EE/CSE 469 Computer Architecture I

Lab 1 ALU and

Register File

Lab Objectives:

1. Refresh memory on digital design, HDL and simulation as covered in prerequisite Digital Design Course.
2. Design and simulate a register file.
3. Design and simulate an ALU.

Big Picture:

One of the major projects within this course is the design and simulation of a simple pipelined processor. Lab 1 will have you build the two integral components of the processor. Lab 2 will have you link these components, as well as some additional logic to create a complete single cycle processor. Lab 3 will have you add pipelining to your processor and resolve the issues that come with the performance enhancement. For that reason, it is important you get this first lab right as the following labs build on it. You have the freedom to implement the labs structurally or with RTL, this is left up to you as the designer.

Introduction:

This lab consists of 3 tasks. In task 1 you will design and simulate a simple module using Quartus. This warmup task is meant to refresh your digital design memory.

In task 2 you will design a register file. A register file can be thought of as a 2-D array of flip flops. You can write to and read from individual rows of this array by providing row addresses, or by enabling write and providing a new row of bits, called a word. You will write the SystemVerilog and simulate your design in ModelSim.

In task 3 you will design an Arithmetic Logic Unit ALU. An ALU is a combinational unit which preforms operations on two operands and produces a result and a set of flags. The operation performed is determined by a control input to the ALU. You will write the SystemVerilog and simulate your design in ModelSim.

Task 1: Warmup Exercise on ModelSim

This task is a refresher on creating a Quartus project, writing a SystemVerilog program, and simulating it in ModelSim.

Using Quartus Prime Software:

Please install Quartus Prime Lite Edition 17.0 on your PC for writing your code, developing your system and simulating it on ModelSim. In the case that you use a Mac, you can install a virtual machine, which allows you to run Windows on a Mac. VMWare Fusion 13 Player is available for free for non-commercial use – you can download it here:

https://customerconnect.vmware.com/en/evalcenter?p=fusion-player-personal-13

To install the Quartus Prime Lite software, go to

http://fpgasoftware.intel.com/17.0/?edition=lite&platform=windows&download_manager=dlm3

and **choose version 17.0 from the top drop-down menu**, download the free web edition, and install it. Note that you will have to register to be able to download the software. The file to download is the 5.8 GB tar file under the "Combined files" tab. Extract the tar file using a program like 7-zip and run the QuartusLiteSetup-17.0windows.exe file. When it asks for the components to install, make sure you select each of these:

- Quartus Prime Lite Edition
- (Free) Devices: Cyclone V
- ModelSim: Intel FPGA Starter Edition (Free)

When the software installation is done, run Quartus, and if asked about licensing just run the software (we use the free version, so no license required).

Making a Quartus Project and ModelSim Simulation:

For this task, follow along with the series of video tutorials that will walk you through the steps of creating a full adder using SystemVerilog as well as simulating the design in ModelSim.

Please follow the tutorials in the following order:

1. Launch the Quartus Prime software.
2. Create a project from scratch. Please follow the steps in the following video and use the same project name as in the video: https://www.youtube.com/watch?v=iLbmSTG7bpA
3. Implement the full adder using SystemVerilog and simulate it using ModelSim. Please follow the following video: https://www.youtube.com/watch?v=BcvclrqZ2fc
   Note: In the video when it refers to compiling the project for the first time, it may give you a compilation error. If you run the video for a few more seconds, it'll tell you about setting the top-level modules so that the program compiles.
4. Finally, certain features of ModelSim will be used to present simulations in a legible and standardized way. Please watch the following video for an introduction to these features: https://www.youtube.com/watch?v=eAqId9DWj7Y

The tutorials above demonstrate a way to simulate your work, but it is not the only. Any valid methods that produce waveforms through ModelSim, such as runwave.do scripts, can be used as well.

