

Project 1: Designing a 16-bit Register File

Adam Sumner

ECE 485

October 6th, 2015

1 Introduction

The purpose of this project is to implement an 8 by 16-bit register file in the VHDL hardware description language. The register file should be able to output the contents of two registers, and it should be able to write into one of the registers at the positive edge of the clock if an external “registerWrite” signal is set to 1.

2 Design

2.1 Architecture

To enable the specific functionality requested in the project, the file needs to have certain attributes. In order to output two register values at the same time, it must have two outputs of 16 bits each: `outReg1` and `outReg2`. This solves how the file will display register values, but since there are 8 registers total, functionality must be included to select which two registers to show. This is why `inReg1` and `inReg2` are a necessity to include in the architecture. They are both three bit inputs to select which register number (in binary) to display. The schematic for this is shown in Figure 1. Now that an architecture has been built to select and output two register’s content, an architecture for writing contents into a register must also be included. Similar to the architecture for reading registers, a 16-bit data field for writing data into a register (`writeData`) and a three bit input (`writeReg`) is used to select which register to write the data to. The schematic for this implementation

is shown in Figure 2. With these both in place, our schematic now at its core looks like that of Figure 3. With the core architecture implemented, a `clk` and `writeEnable` must be included so that both a signal can be set to enable the write process, and also so that writing will only happen at the positive edge of the `clk`. All of the discussed attributes and architectures are declared in the entity section of Section 2.3.

