the content of the **ROM** with an external file.

First, we create a memory initialization file.

- In Quartus, create a new file. Select the **Memory Files** category and select **Hexadecimal (Intel-Format) File**.
- Enter some values.
- Save this file as `ROM.hex` in your **quartus** folder (`quartus/ROM.hex`) and make sure that the file has been added to the project.

Then, we use Intel's memory block generation tool to create the **ROM**.

- If the IP Catalog is not already shown on the right of Quartus' main window, select Tools > IP Catalog
- In the IP Catalog, select Library > Basic Functions > On Chip Memory > ROM: 1-PORT.
- In the filename field enter `../vhdl/ROM_Block.vhd`. Use the file browser ('...' icon) to point towards the **vhdl** folder. Do NOT enter the file path by hand. Click **OK**.
- Set the width of the `'q'` output to 32 bits and the number of words in the memory to 1024. Click **Next**.
- Uncheck `'q'` output port. Click **Next**.

- Enter `../quartus/ROM.hex` in the **File name** field. Click **Finish**.
- When asked what additional files should be generated, uncheck `ROM_Block.cmp`. Click **Finish** again.
- When asked if the Quartus II IP file should be added to the project, select **No**.

The generated **ROM Block** can not be directly plugged on the bus: a tri-state buffer is still required on its **rddata** output. You will use the `ROM.vhd` file to encapsulate the **ROM Block** and include a tri-state buffer on its output.

4 Submission

Submit all vhdl files required by Jenkins (Controller.vhd, decoder.vhd, RAM.vhd, register_file.vhd). If you did not do the *Optional* Controller you can submit the template file. Once you submit the files, you will receive a report describing the tests that were applied to your design and the results of those tests (success or failure). Additionally, you will receive a score. However, this score will **NOT** be taken into account for the grade on the first project.

5 [Optional] Sequential Design with Memories

In this section, you will add a **Controller** to the system implemented in the previous section. This master module will copy data from one location to another in the memory space. To do so, it will read instructions from the **ROM**, which includes the length, the source address and the destination address information. Figure 7 shows the entity of the **Controller**.

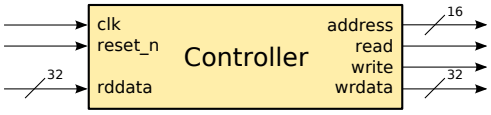


Figure 7: Entity of the **Controller**.

5.1 Controller Description

The **Controller** starts by reading two words from the **ROM**, starting from address 0. The first word contains the length of the transfer (i.e., the number of words to copy). The second word contains the source and the destination addresses (refer to Table 2 for the format). During the transfer, the **Controller** copies the number of words defined by **Length** from the **Source** address to the **Destination** address. When a transfer is finished, the **Controller** reads the next instruction from the **ROM**. It repeats the process until it finds a length of 0.

Table 2: Format of the **Controller**'s words.

Word #	bits 31 .. 16	bits 15 .. 0
1	-	Length
2	Source	Destination

5.2 Exercise

In this exercise, you will implement the **Controller**.

- Open the Quartus project.
- In VHDL, implement the **Controller** in the file named `controller.vhd`. The subsection 5.3 describes a design example of the **Controller**. If you have the time, we suggest that you do not read this description and design the **Controller** on your own.
- Synthesize the project and correct any errors. Look at the resource usage in the compilation report. If less than 32,768 memory bits are used, you probably made a mistake implementing the read process in the **RAM**, or are never accessing the **RAM** from the **Controller**. Look carefully at the warning messages, fix the error and retry.

Before testing you system, you must initialize the **ROM** content with valid instructions. A **ROM** is initialized with instructions by using a `ROM.hex` file. We already provide you with a `ROM.hex` file which contains valid instructions.

- Download the design on your FPGA. A pattern should appear on the LEDs. (If not, verify your code.) You can use the `tb_controller.vhd` testbench provided in the template for the simulation. It only generates a clock signal and does not verify the correctness of your code.

- Modify the instructions in the `ROM.hex` file to copy some data from the **ROM** to the **RAM** and then, from the **RAM** to the **LEDs**. Please refer to section 5.2.1 for step-by-step guidelines on how to modify the `ROM.hex` file.

5.2.1 Editing ROM.hex

- In Quartus, open the `ROM.hex` file.
- Change the radix of the data to hexadecimal as shown on Figure 8.

