Lab 7: Assembly Programming

OBJECTIVES

- Understand the structure of a simple but functional CPU.
- Learn how to write assembly code, hand-assemble it into machine code, and verify its functionality with a simulator.
- Understand how assembly code instructions correspond to the functional hardware simulation of a CPU.

INTRODUCTION

With the knowledge gained in earlier labs, you now have a detailed understanding of the internal structure of a simple Central Processing Unit (CPU). Instead of continuing with your previous existing hardware designs, you are now given a more complete CPU, which is denoted as the Gator CPU (or G-CPU). In Part A of this lab, you will dissect and simulate the assembly code that is given with the G-CPU. In Part B, you will write a G-CPU assembly code program. You will then simulate this new code in Quartus to observe the G-CPU bus and register changes during program execution.

LAB STRUCTURE

In this lab, you will analyze and write assembly programs for the GCPU. In § 1, you will analyze a provided GCPU program to understand how it works. In § 2, you will write your own GCPU program that interprets an input table of data and outputs a table of processed data. You will also simulate and test your program to verify that it works correctly.

REQUIRED MATERIALS

- GCPU Quartus Archive (on the web site)
- G-CPU documentation (on the web site)
- `Lab7 PartA.xlsx` (on the website)
- `Assembly List.xlsx` (on the website)
- G-IDE-Full-v1.4 (on the website)

SUPPLEMENTAL MATERIALS

- Video example using G-IDE (on the website)

PRE-LAB PROCEDURE

GETTING STARTED WITH THE GCPU

The course website has multiple files that describe the GCPU. While working on this lab, you should download the G-CPU documentation PDF from the Lab 7 section on the course website. This document has a GCPU block diagram, information about every GCPU assembly instruction's function and machine code representation, flowcharts, a next-state truth table, and each of the bdf files. This information will be vital as you write and debug programs on the GCPU. To get started using the GCPU in Quartus, do the following:

1. Download GCPU Quartus Archive: gcpu_s25.qar.
2. Double click on the file gcpu_s25.qar (or Open Quartus and select "Project | Restore Archived Project…").
   a. Specify a "Destination folder:". I suggest storing it wherever you keep your other Quartus projects. (Do **NOT** use any dashes, spaces, or underscores in the path.)
   b. In Quartus, open the file computer.vwf. This will open the Simulator Waveform Editor.
      - Select **Simulation**, then **Simulation Settings**, then the **Restore Defaults** button on the bottom of this screen, and then select **Save**. This will fix the path information for the destination you used for the GCPU files.
3. The project has many folders, all under the destination folder you selected above. The main (top level) file, computer_simulation.bdf, is in this folder. The mif files and the simulation (.vwf) files are also in this folder.
   a. Whenever you want to **simulate a program with the GCPU**, make sure to set the top-level entity in Quartus to computer_simulation.bdf.
   b. If you ever want to test your GCPU program on the DE10-Lite hardware (where 7-segment displays will show the values in Accumulator A, Accumulator B, and the low-byte of the program counter), make sure to set the top-level entity in Quartus to computer_programming.bdf. **If you do not set** computer_programming.bdf **as your top-level entity** (**instead** of computer_simulation.bdf)**, the GCPU will behave incorrectly on the DE-10 Lite** and may damage the hardware**.** Switch the top-level entity back to computer_simulation.bdf whenever you want to simulate the GCPU again.

CREATING PROGRAMS FOR THE GCPU

Note: When opening a MIF file in Quartus, be sure to select ***Open as: Text*** (**not *Auto***). If you use auto, **all the comments described below will disappear!**

In § 1 of this lab, you will compile and simulator a pre-existing program, provide in eprom.mif. In § 2 of this lab, you will write a new program, hand assemble your code, and then put the machine codes into the "eprom.mif" file. Key points related to this file are:

1. The comments are surrounded by "%" signs. The left most number (or numbers) represents the **address** (or range of addresses) followed by the hex **value** to the right. For example:

```
37        :7C  % Address=$37, Data=$7C%
[37..42]  :A3  % Address=$37-$42, Data=$A3%
```

2. The last line of code in the "eprom.mif," file (after your program) should insert zeroes for all the remaining data in the ROM. This is accomplished as shown below:

```
[XX..FFF]  :00     %zero remaining memory%
```

   where **XX** represents the next address after the last address of your code. For example, if the last byte of your assembly program resides in memory location $25, then replace XX by $26 as shown below:

```
[26..FFF]  :00     %zero remaining memory%
```

   This will initialize all your remaining unused memory to a known value of zero. (A zero happens to represent the TAB instruction.)

