

## ECE 2504: Introduction to Computer Engineering

### Design Project 4: Simple Computer Assembly Language Programming

**Read the entire specification before you begin working on this project!**

#### Honor Code Requirements

You must comply with all provisions of Virginia Tech's Honor System. The program and report must be your own work. You may ask other students general questions about the Simple Computer instruction set and how Quartus works. You are not allowed to ask anyone except your instructor or a CEL GTA questions about the design of the program. The program design, coding, debugging, and testing must be your own work. It is an honor code violation to share your design with another person or to copy another person's design either as a paper design or as a computer file. Ask your instructor if you have any questions about what is or is not allowed under the Honor Code.

#### Objectives

In this project, you will design, code, debug, and test a simple assembly language program for the Simple Computer. This project will introduce and reinforce concepts related to the load/store architecture, assembly language, branches, jumps, arithmetic operations, assemblers, and simulators.

#### Preparation

You must have access to a computer that can run Quartus. You must have a DE0 Nano board.

Read this project specification in its entirety. Consult the appropriate sections of Chapters 3, 6 and 8 of the textbook. You should also consult the DE0 Nano board user's manual, particularly Chapter 3 and 6, the Quartus instructions from previous assignments, and the Lab Manual.

#### Project Description

The Simple Computer from Chapter 8 of the textbook is a single-cycle, load-store central processing unit (CPU). The single-cycle Simple Computer illustrates many of the major principles and design constraints involved in implementing a CPU. For this project, you will write a small assembly language program to process the values stored in an array in the data memory. You will verify the operation of your program on the Simple Computer in a Verilog simulation and on the DE0 Nano board.

The project files include the Quartus archived project (.qar), which contains a working version of the Simple Computer, and a small starter program. The system takes as input the four DIP switches on the DE0 Nano board (SW[3:0]) and the two pushbuttons (KEY[1:0]). The behavior of the switches and buttons are much the same as in Design Project 3, with a couple of features that were not used in Project 3 (Table 1). As with the previous project, KEY0 is the reset signal for the project, while KEY1 advances the Simple Computer by one clock cycle. Reset is synchronous with the clock, which means that to reset the Simple Computer, you must press KEY0 and hold it down while pressing and releasing KEY1. For this project, KEY0 is also used to select between PC/IR and R6/R7 being displayed on the LEDs, as described in the next paragraph.

The DIP switches control a mux in the top-level project module; the mux is used to select which value is displayed on the LEDs, as shown in Table 1. SW[3:1] select which register, and SW[0] selects between the most significant byte and least significant byte of the register. When SW[3:1] = 110, if KEY0 is not pressed, the PC is displayed, but while KEY0 is held down, R6 is displayed. When SW[3:1] = 111, KEY0 toggles between IR (KEY0 not pressed) and R7 (KEY0 pressed).

*You do not need to modify any of the Verilog files in the project.* You should only modify the `instruction.txt` and `data.txt` memory initialization files to implement your program to meet the specification below.

SW[3:1]	Value displayed on LEDs SW[0] selects between MSB (1) and LSB (0)
000	R0
001	R1
010	R2
011	R3
100	R4
101	R5
110	Program Counter (PC) (KEY0 not pressed) R6 (KEY0 pressed)
111	Instruction Register (IR) (KEY0 not pressed) R7 (KEY0 pressed)

Table 1: DIP switch select lines and value displayed on LED

### Background

The following 16-bit ECE2504 floating point (FP) format will be used for this project. It is comprised of three fields: sign (S), exponent (E), and fractional part of the mantissa (F). Similar to the IEEE 754 format covered in class, these fields represent a number in the format

$$(-1)^S \times 1.F \times 2^E, \\ \text{where } E = e + 7.$$

field	S	E	F
# bits	1	4	11
notes	0: positive mantissa 1: negative mantissa	Excess 7 format $E = e + 7$	Fractional portion of the mantissa

Table 2: ECE2504 floating point (FP) format used for this project

*Example 1:* A 16-bit register contains 0x7345. If this data is in ECE2504 FP format, it represents the value +232.625.

The register contains 0b0111001101000101.

Therefore, S = 0, E = 1110, F = 1101000101

$$(-1)^S \times 1.F \times 2^E = (-1)^0 \times 1.1101000101 \times 2^7 = +11101000.101 = +232.625$$

*Example 2:* A 16-bit register contains 0xACD0. If this data is in ECE2504 FP format, it represents the value -0.400390625.