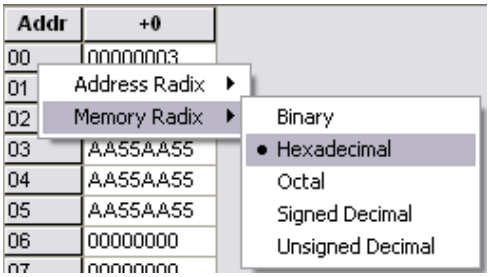


Figure 8: Change the radix of the ROM data to hexadecimal.

- Pay attention to the fact that the **ROM** content is word aligned: address `0x01` in the `ROM.hex` file corresponds to address `0x04` for the **Controller**.
- Fill the cells with some values. The `ROM.hex` file we provide already contains the values from Table 3. You can modify them using the Quartus editor.

Table 3: Data saved in the ROM.

Address	Data
0x00	0x00000003
0x01	0x000C2000
0x02	0x00000000
0x03	0xAA55AA55
0x04	0xAA55AA55
0x05	0xAA55AA55

5.3 Controller Implementation - Example

This is a fully detailed design example of the **Controller**. It includes a description of its state machine and internal registers. If you have the time, we suggest that you DO NOT read this section and design the **Controller** on your own.

In this implementation, the **Controller** uses the following internal 16-bit registers:

- **ROMaddr**, which holds the address of the next instruction in the **ROM**.
- **length**, which holds the number of remaining words to transfer.
- **rdaddr**, which holds the source address.
- **wraddr**, which holds the destination address.