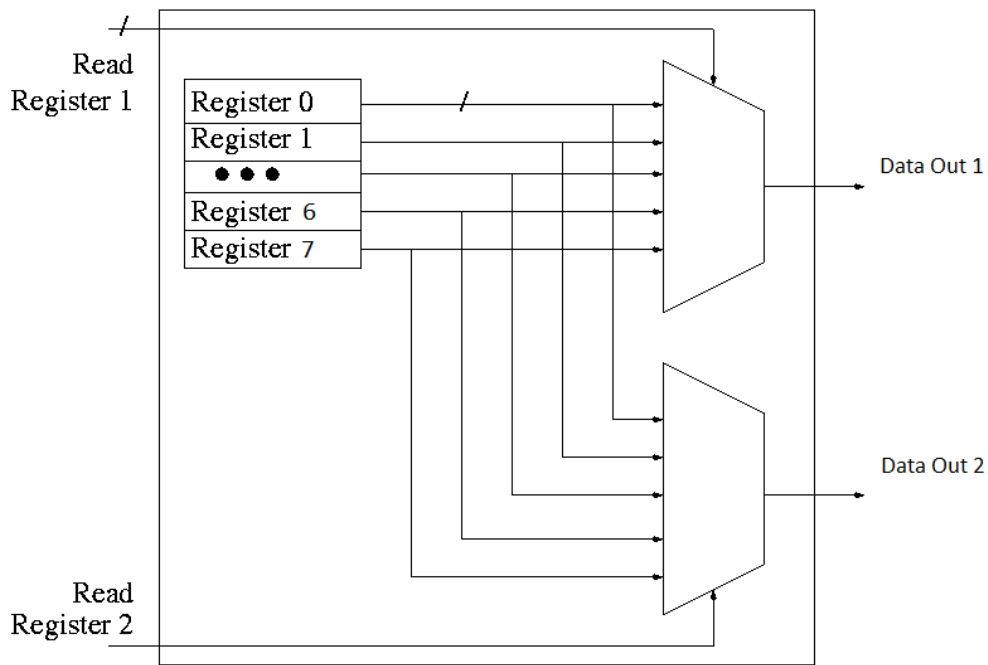


Figure 1: Schematic for Reading Two Registers

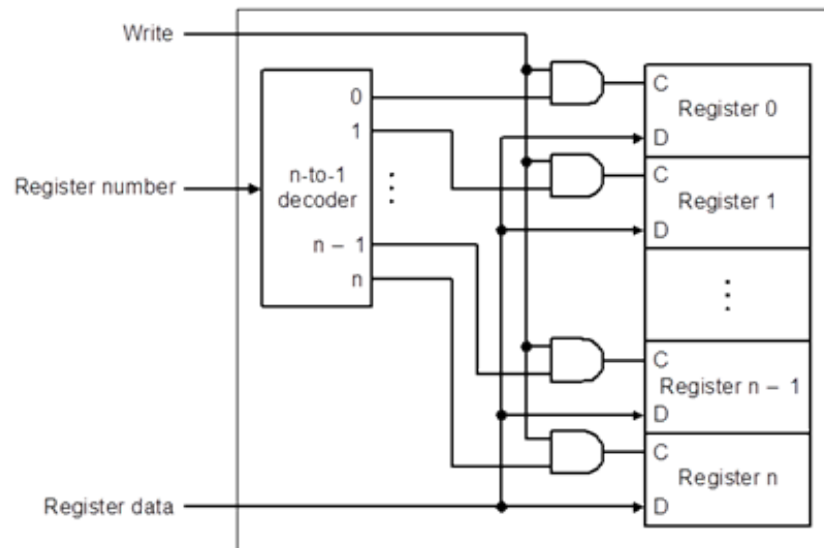


Figure 2: Schematic for n-register Write Functionality

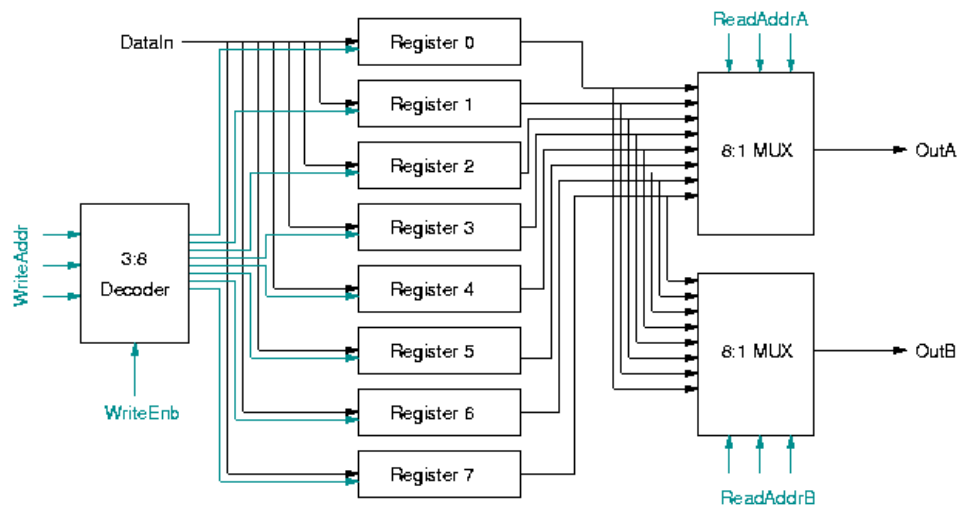


Figure 3: High Level Design Schematic

2.2 Behavior

The behavioral section of the code in Section 2.3 is pretty straightforward. An array of 8 16-bit data vectors is declared, and a process is defined for the `clk` signal. If the `clk` is at its rising edge, then the data from the selected two registers is put into the respective output field. If `writeEnable` is also 1 on the rising edge of `clk`, the the data held in `writeData` is put into the selected register determined from `writeReg`. Note that registers are only read on the rising edge of the `clk`. This was decided to make the design process simple and to define a standard that events will only execute during the start of a `clk` cycle.

2.3 Code

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity register_16 is
6     port(
7         outReg1      : out std_logic_vector(15 downto 0);
8         outReg2      : out std_logic_vector(15 downto 0);
9         writeReg      : in  std_logic_vector(2  downto 0);
10        writeData     : in  std_logic_vector(15 downto 0);
11        inReg1        : in  std_logic_vector(2  downto 0);
12        inReg2        : in  std_logic_vector(2  downto 0);
13        writeEnable    : in  std_logic;
14        clk           : in  std_logic
15    );
16 end register_16;
17
18
19 architecture behavioral of register_16 is
20     type registerFile is array(0 to 7) of std_logic_vector(15 downto 0);
21     signal registers : registerFile;
22 begin
23     regFile : process (clk) is
24     begin
25         if rising_edge(clk) then
```

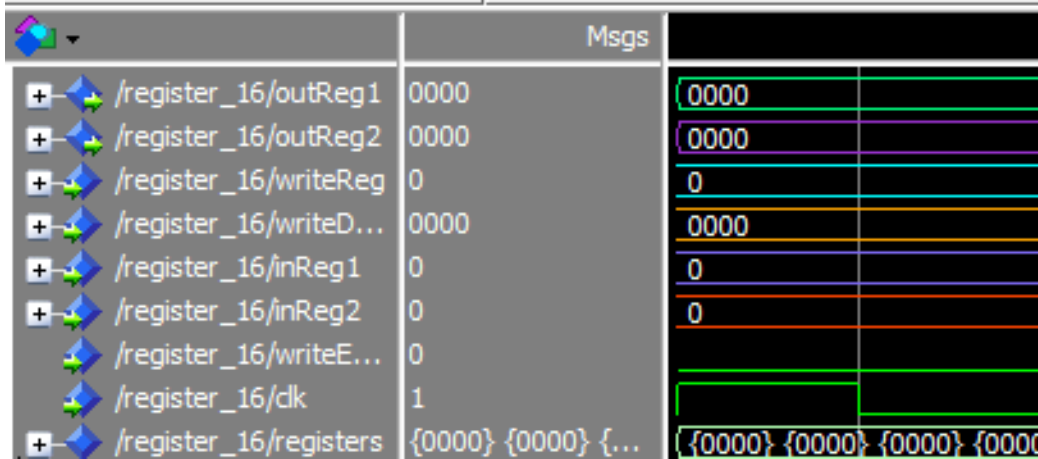
```

