

## Laboratory 3

### 3. Memory Components

#### 3.1 Objectives

Design, implement and test

- **Register File**
- **Read only Memories – ROMs**
- **Random Access Memories – RAMs**

Familiarize the students with

- Xilinx® ISE WebPack
- Xilinx® Synthesis Technology (XST) **XST User Guide**
  - **Chapter 2: XST HDL Coding Techniques**
  - **Chapter 6: XST VHDL Language Support**
- Digilent Development Boards (**DDB**)
  - [Digilent Basys Board – Reference Manual](#)
  - [Digilent Basys 2 Board – Reference Manual](#)
  - [Digilent Basys 3 Board – Reference Manual](#)

#### 3.2 Theoretical Background

##### 3.2.1. Register File

The Register file is the central storage of a Microprocessor.

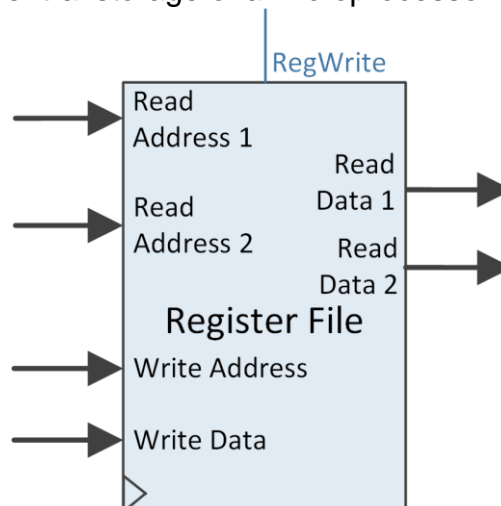


Figure 3-1: A Register File with 2 read ports and 1 write port

Most CPU operations involve using or modifying data stored in the register file. Since the register file runs at the full speed of the processor, it must be small and fast. The real register file is usually implemented as a small, fast **SRAM** memory with multiple accesses.

A register file (specific for MIPS) has two read addresses and one write address. The registers corresponding to the locations indicated by the two read addresses (Read Address 1 & Read Address 2) are delivered at the two output ports (Read Data 1 & Read Data 2). The data provided at the write data input port is written in the register indicated by the write address (Write Address, when the Write control signal (**RegWrite**) is asserted. The **read** operations are **asynchronous**, while the **write** operation is **synchronous**. Therefore, the register file supports 2 reads and one write in each clock cycle.

Appendix 4 presents a possible register file VHDL implementation.

### 3.2.2. ROMs and RAMs

Read-only memory (ROM) is a class of storage media used in computers and other electronic devices; they allow only read operations in usual operation mode. Random-access memories (RAM) are a form of computer data storage implemented as integrated circuits and allows the stored data to be accessed in any order; both read and write operations are permitted. These two memory types are essential for any microprocessor.

An FPGA device comes equipped with a certain amount of BRAM (Block RAM). The BRAM can be configured as either a ROM or a RAM. Depending on how you write the VHDL code, XST either can infer your RAM design as a distributed memory or directly mapped onto a Block RAM block. Distributed memories are built with registers, while Block RAM memories are mapped to available BRAM cell. Distributed RAMs occupy more space inside the FPGA and usually decrease the clock cycle rate, while BRAMs provide more space for auxiliary logic inside the FPGA. The type of inferred RAM depends on its description:

- RAM descriptions with an asynchronous read generate a distributed RAM macro.
- RAM descriptions with a synchronous read generate a block RAM macro. See appendix 5 for an implementation example.

XST covers the following RAM characteristics:

- Synchronous write
- Write enable
- RAM enable
- Asynchronous or synchronous read
- Reset of the data output latches
- Data output reset

- Single, dual or multiple-port read
- Single-port/Dual-port write
- Parity bits
- Block RAM with Byte-Wide Write Enable
- Simple dual-port BRAM

There are three possible modes of implementing a synchronous RAM [1], [2]: write-first, read-first and no change. These modes are reflected in the behavioral description of the RAM (VHDL code) regarding the read/write priority or order of operation. A possible “no change” implementation is presented in appendix 5.

### To Do

- Use the language templates: VHDL → Synthesis Constructs → Coding Examples → RAM and see the differences in behavioral description between a distributed RAM and a Block RAM.
- Use the language templates: VHDL → Synthesis Constructs → Coding Examples → RAM → Block RAM → Single port in order to compare the read-first and write-first implementations.

#### 3.2.3. Declaring an array in VHDL

An example of declaration and initialization for an array used in ROMs, RAMs and Register Files is presented below. First, we describe an array type having N locations of M bits each:

```
type <arr_type> is array (0 to N-1) of std_logic_vector(M-1 downto 0);
```

Next, we declare a signal of the same type as the previously declared one:

```
signal r_name: <arr_type>;
```

If one implements a ROM then the signal must be initialized. Initializations for the RAMs and Register File are also possible.

```
signal r_name: <arr_type> := (
  "00...0", -- M bits, use hexadecimal representation when possible
  "00...1", --
  others => "00...0"
);
```

## 3.3 Laboratory Assignments

At this time, it is mandatory to have a functional “*test\_env*” project that resembles with the description from the previous laboratory (laboratory 2: section 3 → figure 3). Your design must contain the Mono Pulse Generator (MPG) and 4-digit Seven Segment

Display (SSD) components instantiated in the top-level entity of your “*test\_env*” project; here you will write the code for this laboratory.

