Jason Langowski  
Timothy Ginder  
Danielle Tucker

Project 4 – LC + Simulator  
TCSS 371  
2012 November  
Team Name: TeamAwesome

Overview:

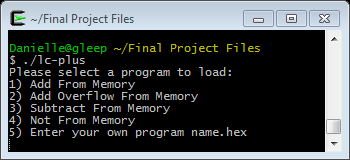
The purpose of this set of programs is to simulate a basic computer’s operations, and in the process of creating them, gain a deeper understanding of the logic and underpinnings of machine architecture. The LC + set of programs include a user interface which serves as the interaction between the core of the simulated computer and the user.

Based upon the project specifications, the LC + has the following machine codes available to it: ADD, AND, BR (with condition codes negative, positive, zero, overflow, and carry), JMP, LD, LDR, LEA, NOT, ST, STR, SUB (subtraction), and HALT. The LC+ user interface accepts .hex files only and assumes the first instruction is to be loaded at Memory = 0x0 and PC starts at 0x0. Four programs have been included with this simulator to illustrate various capabilities of the machine.

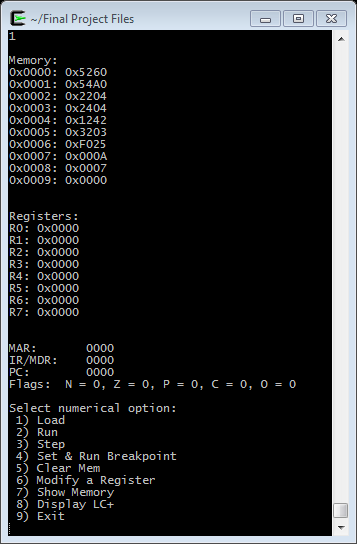
How Run the Program:

For user convenience, a make file has been included to aid in compiling the programs on your computer. Make sure all files for this project are in the same directory and then type “make” (without the “ “) in your console for the files to be compiled for you. You then run the program from the command line: ./lc-plus.exe (you may not need the .exe extension depending on your machine.)

You will first be prompted to enter a program to load into the machine. Four have been provided for you and are described below in the “Example Programs” section of this document. You could also choose to load your own program (Option 5).



After your selection you will be shown the current state of the machine. You will then have several options for how to proceed. For this example, the first option (Add From Memory) has been chosen:



You can now choose to Load, Run, Step, Set & Run Breakpoint, Clear Memory, Modify a Register, Show Memory, Display LC+, or Exit.

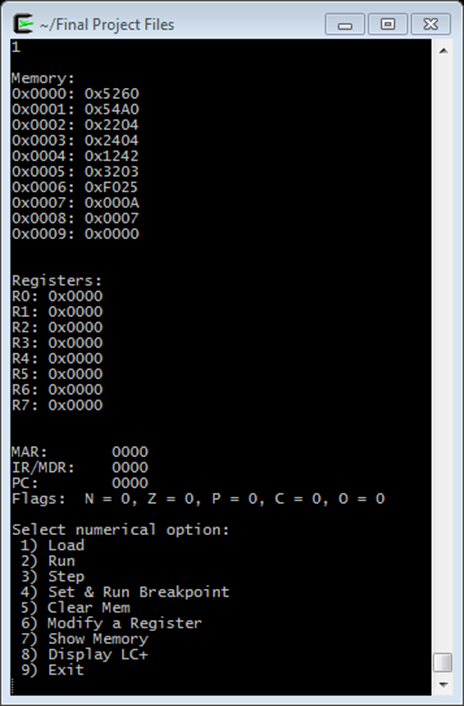
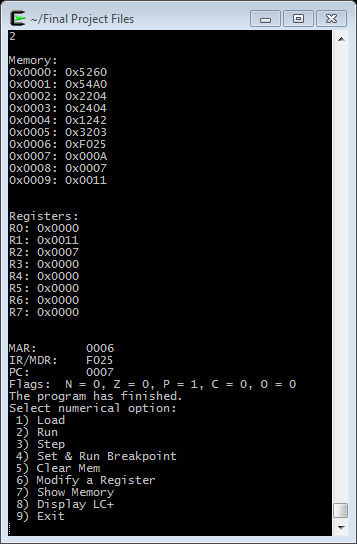
*Load:* Load another program into the LC+ simulator. This will replace the currently loaded program as all programs are assumed to start at memory value 0x0. The PC will be reset to 0x0 for convenience.  
*Run:*  Runs the current program until the HALT (hex: F025) command is encountered. Once the program has run, you will need to select Load again if you wish to re-run your program.  
*Step:* Step through the loaded program one instruction at a time so you can see registers, memory, flags, MAR, IR/MDR, and PC being updated at each instruction. After each step, you will be prompted with this menu again. Do not set the breakpoint past the HALT command.  
*Set & Run Breakpoint:* This option allows the user to set a breakpoint and run until that breakpoint has occurred. The breakpoint should be represented as an integer.  
*Clear Memory:* Clears the memory to 0x0 between the indices you specify. The memory location should be represented as an integer.  
*Modify a Register:* Change the value of a register by providing a value in hex to replace the current value. *Show Memory:* Shows only the memory between the values you provide. The memory locations should be represented as integers.  
*Display LC+:* Refreshes the display with the current values which are in the computer  
*Exit:* Quit the program. The final state of the machine will be displayed before exiting.

