CIS-11 Project Documentation

**Team A**

**Diego Cordova**

**Yoonsu Cho**

**Raybert Salazar**

**Test Score Calculator**

**5/20/2025**

**Advisor: Kasey Nguyen, PhD**

# Part I – Application Overview

## Objectives

## The objective of this project is to design and implement an LC-3 assembly language application that demonstrates key principles of low-level programming and computer architecture. The project will require the use of memory operations, conditional logic, loops, subroutines, and input/output handling in a cohesive program. It emphasizes structured program design, modular development, and clear documentation.

#### Why are we doing this?

* *This project helps achieve the following objectives:*

Reinforce foundational concepts in computer architecture by implementing them directly in assembly language.

Develop problem-solving skills through pseudocode planning, flowchart design, and memory management strategies.

Build hands-on experience in low-level programming using the LC-3 instruction set.

Prepare students for more advanced topics in systems programming, compiler construction, or embedded systems by establishing a strong understanding of how programs interface with hardware.

Who will benefit?

Instructors can use this project to assess students’ depth of understanding and application of course materials.

Future employers or academic advisors benefit indirectly by reviewing a student’s completed work as evidence of their understanding of computer architecture and assembly language.

What if we don’t do it?

Omitting this project would reduce the course to passive learning, without a way to assess practical understanding..

## Business Process

### **Current Process**

### Right now, most of the work I’ve done in this course has been focused on understanding individual LC-3 concepts through lectures and small assignments. These have helped me learn how to use branching, loops, memory storage, subroutines, and input/output operations, but always in separate examples. Each assignment has had a narrow scope, so I haven’t had the chance to connect all the concepts together into a complete, working program.

### So far, the structure has been helpful for learning the basics, but there hasn’t been a project that simulates how a full program would be designed and built from start to finish.

### **Process with the New Application**

### With this LC-3 project, I now have the opportunity to apply everything I’ve learned in a single, structured program. This project requires me to work as part of a team, plan the full design, document the logic in pseudocode and a flowchart, and then actually implement the working code.

### By doing this, I’m simulating the real process of building software. I’ll be thinking more carefully about the flow of logic, how data is stored and manipulated, and how to break the program into meaningful parts. I’ll also be working alongside others, dividing up responsibilities, and making sure all the pieces fit together correctly—just like a real development team would.

### **How the Application Will Be Used**

### The LC-3 program my team creates will be run in the LC-3 simulator. It will accept input from the keyboard and display output on the screen. Depending on the program design, it might perform calculations, manipulate data, or guide the user through a series of prompts and responses.

### This application will be submitted as part of our final project and will show how well we’ve understood the course material. It’s not just about getting the program to work—it’s about designing it well, documenting it clearly, and working effectively as a team. That’s the value this project brings to me in this course.

## User Roles and Responsibilities

### For this LC-3 project, the system (our program) is being designed, built, and tested by the project team. Each team member takes on one or more specific roles that relate directly to the project’s goals. These roles represent the tasks required to successfully complete the application.