26     outReg1 <= registers(to_integer(unsigned(inReg1)));
27     outReg2 <= registers(to_integer(unsigned(inReg2)));
28     if writeEnable = '1' then
29         registers(to_integer(unsigned(writeReg))) <= writeData;
30     end if;
31 end if;
32 end process;
33 end behavioral;

```

3 Simulation Results

This section will walk through a test simulation to ensure the functionality of the register file is correct. All registers will start with value 0000000000000000_2 . Value 1000000000000000_2 will be written into register 2 and value 0000000000000001_2 will be written into register 7. These registers will then be read. A clock cycle of 10 ns will be used. The start of the simulation is shown in Figure 4.



	Msgs	
/register_16/outReg1	0000	0000
/register_16/outReg2	0000	0000
/register_16/writeReg	0	0
/register_16/writeD...	0000	0000
/register_16/inReg1	0	0
/register_16/inReg2	0	0
/register_16/writeE...	0	
/register_16/dk	1	
/register_16/registers	{0000} {0000} {...	{0000} {0000} {0000} {0000}

Figure 4: Start of Simulation

Then the value 1000000000000000_2 is put into `writeData`, the value 010_2 is put into `writeReg`, and the `writeEnable` is set. This is shown in Figure 5.

	Msgs	
+ /register_16/outReg1	0000	0000
+ /register_16/outReg2	0000	0000
+ /register_16/writeReg	2	2
+ /register_16/writeD...	8000	8000
+ /register_16/inReg1	0	0
+ /register_16/inReg2	0	0
+ /register_16/writeE...	0	
+ /register_16/dk	1	
- /register_16/registers	{0000} {0000} {...	{0000} {0000} {8000} {0000} {0000} {0000} {0000} {0000}
+ (0)	0000	0000
+ (1)	0000	0000
+ (2)	8000	8000
+ (3)	0000	0000
+ (4)	0000	0000
+ (5)	0000	0000
+ (6)	0000	0000
+ (7)	0000	0000

Figure 5: First Write

Then the value of 0000000000000001_2 is put into `writeData` and the value 111_2 is put into `writeReg`. This is shown in Figure 6.

	Msgs	
+ /register_16/outReg1	0000	0000
+ /register_16/outReg2	0000	0000
+ /register_16/writeReg	7	7
+ /register_16/writeD...	0001	0001
+ /register_16/inReg1	0	0
+ /register_16/inReg2	0	0
+ /register_16/writeE...	1	
+ /register_16/dk	1	
- /register_16/registers	{0000} {0000} {...	{0000} {0000} {8000} {0000} {0000} {0000} {0000} {0000}
+ (0)	0000	0000
+ (1)	0000	0000
+ (2)	8000	8000
+ (3)	0000	0000
+ (4)	0000	0000
+ (5)	0000	0000
+ (6)	0000	0000
+ (7)	0001	0001

Figure 6: Second Write

Now `writeEnable` is set to 0, `inReg1` is set to 010_2 and `inReg2` is set to 111_2 . This is shown in Figure 7.

Signal	Value
/register_16/outReg1	8000
/register_16/outReg2	0001
/register_16/writeReg	7
/register_16/writeD...	0001
/register_16/inReg1	2
/register_16/inReg2	7
/register_16/writeE...	0
/register_16/dk	1
/register_16/registers	{0000} {0000} {8000} {0000} {0000} {0000} {0000} {0001}

Figure 7: Reading Values from Registers

It is clear from the simulation results that the register is behaving as intended.

4 Conclusion

The construction of an 8 by 16-bit register file was a success. An architecture and behavior was designed around the register file constraints, and the final product performed as expected. This register file can now be used in further projects.