CMPE 260
Digital System Design II
Laboratory Exercise 2: Register File
Revised: 06-12-2016

This exercise investigates the design of a register file for an FPGA. The objective of this exercise is to implement, using VHDL, a digital system that is capable of storing multiple words, or groups of bits, as well as understanding how to properly test such a design. VHDL files are created and tested, both in simulation and in hardware.

Background

Computers need to be able to store and access information very quickly. However, simple digital systems like D flip flops, by themselves, cannot hold enough information to be useful. Thus register files are created that allow for one bus to access multiple locations, each which holds multiple bits. In this exercise, there are two outputs, which allows for two addresses to be read from at the same time, increasing the speed at which the necessary data can be obtained.

Prelab Work

- Draw a schematic of the register file following the specifications laid out below. It is not necessary to show the inner works of each component, just how they are tied together to create the register file.
- Review the use of generics in both entity declarations and instantiations.
- Print the sign off sheet.

Program Specification

The design must meet the following specifications and use all components listed at least once. Some will be used multiple times.

Register Module

The register module has the following features (see Figure 1):

- Parameters:
  - The size of the word to store, in bits. To be called `n`.
  - The value to which the register will reset to. To be called `r`.
- Inputs (One bit unless noted otherwise):

  - `in` (n-1 bits): Parallel input to be stored.
  - `clk`: Clock signal. Register file is rising edge triggered.
  - `we`: Write enable. Enabling allows register contents to be written.
  - `rst`: Resets the register to `r`. Asynchronous.

- Outputs:
  - `out` (n-1 bits): Parallel output of the register module.

Consider using the behavioral style for the register module.

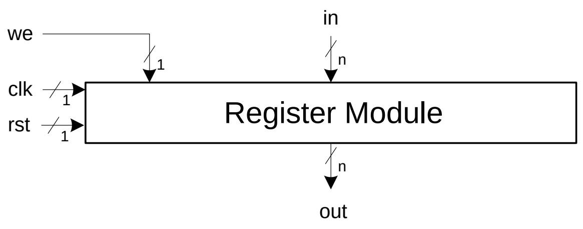

Figure 1: Register Module

Multiplexer

Create an 8-to-1 MUX, whose eight inputs are `n` bits wide.

Decoder

Create a 3-to-8 decoder. The input of this decoder should not be written as eight separate signals, but instead as one multi-bit signal. This is also true of the output.

Register File

The register file contains eight register modules and follows a two-read, one-write format. See Figure 2.

- Inputs (One bit unless noted otherwise):
  - `rd1, rd2` (3 bits): Read1 and read2. Selects which registers to read from.
  - `wr` (3 bits): Write. Selects which register to write to.
    Note that the previous signals are 3 bits long, allow them to address all eight registers.
  - `in` (16 bits): Data to be written to the appropriate register.
  - `clk`: Clock signal.
  - `we`: Write enable. Enabling allows register contents to be written.
  - `rst`: Resets all registers. Asynchronous.

- Outputs:
  - `out1, out2` (16 bits): Parallel outputs containing the data from the selected registers.

Hints

- Use the output of the 3-to-8 decoder in conjunction with the register file's `we` signal to drive the `we` on the register module.
- Use two 8-to-1 MUXs, one to connect the output of the register specified by `rd1` to `out1`, and a second for `rd2` and `out2`.
- Do not use parameters in the entity of the register file, however, generics should still be used when instantiating the various components. (Note: The reason for this is that the generic would then have to be set in a higher level, of which there are none.)

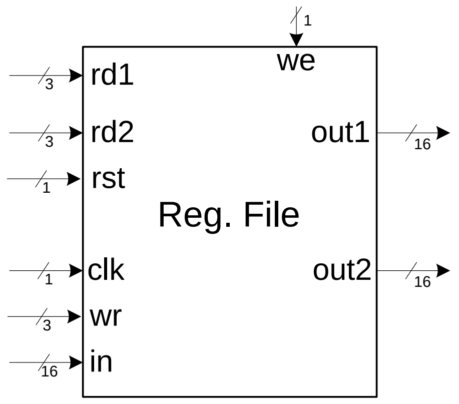

Figure 2: Register File

Testbench

Write a testbench to ensure that the register file is working properly. Some potential scenarios are listed:

- Resetting the register file
  - This should set all register modules to their reset value, `r`.
- Writing to each register module
  - A for loop can be used to write to every register module by modifying `wr`.
- Reading from each register module
  - Ensure that both `rd1` and `rd2` are tested.

Lab Procedure

1. Create a new directory and Xilinx ISE project for this exercise.
2. Write a properly commented and properly formatted VHDL program according to the preceding specifications.
3. Test the design using a behavioral simulation.
4. Synthesize the design and retest it using a post route simulation.
5. Prepare the design for hardware, following the procedure laid out in Exercise One.
6. Program the board and verify that it is working.
7. Demonstrate steps 3, 4, and 6 to the instructor, who will sign the grading sheet.
8. Screen shot the appropriate waveforms for the report, making sure that they will be readable.
9. If necessary, modify the code to acquire the timing results. See the section on Lab Reports for details.
10. Submit the source code and report online to the appropriate myCourses dropbox.

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab report format on myCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic.
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used.
- Timing results (best case achievable). In order to find this value, the post route report will have to be inspected. This can be found in "Your_top_level_entity.par" or by reading the console output. Should the "Best Case Achievable" value not appear, DFFs need to be added to the input and output, in this case `in`, `out1`, and `out2`. Without the addition of the flip flops, there is no critical path. Further details will be discussed in the timing section of the lecture.
- Marked simulation results.