Memories

Learning Goal: Memories in VHDL and on FPGA4U.

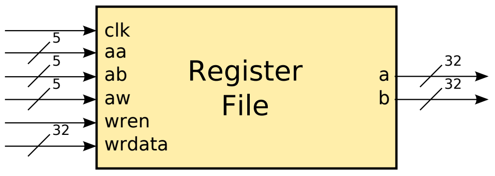
Requirements: Quartus II Web Edition and ModelSim-Altera.

1 Introduction

You will learn how to implement memories in VHDL, and how to instantiate FPGA memory components.

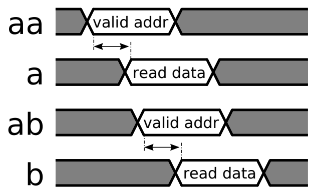
2 32-bit Register File

The first memory component that you will implement for this lab is a **Register File**. This **Register File** has 32 registers of 32 bits and its first register (i.e., at address 0) has a fixed value of 0. You will use this **Register File** in the CPU that you will implement later. The following figure illustrates the **Register File** entity.



2.1 Read in the Register File

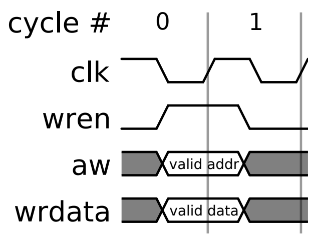
For this **Register File**, the read process is *asynchronous*. Inputs **aa** and **ab** select two registers to be read. Their values are sent on the outputs **a** and **b**, respectively. The following diagram illustrates a typical read process.



2.2 Write in 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.

The following diagram illustrates a typical write process. It has essentially the same behavior than 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.



2.3 Exercise

Implement the **Register File** described in the previous section. 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, you may need the `conv_integer()` function that converts a `std_logic_vector` to an `integer`:

```
  data <= reg(conv_integer(address));
```

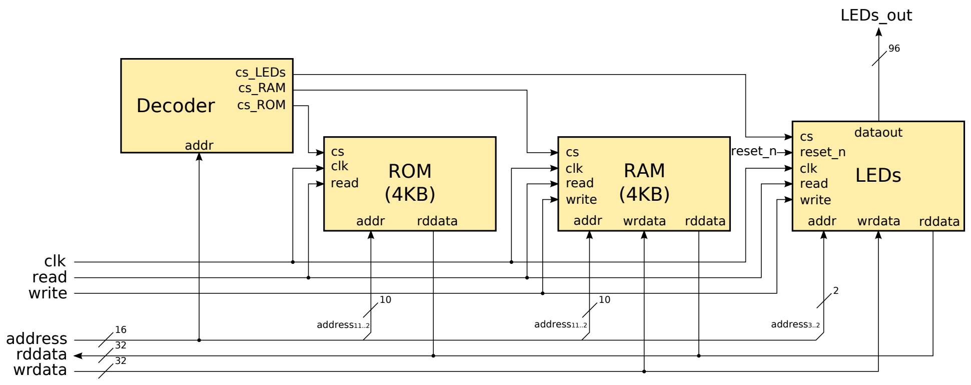
- Create a new `register_file` folder in you projects directory. Inside, create the usual **modelsim**, **quartus**, **vhdl** and **testbench** folders.
- Create a new `register_file` Quartus project.
- Implement the **Register File** in a file named `register_file.vhd`. It will be the top level entity of this Quartus project. Add the usual `ieee` libraries, the **Entity** and the **Architecture**.
  - Do not forget that the register 0 must have a fixed value of 0. Find a way to do this without using extra resources (e.g., a comparator).
- For the simulation with ModelSim, create a testbench in a file named `tb_register_file.vhd`. Save it in the **testbench** folder. You can start with the partial code given below and complete it. You must instantiate your **Register File** component. This testbench template writes values in all the registers by using a loop. Try to add the necessary code to read back and verify these written values.
- Simulate your **Register File** with the testbench.