### **1. Program Designer**

### **Objective:** Plan the structure and logic of the LC-3 application.

### Writes out the program objectives in logical order.

### Constructs the pseudocode based on those objectives.

### Works with the flowchart designer to visualize the logic.

### Ensures the logic includes all required LC-3 features.

### **Timing:** Completed early in the project. Guides all later tasks.

### **2. Flowchart Designer**

### **Objective:** Create a visual representation of the program flow.

### Translates pseudocode into a detailed flowchart.

### Coordinates with the designer to ensure accuracy.

### Makes changes as needed if logic is adjusted during development.

### **Timing:** Follows program objective planning. May be revised throughout.

### **3. Assembly Programmer**

### **Objective:** Implement the application using LC-3 assembly language.

### Converts pseudocode into working LC-3 code.

### Handles instruction sequencing, branching, loops, and I/O.

### Uses LC-3 simulator to test individual code blocks.

### **Timing:** Begins after planning; continues throughout the project.

### **4. Tester & Debugger**

### **Objective:** Test the program in the LC-3 simulator and find bugs.

### Runs the program under different scenarios.

### Identifies and fixes logical and runtime errors.

### Verifies that outputs are correct for various inputs.

### **Timing:** Starts after initial implementation; repeats before final submission.

### **5. Documentation Lead**

### **Objective:** Manage and compile all project documentation.

### Organizes the program purpose, objectives, and process into a single document.

### Includes pseudocode, flowchart, and final reflections.

### Ensures formatting is clean and consistent across all submissions.

### **Timing:** Ongoing; final polish before submission.

### **Workflow and Task Relationships**

### The **Program Designer** and **Flowchart Designer** work closely together in the early phase.

### The **Assembly Programmer** follows the design and refers to both pseudocode and flowchart.

### Once the initial version is ready, the **Tester & Debugger** runs and validates the program.

### Throughout, the **Documentation Lead** collects work from all roles to build the final report.

## Production Rollout Considerations

**Rollout Strategy**

* **Final Testing**: Before submission, the team will run the program multiple times using the LC-3 simulator to verify that inputs are processed correctly and outputs are accurate under different conditions.
* **Bug Fixes and Validation**: Any unexpected behavior or logical errors will be identified and resolved by the tester and programmer.
* **Code Review**: All team members will review the final code to confirm that it adheres to LC-3 syntax, course requirements, and best practices.
* **Documentation Review**: The documentation will be finalized, including objectives, pseudocode, flowchart, and team responsibilities.

**Terminology**

| **Term** | **Definition** |
| --- | --- |
| **LC-3** | A simplified educational computer architecture used for learning assembly. |
| **Simulator** | The software environment where LC-3 programs are assembled and run. |
| **Pseudocode** | A simplified, plain-English representation of a program’s logic and steps. |
| **Flowchart** | A visual diagram that maps out the program’s logical structure and execution. |
| **Register** | A small storage location in the CPU used to hold data temporarily. |
| **Subroutine** | A block of code that performs a specific task and can be reused within the program. |
| **Memory-mapped I/O** | Technique where input/output devices are assigned specific memory addresses. |
| **Instruction Set** | The complete list of operations supported by the LC-3 architecture. |
| **Branching** | Changing the sequence of execution based on a condition |
| **Halt Instruction** | The LC-3 instruction that stops program execution (HALT). |

# Part II – Functional Requirements

## ****Part II – Functional Requirements****

This LC-3 program is designed to input five test scores from the user, store them in memory, and then compute and display the **maximum score**, **minimum score**, **average score**, and **corresponding letter grade** based on a grading scale. The program will use subroutines, stack operations, pointers, conditional branching, and ASCII conversion to meet the outlined requirements.

### **Functional Requirements**

1. **Input Phase**
   * The program shall prompt the user to enter five test scores (one at a time).
   * Each score shall be read from the keyboard as ASCII characters and converted to binary values.
   * Converted values shall be stored sequentially in an array in memory using a pointer.
2. **Data Storage and Memory Allocation**
   * The program shall reserve a memory block to store all five scores.
   * A pointer shall be used to keep track of the current position in the array.
   * Appropriate .FILL and .BLKW directives shall be used to allocate memory.
3. **Computation Phase**
   * The program shall call subroutines to:
     + Find the **maximum** score.
     + Find the **minimum** score.
     + Compute the **average** score (as an integer).
   * Arithmetic operations shall be performed using LC-3 instructions (ADD, AND, NOT, etc.).
4. **Letter Grade Assignment**
   * Based on the calculated **average score**, the program shall assign a letter grade as follows:
     + 90–100 → A
     + 80–89 → B
     + 70–79 → C
     + 60–69 → D
     + 0–59 → F
   * Conditional branching shall be used to implement this grade classification logic.
5. **Output Phase**
   * The program shall display:
     + The **maximum**, **minimum**, and **average** scores.
     + The corresponding **letter grade**.
   * Outputs will be shown using appropriate system calls like PUTS and OUT.
   * All output text will be defined in memory using .STRINGZ.
6. **Subroutines**
   * The program shall contain at least two subroutines:
     + One for calculating the maximum or minimum score.
     + One for computing the average.
   * Subroutines will use the stack for save/restore of registers and maintain local scope.
7. **Stack Management**
   * The program shall implement a stack using PUSH and POP macros or equivalent LC-3 instructions.
   * Before calling a subroutine, relevant registers shall be saved.
   * After subroutine execution, saved registers shall be restored.
8. **Branching and Looping**
   * The program shall use loops (via BR and BRnzp) to:
     + Read multiple inputs.
     + Iterate through the array during computations.
   * Conditional branching will also be used for grading decisions and control flow.
9. **Overflow Management**
   * The program shall ensure that inputs do not exceed valid range (0–100).
   * Any invalid input shall be detected and rejected with an appropriate message (optional for extra credit).
10. **ASCII Conversion**

* Input scores will be typed as ASCII digits.
* The program shall convert ASCII to binary integers before computation.
* During output, binary integers will be converted back to ASCII strings for display.