Note: When opening a MIF file in Quartus, be sure to select ***Open as: Text*** (**not *Auto***). If you use Auto, **all the comments described above will disappear!**

1. SIMULATING EXISTING CODE

1. Open the file called "eprom.mif" in the main folder. **IMPORTANT:** Open this file after setting "Open as" to "Text" from the default "Auto." If you already opened the file with "Auto," do not save any changes you make. Close it and open it with "Open as:" set to "Text." The file eprom.mif contains the code that the GCPU will execute. The GCPU starts executing instructions from address 0 after it is reset.
2. Briefly describe the purpose of this program. Include this description in your pre-lab report.
3. The data for the program in "eprom.mif" can be found in the file "sram.mif." This is just a sample data file; later you will modify this file to create different data to test the program. Create a small table (see Table 2 on the last page of this document, also on the website in `Lab7 PartA.xlsx`) that describes how the registers change as the program is executed. For each row in the table, enter the value in columns labeled A through PC, assuming that the instruction has already been completed.
4. Compile the computer_simulation.bdf file (functionally) with the given program file eprom.mif and data file sram.mif. Open the provided simulation waveform file computer.vwf. Before simulating the first time (after restoring the archive), in the Simulation Waveform Editor, select **Simulation | Simulation Settings**; then select **Restore Defaults** at the bottom, and then select **Save**. Functionally simulate this design.
5. Compare the hand simulation results in your table with the Quartus simulation results in computer.vwf. Break this simulation into at least two sections (but three or more may be necessary to capture all of the relevant information) and insert each into your lab report.
6. Use your table to identify when the flags (i.e., status bits Z and N) change and specify why they are set at a particular time. **Annotate your table and the Quartus-generated simulation** to indicate what is going on during each step of the simulation. You do **not** have to include the entire simulation in your submitted lab document, just enough to prove that you understand what is happening.
7. Use your table and the simulation to identify where data is being written into memory or read from memory. Pay close attention to the address bus. **Annotate your table and the Quartus-generated simulation** with this information.
8. Modify the data in sram.mif and repeat steps 4-7 above. Note: When you change data in either the eprom.mif file or the sram.mif file, you **must recompile** the computer.bdf file. Include the simulation results in your submitted lab document.
9. Compile all the documents described above into your lab document.

2. NEW PROGRAM CREATION

1. Write a program to find the difference between a series of pairs of numbers; store the results in an output table. The input table is arranged as shown in Table 1 and as follows: OutAddr, TabSize, Num1a, Num1b, Num2a, Num2b, etc. OutAddr is the starting address of the output table. TabSize is the number of pairs. For example, subtract Num1b from Num1a, i.e., store the result of Num1a minus Num1b at address OutAddr. Just do the subtraction operation; you do not need to worry about overflow. When you have completed the data processing, execute an endless loop (like a dog chasing its tail).

Table 1: Input Data Table

OutAddrLow
OutAddrHigh
TabSize
Num1a
Num1b
Num2a
Num2b
Num3a
Num3b
Num4a
Num4b
* * *
NumTabSizea
NumTabSizeb

2. Assume that the input table is in ROM starting at \$0E37. It could just as well be in SRAM at address \$1F5C, for example. (Note that the output table must be in SRAM, since you cannot write to ROM.) To test your program, you will have to create the program and some data in the ROM (eprom.mif) file as well as set up the SRAM (sram.mif) file to store your results. (If the input data was instead in the sram.mif file, your program should still work, as long as you knew the starting location.)