Use **View RTL Schematic** after each successful synthesis of your project. You can also view the implemented components in the **Design Summary** (see laboratory 1).

### 3.3.1. ROM Implementation

Include a 256 x 16 bits ROM memory in the *test\_env* project (do not declare a new entity). Initialize the ROM with some arbitrarily chosen values (see 2.3). Use an 8-bit counter to generate the addresses for the ROM. The counter is controlled by the MPG component. The content of the ROM is displayed on the Seven Segment Display. **The behavior of the ROM is asynchronous – use only one line of code.** The design is depicted in the figure below:

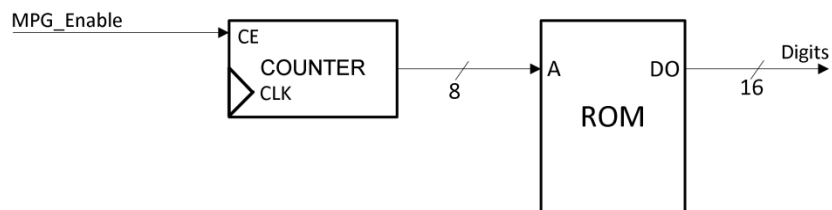


Figure 3-2: Simple ROM design

**Test on the development board!**

### 3.3.2. Register File Implementation

Do not delete the previously written code! Use comments if necessary!

Design and implement a 16x16-bit Register File on the development board (use a new component for the register file, in the “*test\_env*” project). Initialize the Register File with some values. The design is presented in Figure 3-3.

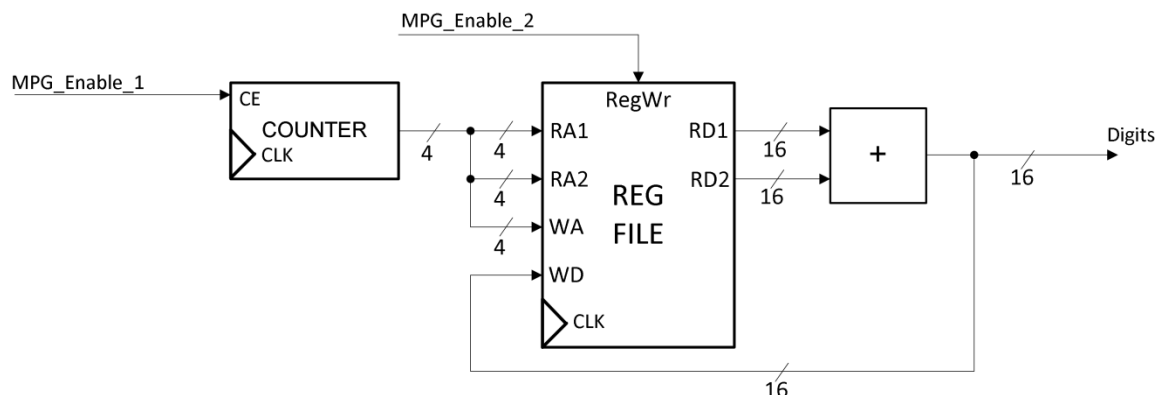


Figure 3-3: Simple Register File Design

Use a counter to generate the read and write address of the Register File. The counter is controlled by a MPG component. The outputs of the Register File are added together; the result of addition is displayed on the Seven Segment Display and written back to the Register File. You have to use another output of the MPG component to enable the write signal of the Register File (RegWr). The design should resemble a multiply by 2 circuit ( $a + a = 2a$ ).

Add a synchronous reset mechanism for the counter that generates the Register File's address such that after going through a few of the Register File's locations you can reset the address counter (return to address 0) and check that the written results in Register File are correct.

**Test on the development board!**

Add some auxiliary components/elements to the design in order to use only one button for the counter increment and RegWr signal. The new circuit should work in the same manner (the results should be the same as in the previous case) and you must see the correct results of addition on the seven-segment display.

You can add the necessary circuit in order to display all the intermediate signal values on the SSD (Hint: use switches).

**Test on the development board!**

### 3.3.3. RAM Design

Replace the Register File previously designed with a RAM memory. Use a shift left 2 operation instead of the addition (Figure 3-3, shift is implemented with concatenation). Use only one address for the RAM. Use a write-first mode of implementation.

## 3.4 References

- [1] XST User Guide, Chapter 2: XST HDL Coding Techniques
- [2] XST User Guide, Chapter 6: XST VHDL Language Support
- [3] XAPP463 (v2.0) March 1, 2005 Using Block RAM in Spartan-3 Generation FPGAs
- [4] Digilent Basys Board – Reference Manual
- [5] Digilent Basys 2 Board – Reference Manual
- [6] Digilent Basys 3 Board – Reference Manual