The register contains 0b1010110011010000.

Therefore, S = 1, E = 0101, F = 10011010000

$$(-1)^S \times 1.F \times 2^E = (-1)^1 \times 1.10011010000 \times 2^{-2} = -0.0110011010000 = -0.400390625$$

### Program Description and Memory Layout

Write an assembly language program that will determine the maximum (closest to  $+\infty$ ) and minimum (closest to  $-\infty$ ) values in an array stored in ECE2504 floating point format.

- Input data (16-bit floating point numbers) are stored in consecutive data memory locations between location 0x10 and 0x1F. The location and length of the data array are variable. The input data array should be stored in data memory prior to execution.
- Data memory location 0x00 contains the pointer M, which indicates the memory address of the first element of the data array. You can assume that M will be a value between 0x10 and 0x18. This value should be stored in memory location 0x00 prior to execution.

- Data memory location 0x01 contains a value N, which indicates how many elements are in the array. You can assume that N will be a value between 4 and 8. **This value should be stored in memory location 0x01 prior to execution.**
- Your program must read the location and length of the data array from data memory. Specific values will be used for testing (Table 5), but your program must be capable of operating properly if other values for the address and length of the data array were to be used. You can assume that values used for validation will follow the specifications above ( $0x10 \leq M \leq 0x18$ ,  $4 \leq N \leq 8$ )
- Your results should be stored in data memory location 0x02 and 0x03, the maximum value in location 0x2 and the minimum in location 0x3.
- In addition to the task described above (determining the max and min values), your program must also write the BCD code for the last four digits of your ID number to memory location 0x04. For example, if the last four digits of your ID are “1234”, the value 0b0001001000110100 (this is the same as 0h1234) should be stored in memory location 0x04. This must be performed during execution of your program, **not** prior to execution.

### Required memory locations

Your program must follow the data memory requirements for variables that will be used during execution shown in Table 3. The data.txt used for validation will be set up according to these requirements and your program will not validate correctly if you do not follow them.

Your program is permitted to modify any locations of data memory other than as listed on the table. For example, you might need temporary storage for other variables. However, such addresses should **ONLY** be used to store values for your program’s use during execution. Your program may **NOT** rely upon values being placed into these addresses prior to your program being run.






Memory Location	Required Content	Comments
0x00 	Location of first array element (pointer to the array)	Will be a value between 0x10 and 0x18 Should not be changed by your program
0x01 	Length of data array	Will be in the range 4-8 Should not be changed by your program
0x02 	Maximum value	Determined and stored by your program
0x03 	Minimum value	Determined and stored by your program
0x04 	Last four digits of your ID	Must be written by your program during execution Must be in BCD
0x10 - 0x1F	Data array (Input)	16-bit floating point numbers Should not be changed by your program

Table 3. Required data memory locations.

To write your program, edit the provided base program “`starter-program.txt`” using a text editor. Your source code program file **must** include sufficient comments to document the overall algorithm that you are using and the operation of the code itself. Note that `starter-program.txt` is a text file that contains assembly language mnemonics (e.g. `add r2,r3,r5`), and `instruction.txt` contains the 16-bit machine instructions that correspond to each line of assembly language (e.g. `0b0000010 010 011 101`, or `0x049D`). The instruction set can be found in Table 8-8 in the textbook.

There are portions of `starter-program.txt` that you must not change, as noted in the comments in the file. You must also refrain from changing the contents of **instruction** memory (in `instruction.txt`) *that correspond to those instructions*. These instructions provide the process by which your program will be validated on your Nano Board. Modifying them may result in your program failing in the validation step.

You should write and test small portions of your program at a time. For example, you could first make sure you can step through the array using the address for the array, and then load each array element into the register file without finding any of the results. Then you should implement your comparison algorithm. During *testing*, you do not have to conform to the

memory layout requirements above; however, your *final program for validation* must conform to the requirements specified in this document.

### Assembling your source code

If you desire, you can assemble your source code manually to create the hex values for `instruction.txt`, the same as you did for the previous project. However, the program for this project is much bigger, so assembling the instructions manually will be tedious. There is a simple assembler available at:

<http://ece2504.ece.vt.edu/assembler>

Copy and paste this link into your browser. The web page will allow you to write source code in a text field, or to copy and paste it from another file. It will then generate a bare (uncommented, no address directives) `instruction.txt` for you.