1. **System Calls**

* The program shall make use of system calls:
  + GETC for input
  + OUT for single character output
  + PUTS for displaying strings
  + HALT to terminate the program

**Statement of Functionality**

The **Test Score Calculator** program will execute the following features in a consistent, predictable, and fully defined manner. Each function described below defines the total and complete functionality expected from the program. Any feature not explicitly stated here is considered out of scope for this implementation.

**1. Input Handling**

* The program will prompt the user, one at a time, to enter **five individual test scores**.
* Each score will be entered via keyboard using ASCII characters (two-digit or three-digit numbers).
* The program will **convert each ASCII input to a binary integer value** before performing calculations.
* Only valid integers in the range of **0 to 100** will be accepted.
* (Optional: if a score is outside this range, the user may be prompted to re-enter the value.)

**2. Data Storage**

* Each score will be stored sequentially in a **pre-allocated memory array** using a **pointer-based storage technique**.
* Scores will be stored in memory for later retrieval during calculations.

**3. Score Calculations**

* The program will calculate:
  + The **maximum** of the five scores.
  + The **minimum** of the five scores.
  + The **average**, computed as the integer division of the total by 5.

**4. Letter Grade Assignment**

* The program will determine a **letter grade** based on the average score using this scale:
  + 90–100: A
  + 80–89: B
  + 70–79: C
  + 60–69: D
  + Below 60: F
* This will be implemented using **conditional branching**.

**5. Output Display**

* The program will display:
  + “Max Score: [value]”
  + “Min Score: [value]”
  + “Average Score: [value]”
  + “Letter Grade: [A-F]”
* All values will be converted back from binary to **ASCII characters** for display.

**6. Subroutine Integration**

* The program will include at least **two subroutines**:
  + One for calculating the average.
  + One for finding the maximum or minimum value.
* Subroutines will use **stack-based save/restore operations**, including **PUSH and POP**.

**7. Branching and Control Flow**

* The program will use **loops** to iterate through the scores.
* **Conditional branching** will control score comparison, letter grade assignment, and input handling.

**8. System Services**

* The program will use LC-3 **TRAP instructions** (GETC, OUT, PUTS, HALT) for input/output and termination.

**Scope**

This project consists of a **single development phase** and will deliver all functionality listed above in one complete submission.

Features included in this phase:

* Full user input for five test scores
* Binary storage and ASCII conversion
* Max, min, average, and letter grade computation
* Stack-based subroutine calls
* Use of branching, pointer, and appropriate LC-3 directives

No future expansion phases are currently planned. The scope is limited to handling **exactly five scores** per session.

**Performance**

* The program must handle all five inputs and compute the required results in a **single session**, with no crashes or infinite loops.
* The program should complete all operations in under **2 seconds of simulation time**, assuming correct user input without delays.
* All memory used must remain within the default LC-3 address space and not exceed 100 words.
* Stack operations must not overwrite program memory or I/O memory space.

**Usability**

* The program will provide **clear and simple prompts** during input stages (e.g., “Enter score 1:”).
* Output will be displayed in an **easy-to-read format** using labeled lines for each result.
* The program will not use menus or advanced navigation, but user input will follow a **linear, predictable sequence**.
* ASCII conversion will ensure that numerical values are displayed properly to match user expectations.

# Documenting Requests for Enhancements

There does come a time when the requirements for the initial release of your application are frozen. Usually, it happens after the system acceptance test which is the last chance for the users to lobby for some changes to be introduced in the upcoming release.

Currently, you need to begin maintaining the list of requested enhancements. Below is a template for tracking requests for enhancements.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Enhancement** | **Requested by** | **Notes** | **Priority** | **Release No/ Status** |
| 2025-05-23 | Add input validation for scores outside 0–100 | Team Lead | Prompt user to re-enter invalid scores. Helpful for user error handling. | Medium | Planned for future release |
|  | Display all five scores in final output | Team Member | Adds clarity for the user to see which scores were processed. | Low | |  | | --- | |  |  |  | | --- | | Optional | |
|  | Allow user to enter more than 5 scores dynamically | Instructor | Upgrade logic to handle flexible array size using pointer iteration. | High |  |

# Part III – Appendices

### **Appendix A – Sample Input and Expected Output**

**Test Case**:  
User inputs the following test scores in order:

52, 87, 96, 79, 61

**Expected Output:**

Max Score: 96

Min Score: 52

Average Score: 75

Letter Grade: C

This test case is used for verifying that logic for max, min, average, and grading scale is implemented correctly.