Incomplete testbench:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity tb_register_file is end;

architecture bench of tb_register_file is
  -- insert the component declaration
  -- signals declaration
  signal aa, ab, aw    : std_logic_vector(4 downto 0);
  signal wren          : std_logic;
  signal a, b, wrdata : std_logic_vector(31 downto 0);
  -- clk initialization
  signal clk : std_logic := '0';
  -- clk period definition
  constant CLK_PERIOD : time := 40 ns;
begin
-- insert an instance of the register file

clock_gen: process
begin
  -- it only works if clk has been initialized
  clk <= not clk;
  wait for (CLK_PERIOD/2);
end process;

process
begin
  -- init
  wren <= '0';
  aa   <= "00000";
  ab   <= "00001";
  aw   <= "00000";
  wrdata <= (others => '0');
  -- insert a small delay
  wait for 5 ns;

  -- write in the register file
  wren <= '1';
  for i in 0 to 31 loop
    -- conv_std_logic_vector(value, bitwidth)
    aw     <= conv_std_logic_vector(i, 5);
    wrdata <= conv_std_logic_vector(i+1, 32);
    wait for CLK_PERIOD;
  end loop;

  -- complete to read in the registers
  wait;
end process;
end bench;
```

3 Synchronous memories

In this section, you must implement a simple system with *synchronous* memories and a module interfacing the LEDs. The FPGA on the FPGA4U board has several *synchronous* memory blocks (SRAM) that we will use to create a **RAM** and a **ROM**. The following schema illustrates this simple system.

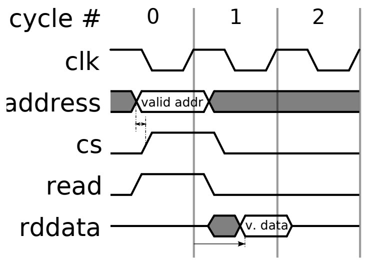


The **rddata**, **wrdata**, **read** and **write** signals of this system are shared by all the slave modules (i.e., the memories and the LEDs). The read and write processes are initiated with the **read** and **write** signals, respectively. Only one slave module can send data on the **rddata** bus at a time. Therefore, the slave modules must have a tri-state buffer on their output. The selection of the module is done by the **Decoder**, which looks at the global address and activate the **cs** signal (*chip select*) of the corresponding module.

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

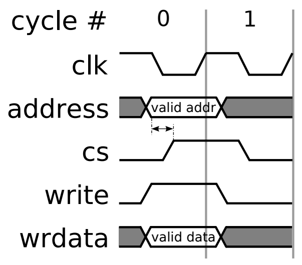
3.1 Read process

The read process of an SRAM is *synchronous* and has a latency of one cycle. The following timing diagram illustrates a typical read process on the bus. During cycle 0, a valid address is provided and **read** 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 **read** and **cs** signals, plus the address. 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.



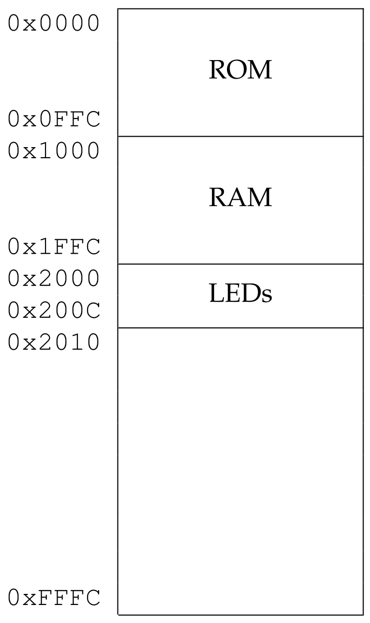
3.2 Write process

The write process of the SRAMs is *synchronous* and has not any cycle latency. The following diagram 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 the cycle 0. The **Decoder** provides the **cs** signal. At the rising edge of the clock, the data will be saved by the module. A new writing process can start during cycle 1.



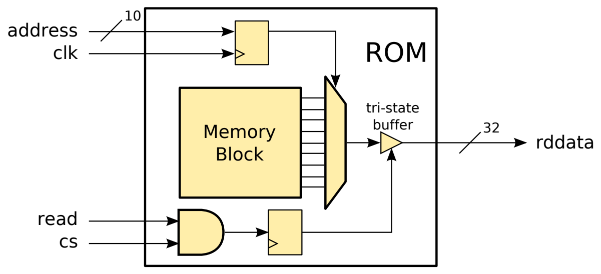
3.3 Decoder

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 (see the the next figure). For example, based on the address space below, the address `0x10F0` would select 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

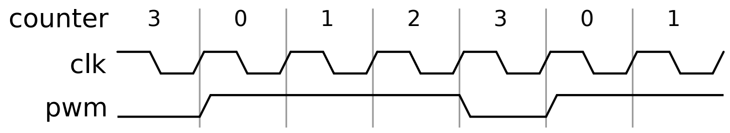
The *synchronous* **ROM** and **RAM** have each a size of 4KB. These memories are word aligned, therefore, the 2 least significant bits of the address are ignored. There is also a tri-state buffer on their **rddata** output as illustrated in the following **ROM** figure.