Example Programs:

The LC+, as mentioned before, expects inputs in the form of a .hex file for the programs. Four programs have been included with the LC + and are described and illustrated in detail below. In addition to the programs being included in Hex (.hex) the associated Assembly (.asm), Symbol Table (.sym), and List File (.lst) have been included for the user’s convenience.

*Add From Memory:* This program loads two numbers from memory, adds them together, and stores the result in a third memory location.

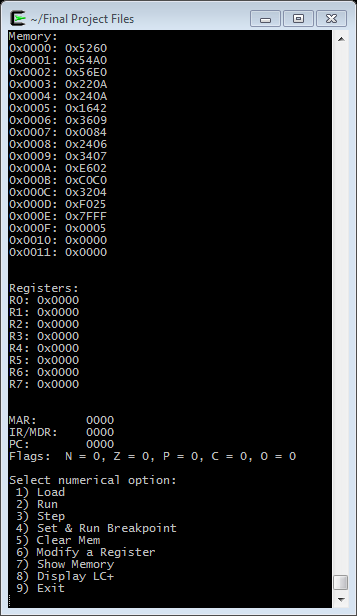
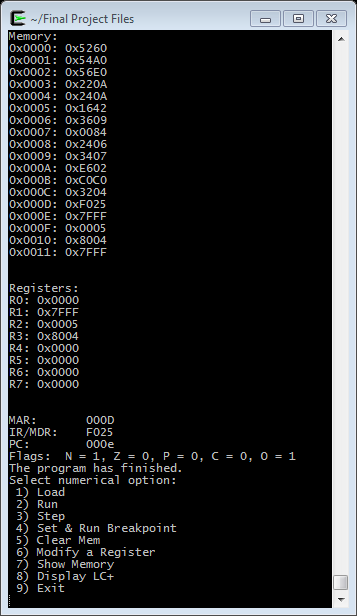
Program Before Execution: Program After Execution:

In this program the number 10 (0x0A) and 7 (0x07) have been loaded from memory 0x07 and 0x08 respectively (yellow arrow). They are then added together, and the result is in R1 and loaded into memory location 0x09 (green arrows).

*Add Overflow From Memory:* This program loads two numbers from memory which will cause overflow when added and puts the result in a third memory location. This program branches on overflow to store the first operand to another memory location.

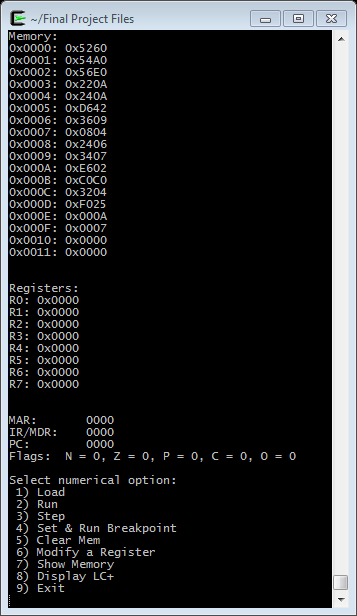
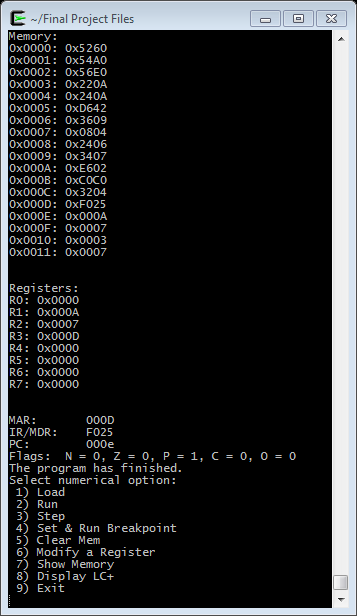
Before Run: After Run:

In this program the values in memory 0x00E and 0x00F (yellow arrows) are to be loaded into registers and added together which will cause an overflow (green arrow). The result of the addition will be loaded into Register 3 and loaded into memory location 0x010 (orange arrows) . Because of the overflow, there will be a branch in which the first operand (0x7FFF) is loaded into memory at 0x0011 (blue arrows).

*Subtract From Memory:* This program subtracts two positive values from memory and stores the result in a third memory location. If the result is negative, the first operand is stored in another memory location; otherwise the second operand is stored.