Task 2: Register File

The register file is a core component of any processor and acts as the primary operational memory device. In English all this means is that the processor has a small memory built into it so that it can evaluate and and store operations for future use or saving to the larger memory. In SystemVerilog, register files are sequential circuits and an example of a **16x32 one read, one write, synchronous** register file is given below:

```
module reg_file_ex(input  logic     clk, wr_en,
                   input  logic [31:0] write_data,      input
                   logic [3:0]  write_addr,             input
                   logic [3:0]  read_addr,              output
                   logic [31:0] read_data);

        logic [15:0][31:0] memory;

    always_ff @(posedge clk) begin
            if (wr_en) begin
            memory[write_addr] <= write_data;
            end

                read_data <= memory[read_addr];
    end
endmodule
```

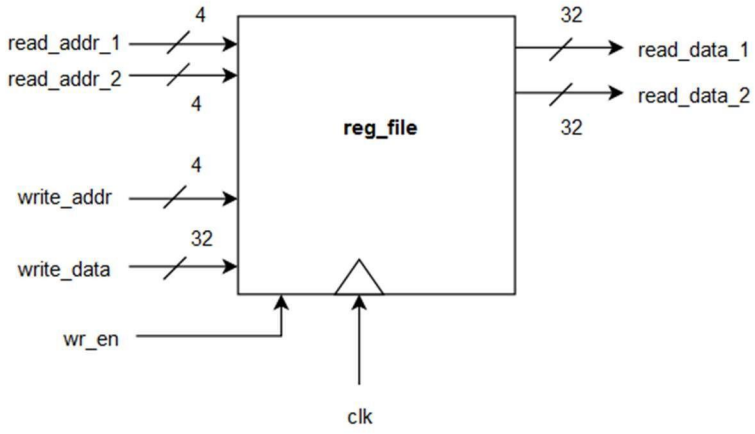
Pay attention to the terms we've used to define this memory and how they are expressed in the code. You can use this code as a framework for the register file you will be creating which will have a different set of terms it is defined in.

SystemVerilog Code:

Create a 16 by 32 register file in SystemVerilog. Name it reg_file.sv. It should have the following module declaration:

```
module reg_file(input  logic       clk, wr_en,
                   input  logic [31:0] write_data,
```

```
input  logic [3:0]  write_addr,
read_addr1, read_addr2,
read_data1, read_data2);
                                    input  logic [3:0]
                                    output logic [31:0]
```



This register file needs to be a **16x32**, **two read port, one write port,** and **asynchronous**. In other words, this ram should have 16 addresses, each storing a 32 bit word. There are two read ports and one write port, each of which is controlled independently. Finally, the reads are asynchronous, meaning that the moment a new address is supplied, the output port should change to reflect the data at the new address. You are permitted to write the code using RTL or structurally, whichever you prefer.

Simulation and Testing:

The register file is a sequential element that needs its input/output timing verified. Create a testbench named reg_file_testbench.sv and verify at least the following:

1. Write data is written into the register file the clock cycle **after** wr_en is asserted.
2. Read data is updated to the register data at an address the **same** cycle the address was provided. Do this for both read addresses and data outputs.
3. Read data is updated to write data at an address the cycle **after** the address was provided if the write address is the same and wr_en was asserted. Do this for both read addresses and data outputs.

Compile your register file and testbench in ModelSim and simulate the design. Your test waveforms should include only the following signals in the following order, from top to bottom: clk, wr_en, write_addr, write_data, read_addr1, read_addr2, read_data1, read_data2. The simulation should also show the internal registers of the register file below those signals. Ensure you are using radix so that these signals are either unsigned or hexadecimal.

Task 3: ALU

An ALU (Arithmetic Logic Unit) is a flexible combinational block which computes any of several operations. Under the hood the ALU computes every operation but selects one result based on the control signals given. In our processor we will have a relatively simple, but complete, ALU which preforms addition, subtraction, ANDing and ORing.

SystemVerilog Code:

Create a 32-bit ALU in SystemVerilog. Name the file alu.sv. It should have the following module declaration:

```
module alu(input  logic [31:0] a, b,
           input  logic [1:0]  ALUControl,
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags);
```

The four bits of ALUFlags should be TRUE if a condition is met. The four flags are as follows:

ALUFlag bit	Meaning
3	Result is negative
2	Result is 0
1	The adder produces a carry out
0	The adder results in overflow

An adder is a relatively expensive piece of hardware. Keep in mind that both addition and subtraction can be implemented using just a single adder. You may use RTL or design your ALU structurally, whichever you prefer.

