ECE 2504: Introduction to Computer Engineering (Spring 2015)

# 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. You must complete all elements of this project *individually*. 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 Web Edition. You must have a DE0 Nano board.

Read this project specification in its entirety. Consult the appropriate sections of Chapter 4, Chapter 7 and Chapter 9 of the textbook. As needed, you should also consult Quartus instructions from previous assignments and the DE0 Nano Board User Manual (which is on the DVD included with your board) – particularly Chapter 3 and 6.

*Project Description*

The Simple Computer from Chapter 9 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 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 files for the project are in the Quartus archived project posted with this description. The project includes a working version of the Simple Computer with a 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 Project 3, but with a couple of differences. 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 in operation from 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 multiplexer in the top-level project module; the multiplexer is used to select which value is displayed on the LEDs. The Verilog model provided for the module already has an 8-bit wide, 16-to-1 multiplexer instantiated in it with the DIP switches connected to the select lines. The multiplexer provides the outputs shown in Table 1.

|  |  |
| --- | --- |
| Value of 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 LEDs

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, KEY0 switches between displaying the PC and R6. When SW[3:1] = 111, KEY0 switches between IR and R7.

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.

Program Description and Memory Layout

Write an assembly language program that will perform a particular set of data analysis functions on an array stored in data memory. The array (contained in data memory addresses 0x0-0x7; see Table 2 below) consists of values stored in memory as signed 2’s complement values. Your program must find the array’s largest value, the array’s smallest value, and the (arithmetic) mean value of the array’s elements.

*Since the values in the registers represent integers, your calculated average might not represent the true average of the data values. Different processors, compilers, and computer languages handle integer operations differently. Based on the operations you have available, you might consider it a reasonable goal to produce an “average” that is equal to the actual average, rounded down. You might also decide to write code to implement a more accurate rounding scheme. The main idea is that in the absence of hardware that can perform certain operations, the problem of doing those operations becomes a software problem.*

You may assume that overflow will not occur as your program is performing arithmetic on the elements of the data array. While you should test your program on a mixture of positive and negative array values, you may choose them subject to this assumption.

The array will always have exactly eight elements, and it will always be located in the data memory addresses indicated. (Use these facts to establish useful memory pointers in particular registers.) While specific values will be used for validation, your program must be capable of operating properly if other values were to be used – subject to the previous assumption about overflow.

You should use looping subroutines wherever possible to perform repeated functions. Most of the process of traversing and operating upon the elements of your data array can be performed most efficiently with looping structures.

*Required memory locations*

Table 2 summarizes how you should plan to have your program use the *Data Memory*. Your program must follow these requirements for variables that will be used during execution.

|  |  |
| --- | --- |
| Data Memory Address  (or Address Range) | Purpose |
| 0x00-0x07 | The eight-element data array. *You will set the contents of these addresses in data.txt for each test you perform. Your program should not change these values during execution.* |
| 0x08 | Your program should store the data array’s *maximum value* here. |
| 0x09 | Your program should store the data array’s *minimum value* here. |
| 0x0A | Your program should store the data array’s *mean value* here. |
| 0x0B-0x1F | Available for your program’s general use. *These data memory addresses are available as temporary storage for other variables that your program might use.* |

Table 2: Summary of Data Memory Usage

Table 3 summarizes how you should plan to have your program use the *Instruction Memory*.It is absolutely important that you avoid modifying certain sections of the Instruction Memory. You will accomplish this by not modifying certain sections of the source program starter\_program.txt, which has been provided to you.

|  |  |
| --- | --- |
| Instruction Memory Address Range | Purpose |
| 0x00-0x02 | Branch to Main: These instructions set up and carry out a branch to Instruction Memory address 0xC, which is where the user code should start. The branch skips over the code that provides the means for validation. DO NOT MODIFY THE INSTRUCTIONS IN THIS SECTION, OR CAUSE THEM TO APPEAR IN ANY OTHER SECTION OF INSTRUCTION MEMORY. |
| 0x3-0xB | Validation and Termination: The instructions from 0x3 to 0x9 load the values in Data Memory addresses 0x8, 0x9, and 0xA into registers r1, r2, and r3, respectively. The instructions from 0xA to 0xB set up and execute an infinite loop to terminate the action of the processor. DO NOT MODIFY THE INSTRUCTIONS IN THIS SECTION, OR CAUSE THEM TO APPEAR IN ANY OTHER SECTION OF INSTRUCTION MEMORY. |
| 0xC-? | User Code: This is where you should place your code to solve the problem described in this specification. The last lines of your User Code should set up and execute a branch to Instruction Memory address 0x3, so that your program can perform the Validation and Termination sequence of instructions. |

Table 3: Summary of Instruction Memory Usage

Use a text editor to modify the provided base program to create and the source code for your program. 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.