3.5 LEDs description

The **LEDs** module interfaces the LEDs of the FPGA4U. The **LEDs_out** output is directly connected to the 96 LEDs. The **LEDs** module is word aligned, therefore, the 2 least significant bits of the address are ignored. Writing to **LEDs** will modify the value of one of its three 32-bit internal registers, which are connected to the **LEDs_out** output. The address selects which of these register is selected (see next table).

The address can also select an 8-bit register which is used to generate a **PWM** (Pulse-width modulation) to control the luminosity of the LEDs. The figure below shows how a 2-bit register set with a value of 3 can be used to control the generation of the **PWM** signal. As seen in the figure, when the counter equals 0, the **PWM** goes high and when it reaches his threshold it goes low until the counter overflows. This **PWM** signal can now be used to drive the **LEDs** to control its luminosity. Note when setting the threshold value that the internal-counter will be reset at 0.



The read process has also one cycle latency; this ensures compatibility among the different components connected to the same bus. A read will return 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.

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

3.6 Exercise

- Download the project template.
- Open the `memories` Quartus project.
- The system structure is already defined in the `bdf` file.
- For the simulation, you will have to create a modelsim project. When every component is implemented, you can use the provided `tb_memories.vhd` testbench, which verifies automatically the reads and writes to the RAM and the LEDs. The testbench can't verify the correctness of the access to the ROM during the simulation. You will have to verify it manually. Do not forget to generate the VHDL file from the `.bdf` file and to include all the files in your modelsim project.
- Implement the decoder.
- Implement the LEDs.
- Implement the RAM. Compile the Quartus project and 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. Look carefully at the warning messages, fix the error and retry.

3.6.1 Synchronous ROM

For the **ROM**, we will use the Altera's memory block generation tool. Doing this will allow you to initialize the content of the **ROM** with an external file.

- In the menu, select Tools > MegaWizard Plug-In Manager...
- Click **Next**.
- In the megafunction list, select Installed > Memory Compiler > ROM: 1-PORT.
- In the filename field enter `../vhdl/ROM_Block.vhd`. Click **Next**.
- Set the signal width to 32 bits and the number of words in the memory to 1024. Check `M4K` and click **Next**.
- Uncheck `'q' output port`. Click **Next**.
- Enter `../quartus/ROM.hex` in the **File name** field. Click **Finish**.

Now we will create the initialization file.

- In Quartus, create a new file. Select the **Other Files** tab and select **Hexadecimal (Intel-Format) File**.
- Enter some values.
- Save this file to `ROM.hex` and make sure that the file has been added to the project.

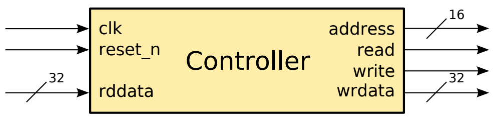
The generated **ROM Block** cannot be plugged directly on the bus: a tri-state buffer is still required on its **readdata** output. You will use the `ROM.vhd` file to encapsulate the **ROM Block** and include a tri-state buffer on its output.

The following piece of code is the **ROM block** component declaration:

```
component ROM_Block is
  port(
    address : in  std_logic_vector(9 downto 0);
    clock   : in  std_logic;
    q       : out std_logic_vector(31 downto 0)
  );
end component;
```

4 Sequential Design with Memories

In this section, you will add a **Controller** into the system that you implemented in the last section. This master module will copy data from one location to another in the memory space. To do that, it will read instructions in the **ROM**, which will include the length, the source address and the destination address information.



4.1 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 the following table 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 in the **ROM**. It repeats the process until it finds a length of 0.

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

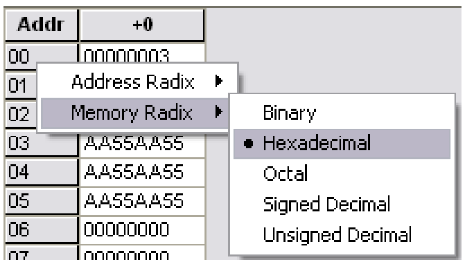
4.2 Exercise

In this exercise, you will implement the **Controller** and add it into the system that you implemented in the last section.

- Open the `memories` Quartus project.
- Create a new file named `controller.vhd` for your **Controller**.
- In VHDL, implement the **Controller**. The subsection 4.3 describes a design example of the **Controller**. If you have the time, we suggest you to not read this description and design the **Controller** by yourself.
- Generate the block diagram of the **Controller**. In the schematic file of the system, remove the **rddata**, **wrdata**, **write**, **read** and **address** pins. These signals are now provided by the **Controller**. Add the **Controller** block and connect it correctly to the system.
- Synthesize the project, and correct errors.

Before testing you system, you must initialize the **ROM** content with valid instructions.

- In Quartus, open the `ROM.hex` file.
- Change the radix of the data to hexadecimal.



Addr	+0
00	00000003
01	
02	
03	AA55AA55
04	AA55AA55
05	AA55AA55
06	00000000
07	00000000

- 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 the following example. Try to understand it.

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

The last step before downloading your design onto the FPGA is to set the pin assignment.

- In Quartus, open the Tcl Console and run *source pin_assignment.tcl*.
- Compile the Quartus project and download your design onto the FPGA. A pattern should appear on the LEDs. If it does not occur, verify your code. You can use the `tb_controler.vhd` testbench provided in the template for simulation. It only generates a clock signal and does not verify the correctness of your code. Do not forget to regenerate the VHDL file from the `.bdf` file.
- 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**.

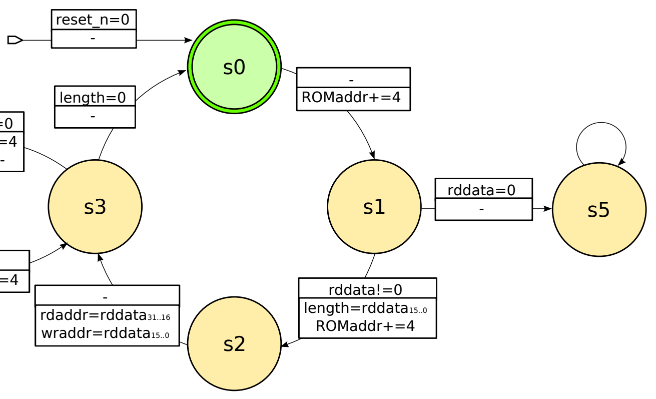
4.3 Controller example of implementation

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 you to NOT read this section and design the **Controller** by yourself.

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.

The following figure is the state machine of the **Controller**:



- The **reset_n** signal resets the state machine to state **S0** and the **ROMaddr** register 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 to 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 null, 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**.

5 Submission

Submit all vhdl files related to the exercises in sections 2.3, 3.6 and 4.2 (`LEDs.vhd`, `RAM.vhd`, `ROM.vhd`, `controller.vhd`, `decoder.vhd` and `register_file.vhd`)