ECEN 4243

Lab 1: Revisiting Digital Logic and Verilog Simulation

Moser, Christian A20173104 Cook, Isabell A20208072

January 29, 2022

1. Introduction

The purpose of this lab is to become familiar with using Verilog tools such as ModelSim by creating a 32-bit register file that can read up to two data ports combinatorially and write into the register file sequentially. This connects to the lecture material by showing basic functions of the SystemVerilog language and how to incorporate test benches into our designs, and how to implement them into ModelSim to virtually test designs before implementing them into hardware. We learned how to use ".do" and ".sv" files to properly run and test designs within ModelSim by first adding ModelSim to the PATH environment variable and configuring the .do file to immediately implement and test the hardware with a waveform output that could easily be understood to verify the specifications of the alternative design.

Our design correctly functions as specified by the baseline design. The design correctly writes to the specified location in the register file as per the definition of the destination addresses, and correctly reads in data from the source addresses to the data ports from the register file. The read operations are combinatorial and therefore blocking, while the write operations are sequential and therefore non-blocking. This connects to the ideas presented in lecture by showcasing a model operation of how these blocking and non-blocking concepts work.

The operations performed were executed immediately when required inputs were high and would not trigger when low. Each clock cycle operated at 10ns and we used a clock period of 5ns to test the register file. The register file in the implemented design had an immediate response of ¡ 1 ns as the impulse of the clock edge was enough to trigger both read and write operations to and from the register files.

2. Baseline Design

The baseline design is to create a register file that is able to execute one ARM instruction per cycle, which means that the register file must be able to take in two inputs and produce a result. This is also known as dual-porting. In the case of this baseline design the two inputs will be two concurrent reads to 32 bit ports and the result is a write to a 32 bit port.
The register file is to contain the following ports:

- Input ports:
  - Two 5-bit source register numbers for reference for the two 32-bit read ports (ra1, and ra2)
  - One 5-bit destination register number for reference to the 32-bit write port (wa3)
  - One 32-bit data port for writing (wd3)
  - One write enable signal (we3)
  - One clock (clk)

- Output ports:
  - Two 32-bit register values for the read ports (rd1, and rd2)

The register file writes the 32-bit data (wd3) at the location specified by the 5-bit write register (wa3) only when write is enabled (we3 is high) and only on the rising edge of the clock (clk). Also, the register file, on the same clock cycle, can combinatorially read two 32-bit data ports (rd1 and rd2) specified by their respective 5-bit registers (ra1 and ra2). The register file's register zero should always be zero, and writing to register zero has no effect on the register value, so register zero will remain zero no matter what is written to it.

For example, if write is enabled (we3) and the 5-bit register destination for write (wa3) is set to 5'b01 then on the next rising clock edge the register file's register 1 will be written to be what was in the 32-bit data port for write (wd3). However, the register file will not be written to until the clock reaches its rising edge, and only if write is enabled. The read ports can read any register in the register file, so if one of the 5-bit register source numbers (ra1 or ra2) is set to the same source number that was just written to, which is 5'b01, then on the next rising edge of a clock the 32-bit register read port (rd1 or rd2) would read from the register file's register 1. In this case the 32-bit register read port (rd1 or rd2) would read what was just written the cycle before by the 32-bit data writing port (wd3).

This design illustrates the principle of hierarchy inside its write functionality. The write port (wd3) waits on the enable write (we3) to tell it whether or not it can write to the register file, and also when the register file actions happen only under the clock's rising edge.

3. Design

The alternative design is simple and implements all points of the baseline design. The design parameters specified that the data should be read from the register file combinatorially and should always be read as a 0 if the register address is $0. We completed this specification by implementing a ternary operator that would return "32'b0" if the address Boolean value is false and the register value at the specified location if otherwise true. Programmatically:

```
rd1 = ra1 ? rf[ra1] : 32'b0;
rd2 = ra2 ? rf[ra2] : 32'b0;
```

This design is efficient and works well due to the fact that non-zero numerical values are always evaluated to true in a boolean expression and null will always return false.

For writing to the register, we implemented similar logic. Both read and write operations are encapsulated inside an @always() function that triggers on a rising clock edge. In order to write, we need not only the clock to be high,

but we need the write enable to be high as well. We also cannot write the $0 register. To accomplish this, we used an if statement that evaluates both of these to be true:

```
if (we3 & wa3)
            rf[wa3] <= wd3;
```

This if statement will be evaluated to true only if both the write address is non-zero and the write enable trigger is simultaneously high. This prevents the $0 register from being written to. If both evaluate to true, the write address specified is sequentially written with the data found in the 32-bit vector wd3.

We achieve both encapsulation and hierarchy within this design. By enclosing both functions in the always(@posedge clk) function, we are demonstrating the importance of the rising edge. We then create another hierarchy in the design by encapsulating the write function in the write enable trigger's rising edge. Due to the simplicity and straightforwardness of the design, many design principles are not able to be demonstrated in this case.

For our design strategy, we chose a simple and efficient design over a verbose one to maintain elegance and simplicity. Using ternary operators can sometimes confuse programmers when reviewing code, especially in the case that they are not the authors. In this case, this concern was not at the forefront as it seemed the most logical decision to make. This also reduces the complexity of the code and will allow for our design to be as modular as possible so we can implement the register file in later designs to make future design decisions simple by having a well-defined codebase.

