ECE 2504: Introduction to Computer Engineering (Fall 2017)
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 differences (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. The change from the operation in the previous project is that 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

Recall that BCD, or 8421, is a 4-bit code used to represent the decimal digits 0-9. Addition of BCD numbers is performed digit by digit and a correction must be made for any digit that results in a value greater than nine. Consider the following example:

Addition of two four digit numbers:

```
 0011 1000 0111 0010
+0010 0001 0100 0110
```

1. Adding the first digit results in 1000, a valid BCD code. No correction is needed.

```
 0011 1000 0111 0010
+0010 0001 0100 0110
           1000
```

2. Adding the 2nd digit results in 1011, not a valid BCD code. It must be corrected (see #3).

```
 0011 1000 0111 0010
+0010 0001 0100 0110
      1011 1000
```

3. Correct the 2nd digit by adding 6 to that digit. This generates a carry into the next digit:

```
        1
 0011 1000 0111 0010
+0010 0001 0100 0110
      1011 1000
     +0110
      0001
```

4. Adding the 3rd digit (including the carry) results in 1010, not a valid BCD code (see #5).

```
        1
 0011 1000 0111 0010
+0010 0001 0100 0110
 1010 1011 1000
     +0110
      0001
```

5. Correct the 3rd digit by adding 6 to that digit. This generates a carry into the next digit:

```
    1   1
 0011 1000 0111 0010
+0010 0001 0100 0110
 1010 1011 1000
+0110+0110
 0000 0001
```

6. Adding the 4th digit (including the carry) results in 0110, a valid BCD code. No correction is needed.

```
    1   1
 0011 1000 0111 0010
+0010 0001 0100 0110
 0110 1010 1011 1000
     +0110+0110
      0000 0001
```

The final BCD result is

```
0110 0000 0001 1000
```

Program Description and Memory Layout

Write an assembly language program that will add pairs of BCD numbers and produce a valid BCD result for each pair.

- Two input data arrays are stored in data memory. The first occupies addresses 0x10 – 0x1F; the second occupies addresses 0x20 – 0x2F. The result of each sum will be stored in a third array, which occupies addresses 0x30 – 0x3F. Any memory location in the range 0x30-0x3F that is not used should contain 0x0000.
- Data memory location 0x00 contains a value M, which indicates the memory address of the first array element to be added. Data memory location 0x01 contains a value N, which indicates how many pairs of array elements should be added together.

Example 1: if N=3 and M = 0x10, then your program should add the array elements in data memory locations 0x10 and 0x20, the array elements in data memory locations 0x11 and 0x21, and the array elements in data memory locations 0x12 and 0x22. The program should store the valid sums or error indicators in data memory locations 0x30, 0x31 and 0x32, respectively. Locations 0x33-0x3F should contain 0x0000.

Example 2: if N = 4 and M = 0x13, then your program should add the array elements in data memory locations 0x13 and 0x23, the array elements in data memory locations 0x14 and 0x24, the array elements in data memory locations 0x15 and 0x25, and the array elements in data memory locations 0x16 and 0x26. The program should store the valid sums or error indicators in data memory locations 0x33, 0x34, 0x35 and 0x36, respectively. Locations 0x30-0x32 and 0x37-0x3F should contain 0x0000.

- Before adding, your program must check both operands to see if they are valid BCD codes
- If both operands are valid, the BCD sum should be stored in the appropriate memory location. Results should be valid BCD codes.
- If either of the operands is invalid, the sum should not be calculated. Instead, write the value 0xFFFF to the correct output location. 0xFFFF will never be the sum of two valid binary-coded decimal numbers. In the case of this program, we are using it as an error flag.
- You do not need to consider overflow. For example, the results of adding 1000 0101 0111 0010 and 0100 0110 0011 0001 would be 5 digits, or 0001 0011 0010 0000 0011. You only need to store the lower four digits, or 0011 0010 0000 0011.
- Before adding the array elements, your program must write the BCD code for the last four digits of your ID number to memory location 0x14. 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 0x14. This must be performed during execution of your program, not prior to execution.
- The location of the first element to be added should be stored in memory location 0x00.
- The length of the data arrays to be summed should be stored in memory location 0x01.
- Your program must read the location and length of the data arrays to be summed from data memory. Specific values will be used for validation, but your program must be capable of operating properly if other values for the address of the data array were to be used.
- You should use "subroutines" wherever possible to perform repeated functions. For example, you may write a subroutine to that calculates and adds to the checksum the appropriate weight for each digit.

Required memory locations

Your program must follow the data memory requirements for variables that will be used during execution shown in Table 2. 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 to be added	Will be in the range 0x10-0x1F Should not be changed by your program
0x01	Number of BCD pairs to be added together	Will be in the range 0x1-0xF Should not be changed by your program
0x10-0x1F	Data array containing the first operand of each sum	Must be in BCD With the exception of location 0x14, should not be changed by your program
0x14	last four digits of your ID	Must be written by your program during execution Must be in BCD
0x20-0x2F	Data array containing the second operand of each sum	Must be in BCD
0x30-0x3F	Data array containing the result of each sum	Must be in BCD, or the error code 0xFFFF Any unused location should contain 0x0000

Table 2. 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 text book.

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 BCD addition 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:

https://rawgit.com/vtsman/ECE-2504-assembler/noLabels/index.html

Copy and paste this link into your browser; **do not try to use the hyperlink**. The web page will allow you to write source code 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 DO NOT MODIFY	These files form a working model of the Simple Computer. DO NOT MODIFY ANY OF THESE FILES
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 2)

Table 3. 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 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. The pushbuttons should change on the negative edge of clock and should hold their value for several clock cycles after any change to reflect the operation of the actual hardware.

You will want to test a number of different data sets. Your program must work for any array start location in the range 0x10-0x1F, and for any array length such that the end of the array will not exceed 0x1F. For example, for a starting address of 0x29, the greatest number of BCD pairs that can be added together is 7.

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	Memory Location	Contents for Test 1	Contents for Test 2
0x00	0x13	0x1A	0x01	0x04	0x03
0x10	0x2345		0x20	0x8765	
0x11	0x7654		0x21	0x1234	
0x12	0x1234		0x22	0x7654	
0x13	0x3742		0x23	0x1276	
0x14	0xXX		0x24	0x1881	
0x15	0x9538		0x25	0x2151	
0x16	0x5267		0x26	0x4324	
0x17	0x0032		0x27	0x0089	
0x18	0x4321		0x28	0x4321	
0x19	0x1234		0x29	0x1234	
0x1A	0x7572		0x2A	0x4631	
0x1B	0x0505		0x2B	0x2837	
0x1C	0x4332		0x2C	0x5181	
0x1D	0x5432		0x2D	0x6543	
0x1E	0x2345		0x2E	0x9876	
0x1F	0x8765		0x2F	0x2345	

Table 4. Test data. Include these simulations in your report. Note that the initial contents of location 0x14 don't matter since your program will write the last four digits of your ID number to this location.

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. 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* <u>*not*</u> `instruction.txt`*. It is the source code you used to generate it (contained in* `starter-program.txt`*).*
2. 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 is 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.*