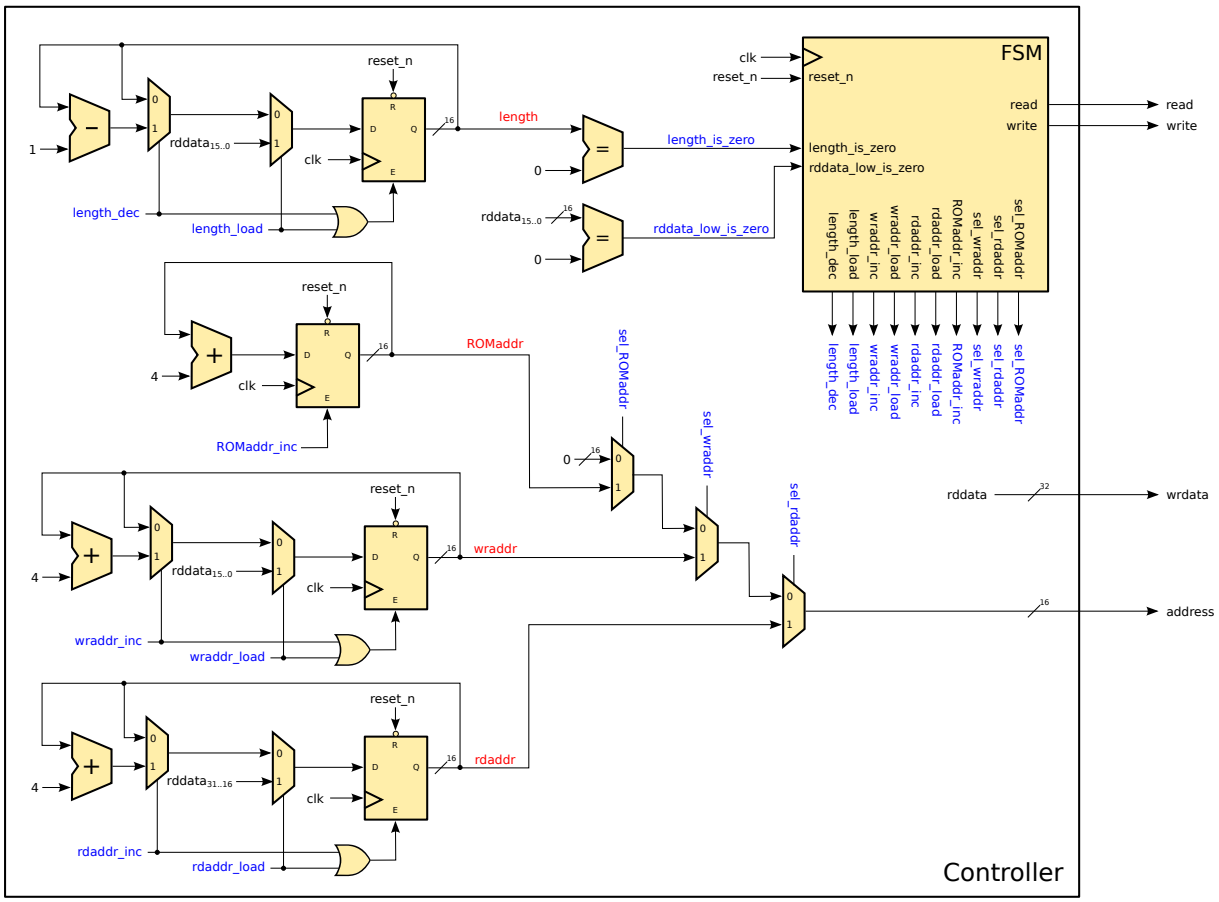


Figure 9: Architecture of the **Controller**. *Internal* control signals in blue, *internal* data signals in red, and *external* data and control signals in black.

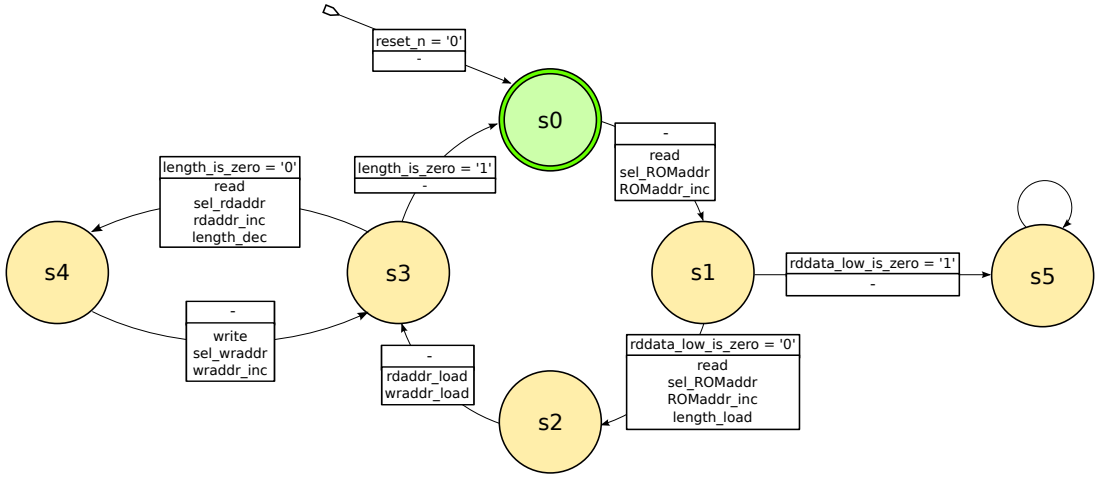


Figure 10: Finite state machine of the **Controller**.

Figure 9 shows the architecture of the **Controller**, and Figure 10 shows its finite state machine.

- The **reset_n** signal resets the state machine to state **S0** and all registers to 0.

- State **S0**: During this state, we start a read process to the **ROMaddr** address. Input **rddata** will contain the **length** of the current transfer during the next cycle. The next state is **S1**.
  - The **ROMaddr** register is incremented by 4.
- State **S1**: During this state, we start a read process of the **ROMaddr** address to read the source and destination addresses of the transfer. If the **rddata** value is zero, the length of the transfer is zero, which means that there is no more transfer remaining. In that case the next state is **S5**; otherwise the next state is **S2**.
  - The **ROMaddr** register is incremented by 4.
  - The **length** register takes the value of the 16 least significant bits of **rddata**.
- State **S2**: The next state is **S3**.
  - The **rdaddr** register takes the value of the 16 most significant bits of **rddata**.
  - The **wraddr** register takes the value of the 16 least significant bits of **rddata**.
- State **S3**: During this state, we read the **rdaddr** address to get the next word to copy. If the **length** register is 0, all the data of this transfer has been copied and we go to state **S0** to prepare for the next instruction; otherwise the next state is **S4**.
  - The **rdaddr** register is incremented by 4.
  - The **length** register is decremented by 1.
- State **S4**: During this state, we write the **rddata** value to the **wraddr** address. The next state is **S3**.
  - The **wraddr** register is incremented by 4.
- State **S5**: This state is a dead end, the next state is **S5**.

6 [Optional] Submission

To test the correctness of your controller, submit all vhdl files related to the exercises in sections 2.3, 3.6 and 5.2 Remember that the score you will receive will **NOT** be taken into account for the final score on the first project "Designing a Multicycle Processor".