### **Appendix B – ASCII to Integer Conversion Chart**

| **ASCII Character** | **LC-3 Binary Value** | **Decimal Value** |
| --- | --- | --- |
| '0' | x0030 | 48 |
| '1' | x0031 | 49 |
| '2' | x0032 | 50 |
| '3' | x0033 | 51 |
| '4' | x0034 | 52 |
| '5' | x0035 | 53 |
| '6' | x0036 | 54 |
| '7' | x0037 | 55 |
| '8' | x0038 | 56 |
| '9' | x0039 | 57 |

**Note**: To convert ASCII digit input to integer value, subtract ASCII code for '0' (x30 or decimal 48) from the input character.

### **Appendix C – Subroutine Plan**

**Subroutine 1: FIND\_MAX**

* Input: Pointer to score array
* Output: Maximum value
* Description: Iterates through all 5 values and tracks the maximum score.

**Subroutine 2: FIND\_MIN**

* Input: Pointer to score array
* Output: Minimum value
* Description: Same as above, but for the minimum.

**Subroutine 3: CALC\_AVG**

* Input: Pointer to score array
* Output: Integer average (total / 5)
* Description: Adds all five scores, divides total by 5.

Each subroutine will use stack-based PUSH/POP operations to save and restore any modified registers.

### **Appendix D – Memory Map Plan (Initial)**

| **Address** | **Label** | **Description** |
| --- | --- | --- |
| x3000 | START | Entry point of the program |
| x3100 | SCORE\_ARR | Start of array for storing scores |
| x3200 | PROMPTS | User prompt strings (.STRINGZ) |
| x3300 | OUTPUT\_STR | Output message templates |

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Appendix E – Custom Subroutine: Array Insertion Helper (STORE\_BY\_INDEX)**

**Purpose**

This subroutine allows the program to store an integer value into a specific index of an array using globally accessible parameters, making it modular and flexible without passing memory addresses directly.

**How It Works**

* **Inputs (via global labels):**
  + VAL\_SLOT: Contains the value to store
  + IDX\_SLOT: Contains the index at which to store the value
* **Global Dependency:**
  + ARRBASE: A .FILL pointing to the start of the SCORES array
* **Process Flow:**
  + Save registers (R0–R5, R7) using fixed memory labels
  + Load VAL\_SLOT into R0 and IDX\_SLOT into R1
  + Load the array base address from ARRBASE into R4
  + Calculate the memory address: SCORES[R1]
  + Store the value into that location
  + Increment the index and update IDX\_SLOT
  + Restore registers and return

**Caller Usage Example (Inside Main Code)**

; Assume R5 holds the value to store, R0 holds index

ST R5, VAL\_SLOT ; pass value

ST R0, IDX\_SLOT ; pass index

JSR STORE\_BY\_INDEX ; write value into SCORES[i]

LD R0, IDX\_SLOT ; update i ← i + 1

**Rationale & Best Practice**

* **Why not pass pointers?**  
  Since ARRBASE holds the base address of the array, there's no need to compute or pass addresses in registers. This keeps the interface clean.
* **Why use mail-slot style (VAL\_SLOT, IDX\_SLOT)?**  
  Using fixed global memory slots mimics parameter passing by reference and simplifies modularity without needing stack frames or parameter-passing conventions.
* **Why not use stack for register saving?**  
  Because we assume no nested function calls, we use fixed save locations to keep the subroutine light and simple.

A computer screen shot of a computer flowchart

AI-generated content may be incorrect.

**Appendix F – Input Conversion and Integration**

**Subroutine: READ\_3DIGIT\_NUMBER**

This helper allows the user to input a number from 000–999 using three digit characters and stores the corresponding binary value.

**Steps:**

1. Display prompt (e.g., "Enter a number between 000–999...")
2. Read three characters (hundreds, tens, ones) using GETC
3. Convert each from ASCII by subtracting 48
4. Convert digits into binary value:
   * Multiply hundreds digit by 100
   * Multiply tens digit by 10
   * Add all together
5. Store final result into INPUT\_INTEGER

**Integration Pattern**

assembly

JSR READ\_3DIGIT\_NUMBER ; result stored in INPUT\_INTEGER

LD R5, INPUT\_INTEGER ; load result

ST R5, VAL\_SLOT

ST R0, IDX\_SLOT

JSR STORE\_BY\_INDEX

This sequence takes the input and inserts it into the array.

## Flow chart or pseudo-code.

General Flow Chart ( The flowchart presents a broad architectural view of a grading utility)

A diagram of a software

AI-generated content may be incorrect.

The diagram sketches the life-cycle of a compact I/O helper routine. (Prompt for Scores function)

A diagram of a software flow

AI-generated content may be incorrect.