The assembler is stable but experimental. It does generate assembler errors, but it may not provide detailed warnings about incorrect formatting or incorrect instruction use. Therefore, you must check your source file carefully before uploading it to the web page. The assembler has *at least* the following restrictions:

1. The source code file must be in plain text.
2. Comments are delimited by `//`. Comments can be on their own line or in-line with an instruction. If the comment is on its own line, the slashes must start in column 1 of the line.
3. Instructions and register operands are not case sensitive.
4. There is syntax-checking, but you should expect that improper syntax might not create warnings or errors.
5. Numeric values for immediate operands and branch offsets must be specified in decimal.
6. The behavior of the assembler for input that does not conform to the above restrictions is indeterminate.

There might be other restrictions that are not known at this time. If you believe you have found a bug in the assembler, please create as small a program as you can that demonstrates the bug and send the program along with a brief description of the bug to your instructor.

***Make sure that you add or maintain updated `instruction.txt` and `data.txt` files within your set of project files. When you change either file, you must recompile the project and reprogram your board.***

File	Format	Purpose
*.v	Verilog <b>DO NOT MODIFY</b>	These files form a working model of the Simple Computer. <b>DO NOT MODIFY ANY OF THESE FILES</b>
starter-program.txt	Text (assembly)	Edit this source code to create your assembly language program.
instruction.txt	Hexadecimal (16-bit machine instructions)	Contents of instruction memory. Create this file by either using the online assembler, or by assembling your source code by hand.
data.txt	Hexadecimal (16-bit data)	Contents of data memory. Edit this in a text editor to store your data (according to Table 3)

Table 4. Project files.

### Debugging and Testing your Program

You should use Quartus environment to debug and test your program. You do not need to modify the Verilog model provided for this project except for the `instruction.txt` and `data.txt` files.

During simulation, you will want to include sufficient signals in the waveform window that you can see how your program is executing. At a minimum, you will want to include PC, IR, and registers R0-R7. As with the previous projects, you must

create input waveforms for the clock, the pushbuttons, and the switches.

You will want to test a number of different data sets. Your program must work for any array start location in the range 0x10-0x18, and for any array length in the range 4-8.

Simulate your final program using the following two data sets, and include a waveform of each in the report. Make sure they clearly show the values of at least the PC, IR, and registers R0- R7. Given the number of cycles it will take your program to complete, you will likely have to show the waveforms in multiple figures so that the values in the signals are legible. Clearly label any outputs stored to memory in the waveform.


Memory Location	Contents for Test 1	Contents for Test 2
0x00	0x12	0x18
0x01	0x04	0x07
0x02	0x0000	
0x03	0x0000	
0x04	0x0000	
0x10	0xB470	
0x11	0x29F3	
0x12		0x11F1
0x13		0x3D98
0x14		0xEB2C
0x15		0xB6C6
0x16	0xF32C	
0x17	0x5370	
0x18	0x4AEC	
0x19	0x1DA0	
0x1A	0xA75B	
0x1B	0x60D3	
0x1C	0x8756	
0x1D	0xD6E8	
0x1E	0x45A0	
0x1F	0x7F68	

Table 5. Test data. Include these simulations in your report.

After your model simulates satisfactorily, you must compile it and then program the DE0 Nano board with it. Use the DIP switches to control the values displayed on the LEDs and verify that the program behaves the same on the board as it does in simulation.

## Submission Requirements

### Validation

This project does not require CEL validation. Instead, you will provide the source files that will allow the GTA to compile and test your design on the DE0 Nano Board. The `data.txt` used for validation will be set up according to the requirements listed in Table 2. Your program will not validate correctly if you do not follow them.

### Report

The reporting requirements for this project are minimal and should include only the following items in the order listed below. This report is to be submitted as a single PDF file on the Canvas Assignment page. The single PDF file should contain the following items in this order:

1. Your name and last four digits of your student ID.
2. A well-formatted, well commented version your assembly source program. *You will also have to submit `instruction.txt`, but this element of the report is not `instruction.txt`. It is the source code you used to generate it (contained in your modified `starter-program.txt`).*
3. Simulation waveforms clearly demonstrating the proper execution of your program using the provided test data. If your program does not execute correctly, include a statement on the extent to which it does work and what problems you encountered.

In addition to submitting the report, you must also submit the final versions of `starter-program.txt`, and `instruction.txt` files from your *completed* Quartus project. *Do not submit the original versions of these files. Submit the versions that correspond to your final results.*