3. Use the X register to point to the data that is read from the input table and use the Y register to point to the output table.
4. REG A should be used as a loop counter. REG B should be used for calculations. (Hint: You may need to temporarily store the loop counter in memory, i.e., SRAM, while REG A is used for other purposes.)
5. Install G-IDE-Full-v1.4 (the GCPU integrated Development Environment) and then use the G-IDE to assemble and simulator your program. Continue to test your program until you are satisfied that it is functioning properly. Include a screen shot of your program in G-IDE in your lab report.
6. Hand-assemble your program. Verify that you hand-assembly matches what was produced in rom.mif from G-IDE. (Note that you will be expected to be able to perform hand assembly during your Lab 7 Quiz and during your Final Exam.) Write a "list" file that has addresses, opcodes (machine codes), assembly language instructions, and comments. To create the list file, use `Assembly List.xlsx` (available on our website) or Table 3 on the last page of this document. Copy the completed table into your lab document.
7. Verify the program works by creating an eprom.mif. For this verification, use OutAdd = \$1A37, TabSize = 3, and use three sets of data. Predict the simulation result with a table like that described in Part 1. Annotate this table as in Part 1. Recompile Quartus with your program and eprom.mif. The simulation results and annotations with the instructions should be submitted in your lab document. Your program must work (with **no** changes) for a completely different set of data (in either eprom.mif or sram.mif) file, i.e., do **not** embed the number \$1A37 for the output address or 3 (the table size) in your program. These values should be in the table.
8. Copy the "list" file into your lab document. Also include the annotated table from Part A in your lab document. Submit (through Canvas) this file along with your design archives, and the lab report document. Finally, archive and submit this through Canvas.
9. Once your program works correctly, switch the top-level entity of the GCPU to computer_programming.bdf. You can then perform a full compilation and program the GCPU to your DE-10 Lite. Verify that your program works correctly on the DE-10 Lite by comparing the REGA, REGB, and PC values to your hand-simulation and Quartus simulation.

IMPORTANT NOTES:

- There is **no way** to read the data written to sram.mif.
- If you wish to use variables or store values in your code, you must reference the RAM area of your memory space. RAM is located at \$1000 through \$1FFF. ROM is read only (and is located at \$0 through \$0FFF).
- When you change data in either the eprom.mif file or the sram.mif file, you **must recompile** the computer_simulation.bdf file before trying to simulate.
- **Never** program your DE10-Lite with computer_simulation.bdf compilation. (Only program after the computer_programming.bdf.)

PRE-LAB PROCEDURE SUMMARY

1. Analyze the machine code and execution of a pre-written GCPU program in § 1.
2. Write, hand-assemble, and test your own GCPU program in § 2.

IN-LAB PROCEDURE

In the lab quiz, your PI will ask you to write a simple GCPU program and hand assemble it. You will be allowed to use G-IDE for this quiz to help with simulation and debugging.

For your prelab demo, you will demonstrate your Part 2 program running on your DE10-Lite **and** in G-IDE. Feel free to adjust which 7 segments are used for REGA and REGB if some of the segments on your board are no longer functioning. (Please tell your PI if you have any bad segments on your 7-segment displays.)

Table 2: Sample assembly hand-simulation table format for Prelab Part 1, #3 (and elsewhere). Notes: All values in hexadecimal except Z & N. The first row past the header row is an example row. This table is available on our website in `Lab7 PartA.xlsx`.

Address(es) [$]	**Opcodes [$]**	**Instruction**	**A [$]**	**B [$]**	**X [$]**	**Y [$]**	**Z**	**N**	**PC [$]**	**Comments**
0000-0002	08 00 18	LDX #$1800	00	00	1800	0000	1	0	0003	

Table 3: Sample assembly list table format for Prelab Part 2, #5. This table is also available on our website in `Assembly List.xlsx` and `Assembly List.docx`.

Addr [$]	**Opcodes [$]**	**Assembly Instruction**	**Comments**
0000-0002	08 00 18	LDX #$1800	