Simulation and Testing:

Now you can test the 32-bit ALU in ModelSim. It is prudent to think through a set of input vectors.

Develop an appropriate set of test vectors to convince a reasonable person that your design is probably correct. Complete Table 1 to verify that all four ALU operations work as they are supposed to. Note that the values are expressed in **hexadecimal** to reduce the amount of writing.

Test	ALUControl[1:0]	A	B	Y	ALUFlags
ADD 0+0	0	00000000	00000000	00000000	4
ADD 0+(-1)	0	00000000	FFFFFFFF	FFFFFFFF	8
ADD 1+(-1)	0	00000001	FFFFFFFF	00000000	6
ADD FF+1	0	000000FF	00000001		
SUB 0-0	1	00000000	00000000	00000000	6
SUB 0-(-1)		00000000	FFFFFFFF	00000001	0
SUB 1-1		00000001			
SUB 100-1		00000100			
AND FFFFFFFF, FFFFFFFF		FFFFFFFF			
AND FFFFFFFF, 12345678		FFFFFFFF	12345678	12345678	0
AND 12345678, 87654321		12345678			
AND 00000000, FFFFFFFF		00000000			
OR FFFFFFFF , FFFFFFFF		FFFFFFFF			
OR 12345678 , 87654321		12345678			
OR 00000000 , FFFFFFFF		00000000			
OR 00000000 , 00000000		00000000			

Table 1. ALU operations

It may be helpful to review how to read vector files into SystemVerilog testbench to complete the simulation. A useful video on the topic can be found here:
https://www.youtube.com/watch?v=8-8h2tImRD4
Build a vector based testbench in a separate file called testbench.sv to test your 32-bit ALU. To do this, you'll need a file containing test vectors. Create a file called alu.tv with all your vectors. For example, the file for describing the first three lines in Table 1 might look like this:

0_00000000_00000000_00000000_4
0_00000000_FFFFFFFF_FFFFFFFF_8
0_00000001_FFFFFFFF_00000000_6

Hint: Remember that each hexadecimal digit in the test vector file represents 4 bits. Be careful when pulling signals from the file that are not multiples of four bits.

You can create the test vector file in any text editor, but make sure you save it as text only, and be sure the program does not append any unexpected characters on the end of your file. For example, in WordPad select **File→Save As**. In the "Save as type" box choose "Text Document – MS-DOS Format" and type "alu.tv" in the File name box. It will warn you that you are saving your document in Text Only format, click "Yes".

Now create a vector based testbench for your ALU. Name it alu_testbench.sv. Remember to use readmem.h to read in the test vectors you just made.

Compile your alu and testbench in ModelSim and simulate the design. Run for a long enough time to check all of the vectors. If you encounter any errors, correct your design and rerun. Your test waveforms should include only the following signals in the following order, from top to bottom: ALUControl, a, b, Result, ALUFlags.

Deliverables:

Upload the following to canvas:
A detailed lab report that includes procedure for tasks 2 and 3, results of tasks 2 and 3, and an appendix for the whole lab. The report should also include:

1. Your table of test vectors (Table 1) embedded into your procedure section.
2. Your simulation waveforms embedded into the results section. The signals should be ordered as described earlier in the spec. The simulations will need to show the signals outlined in the simulation and testing sections of each task, and will need to verify any cases mentioned.
3. Your design files (alu.sv, alu.tv, alu_testbench.sv, reg_file.sv and reg_file_testbench.sv) in the appendix section. To include the files in the appendix, follow instructions on the document called “exporting code to a pdf file”.
4. All design files: alu.sv, alu.tv, alu_testbench.sv, alu.tv, reg_file.sv and reg_file_testbench.sv Although you included the code in the appendix of your report, you must also submit

   all the source files for testing. Please note that your code must be well commented. Please refer to the “commenting guide” document on canvas.
5. Please follow the report format as outlined in “lab report outline” and “lab report template” documents on canvas.