4. Testing Strategy

This design was tested through the testbench. We used random value tests to manually ensure that all the requirements of the register file are met, as well as ensuring that the register file is not doing anything outside of those requirements. Using random value tests allowed us to heavily test each requirement in several different ways. Since this lab does not heavily involve a variety of different outputs or calculations there were not that many different aspects of the design that needed to be tested.

In the testbench we assigned values to all of the inputs randomly to ensure that the read outputs came out as expected. We changed the values of all of the inputs and checked that the output values came out as expected. The clock however was set to change every time unit to keep the flow of the design going since actions only occurred on the rising edge of the clock.

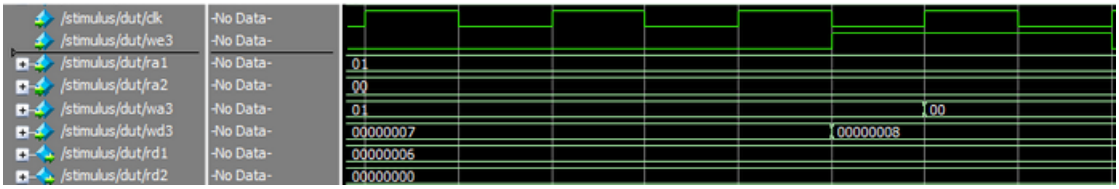
We ensured that when the read enable was off or when the clock was not on a rising edge that nothing would be written, and at the same time ensure that the register file is able to read from the specified ports. To do this we would

set the write enable on and set the write port (wd3) to a random value. The 5-bit source registers (ra1, and wa3) were set to point to the same register on the register file. Then if on the rising clock edge the read ports set to the value of the write port we would know it is working.

We also did a couple more tests in the same manner to account for not being able to write when the enable is not enabled, and for the cases of register zero always needing to be the value of zero and not being able to be changed.

5. Evaluation

We evaluated the design by attempting to write out of enable edges, during enable edges, and observing how the addresses were interpreted.



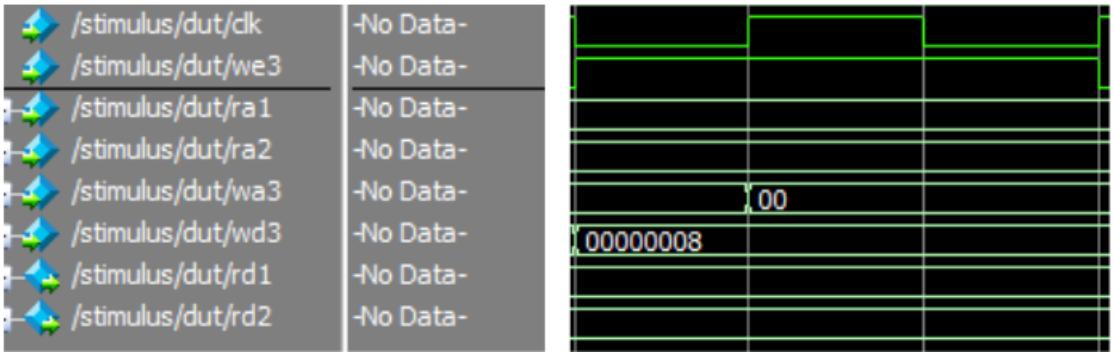
In this series of waveforms, we can see the enable signal, we3, is low for 2.5 cycles, or 5 periods, and is then triggered high for 1.5 cycles, or 3 periods.

During this period we see rd1 is 6 and rd2 is 0. This is because the initial value at the register file location ra1 which is the location 1 in the register, is 6. rd2 is initially set to 0, which should always return zero, as it does.

We initialized wd3 to be 7 and set the write address, wa3 to be 1, the location of ra1. We can see that this proves two major points of the design:

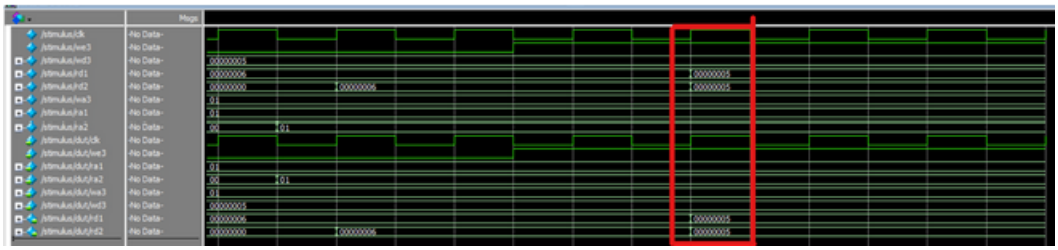
- The register file will always return 0 for register 0
- The write enable trigger works correctly and will only write out when triggered.

Additionally, we can notice another important feature from the design criteria, which is register zero cannot be written to:



The waveform written “00” is the write address, which should trigger the rda2 to change to 7 if it was allowed to write to register 0. This is not depicted, therefore proving that register 0 cannot be written out.

The following waveforms highlight that our read operations are combinatorial and can occur simultaneously.



These tests conclude that our alternative design matches the requirements listed in the baseline design.

6. References

1. https://www.javatpoint.com/verilog-scalar-and-vector