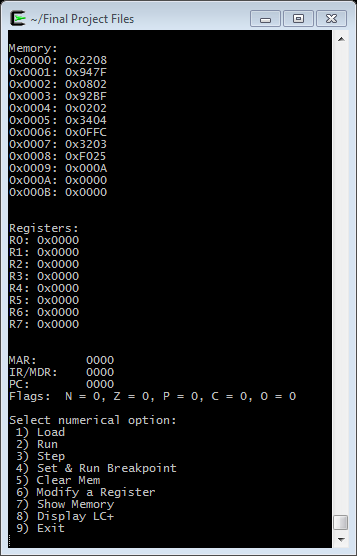
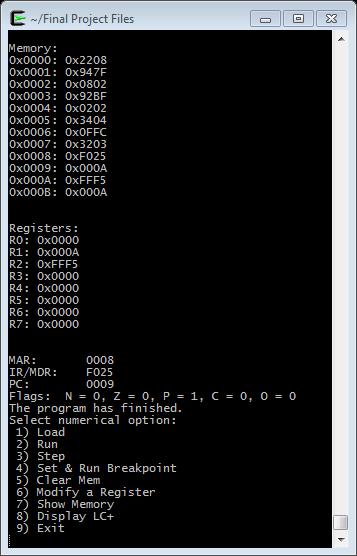
Before Run: After Run:

In this example 0x00A (ten) and 0x007 (seven) are loaded from memory location 0x0E and 0x0F then subtracted from each other. The result is stored in memory location 0x010 (orange arrow). If the result is positive (green arrow), which it is, the second operand is stored in memory location 0x011 (blue arrow). Had the result been negative, the first operand would have been stored in memory location 0x011.

*Not From Memory:* This program loads a positive number from memory, performs a logical not, branches on negative to store the operand to a memory location. It then performs a logical not on the result, branches on positive and stores the result to a memory location.

Before Run: After Run:

In this example, a value is loaded from memory location 0x09 (yellow arrow) and then a logical NOT is performed and stored in Register 2. If the result is negative (which it is) then the value is stored in memory location 0x0A (orange arrows). Then the value in R2 has another logical NOT performed and if the result is positive (which it should be) the result is store in memory location 0x0B (green arrows).

Team Workflow:

To begin this project, Jason, Tim, and Danielle got together and white boarded out the project so that everyone was clear about the project’s scope and which individual components would be “in charge of” which particular operations of the machine cycle. To do this we started by drawing a diagram on the whiteboard of all of the files required by the project specifications and drew arrows to connect the flow in and out of each program/class for one machine instruction. This helped us clarify which structures needed access to memory, to the ALU, the controller, and other parts of the machine. We determined that the CPU would be responsible for creating the memory, fetching the instruction, holding the value of the PC, MAR, MDR and IR, and instructing the controller to execute the command in the MDR. The controller would be in charge of creating the registers and ALU, parsing the opcode from the MDR, and finally setting up and executing the operations. The ALU, we decided, was in charge of performing the operations the controller tells it to and returning a result which the controller then puts in the correct location. Finally, we decided that the LC-plus was in charge of creating the CPU and the debug monitor while the debug monitor was in charge of all of the user interactions such as loading a file, stepping through a program, and displaying memory.

Once the operations of all components were clear, we each took one part of the computer to code. We utilized Jason’s ALU from the previous project as the core of the computer. Jason was also in charge of creating the memory, registers, and controller for the project. Danielle was in charge of the CPU, makefile, the assembly programs, and this write up. Tim was in charge of the user interface and the debug monitor. We were all in charge of debugging our component and the larger program as we went along. We coded in parallel as much as we could at our first meeting which helped clarify our thinking about how the components interacted with each other. Each component was debugged independently as much as possible before passing it off to the rest of the team. By the end of the first meeting we had the memory, registers, CPU, ALU, and three of the four assembly programs completed. We then decided to meet again the next week with our independent code as complete as we could make it.

At our second meeting we attempted to put all of the pieces together. As many of the components required others to be complete before they could be fully tested, we ran across some compile issues when we first put all of the pieces together. During this meeting we worked together to debug the components and clarify some of the assumptions we had made about how the components worked together. One such issue was caused by our decision that the controller needed a pointer to the CPU but then also the CPU needed a pointer to the controller. This clearly did not work and so we then had to distill down what the minimum the controller needed in its struct, which seemed to be access to the memory and a pointer to the PC. After adjustment to the code, we then used the makefile to compile our code, debugging as we went along. By the end of this second meeting we had a working CPU, Controller, and basic user interface.

With this start, we each took our individual components and made sure that the assembly programs were operating as expected, which required some further modification of the controller and debug monitor. Jason and Tim worked on the user interface and controller some more once all of the assembly programs had been verified to be accurate. Then, the finishing touches were added to make sure the final user experience was what was expected and to add some extra features to the program. Throughout the process everyone was involved in debugging and making sure the operation of the LC + was an accurate reflection of our understanding of how a “real” computer operates.