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 operand into the register file without finding any of the results. Then you should implement your algorithms for finding the mean and extreme values of the array.

*Assembling your source code*

If you desire, you can assemble your source code manually to create the hex values for instruction.txt, as you did for the previous project. However, the program for this project is longer, so assembling the instructions manually will be tedious. There is a simple assembler available at <http://filebox.ece.vt.edu/~tlmartin/>. The web page will allow you to upload your source file, and will then generate a bare (uncommented, no address directives) instruction.txt for you.

The assembler is experimental and minimal: It has not been extensively tested, nor does it provide many warnings about poor 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:

* The source code file must be in plain text.
* Blank lines are not permitted. The assembler will stop at a blank line.
* The first two characters of a comment line must be //. Comments are only permitted on lines by themselves. Comments are not permitted on the same line with an instruction.
* Instructions and register operands are not case sensitive.
* There is minimal checking of syntax. In most cases, improper syntax will not create a warning but will silently generate improper code. For example, if you leave out a register operand with an ADI instruction – for example, using ADI r0, 3 instead of ADI r0, r0, 3 – there will be no warnings, but the machine code value will not be what you intended.
* Numeric values for immediate values and branch offsets must be specified in decimal.
* The ranges of target address offsets for branches are not checked. Negative offsets can be specified either as negative decimal numbers or as the decimal equivalent of the unsigned 6-bit number (e.g., BRZ r0, -3 is the same as BRZ r0, 61).
* 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.

*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.

After you have designed, written, and assembled your program, use Qsim to simulate the program’s execution and to verify that it works correctly. Once the program works, do the following test for your project submission.

1. Edit data.txt to change the contents of Data Memory locations 0x0-0x7. (This changes the data array.) Recompile.
2. Simulate your program’s execution to verify that finds the correct minimum, maximum, and mean values of the array. Include a waveform of the simulation of your final program in the report that clearly shows 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 extend the simulation time and show the waveform in multiple figures so that the values in the signals are legible.
3. Repeat steps 1 and 2 for at least one more set of data values.

Note that in the test above you should execute the program twice, on two different sets of values contained in Data Memory addresses 0x0-0x7.

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

After you have tested your program on your DE0 Nano board and are satisfied with its operation, take your computer and DE0 Nano board to the CEL and have the GTA validate the circuit by completing the included validation form. The waiting lines in the CEL can be very long as the due date approaches, so validate as early as you can. You should have your Quartus project open on your computer before validation for the GTA so that he can examine your design if necessary.

Before validating, you should set up the array in data memory so that it contains the values shown in Table 4 using the last four digits of your student ID number. Values should be put in memory as BCD.

|  |  |
| --- | --- |
| Data memory address | Value in BCD |
| 0x0 | Left two digits of the last four ID digits |
| 0x1 | Right two digits of the last four ID digits |
| 0x2 | First (left-most) digit followed by the last (right-most) digit of the last four ID digits |
| 0x3 | Second digit followed by the third digit of the last four ID digits |
| 0x4 | Sum of the first, second, and third digits of the last four ID digits |
| 0x5 | Sum of the last four ID digits |
| 0x6 | Sum of the last second, third, and fourth of the last four ID digits. |
| 0x7 | Third digit followed by the first digit of the last four ID digits. |
| 0x8 | Fourth digit followed by the second digit of the last four ID digits. |

Table 4. Array values to be used for validation

For example, if the last four digits of the student ID number were “1854” then data memory addresses 0x0-0x7 would (in order) be set to 0x18, 0x54, 0x14, 0x85, 0x14, 0x18, 0x17, 0x51, and 0x48.

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 Scholar assignments page. The single PDF file should contain the following items in this order:

* Cover sheet (provided at the end of this document)
* A well-formatted, well commented version your assembly source program (not instruction.txt, but the source you used to generate it).
* Simulation waveforms clearly demonstrating the proper execution of your program.

In addition to submitting the report on Scholar, you must also submit your instruction.txt and data.txt files from your Quartus project that you used for validation.

Your lab report should be submitted on Scholar. Your validation sheet should be submitted as a hard copy.