CIS-11 Project Documentation Template

**Team Name**

**Team A**

**Team Members**

**Diego Cordova**

**Yoonsu Cho**

**Raybert Salazar**

**Project Name**

**Test Score Calculator**

**Date**

**05/25/2025**

**Advisor: Kasey Nguyen, PhD**

# Part I – Application Overview

## Objectives

### The purpose for this Test project is to develop a Score Calculator that lets the user to input five test scores from 0 to 100, and receive the grade letter for each one, the minimum score, maximum score, and average score. The application is designed to simplify the process of grading for the users by making it quicker than doing it manually, and more accurately (this is done by validating the input). Anyone could use the application, whether it is an instructor grading their students’ work, a student that wants to check their grade, or someone that just wants to check it out.

#### Why are we doing this?

### This project helps achieve the following objectives:

* **The application will simplify the process of grading test scores. Instead of manually calculating the minimum, maximum, average, and their letter grades. This saves time and time is money. The program also reduces errors. It does this by validating whether the user inserted the correct values (0 – 100). If it catches an error, then it will ask you to insert a valid score. Now when it comes to manually do it, we tend to make mistakes and not catch them.**

### Why work on a project like this?

### We are working on a project like this to develop our skills further, whether it be for flowchart design or problem-solving. If we do it later then our skill may fade away due to the lack of practice. And if we don’t do it at all, then we will miss the opportunity to have firsthand practice.

### Benefactors

* **There are a lot of users that will benefit from this project: Instructors (to grade more efficiently), future student programmers (They will be able to get more of an idea of how to structure their programs), regular students (To calculate their own grade), and also for the everyday people that just want to try it out. I believe that the benefactors will think of this as an important improvement because it will save them time. One example can be for instructors, if this application saves them time to grade, that just gives them more time to teach the coming generations, which are our “future.” I do not think we should work on a different project because this is something necessary that cannot be overlooked due to the previously stated fact.**

## Business Process

### Current business process

### Until now, most of what this class has been focusing on is learning individual LC-3 concepts through our activities, and assignments. These have helped me get experience with input/output, loops, branches, and subroutines. These exercise have helped me learn the basics, but now it is the time to put it all together.

### Process with the new program

### I am now taking all of my LC-3 knowledge I have built and applying it to one program, with real planning involved. I have worked with my team to come up with a design, draw out a flowchart, and break down the program to small parts to be able to understand it better. We each have our assigned tasks to do and to make sure it all fits well. This process feels like what is done in a job, which is thinking about how the user interacts with the program, how to organize the code, and how to catch the errors.

### Process with the new program

### Our program will run in the LC-3 simulator. It will take five test scores from the user, check that the inputs are valid, and then calculate and display the minimum, maximum, average, with their corresponding letter grades. Our final goal is not to just make the program work, but to show that we understand what we have been doing this semester. That is what makes this a valuable experience.

## User Roles and Responsibilities

### User Roles

### For this project, our program is being designed, built, and tested by the project team. Each member takes on multiple roles related to the project’s goal.

### Responsibilities

## 1. Program Designer Objective: Plan out how the LC-3 program should work from start to finish.

## Breaks down what the program needs to do and lists the goals clearly.

## Writes the pseudocode to show the program's step-by-step logic.

## Collaborates with the Flowchart Designer to map out the structure.

## Makes sure the plan includes key LC-3 features like loops, branches, and input/output. Timing: Done early on to guide the rest of the project.

## 2. Flowchart Designer Objective: Build a visual layout of the program's logic.

## Turns the pseudocode into a detailed flowchart showing the program flow.

## Works closely with the Program Designer to keep everything accurate.

## Updates the flowchart if the logic changes during coding. Timing: Created after pseudocode; updated as needed.

## 3. Assembly Programmer Objective: Write the program in LC-3 assembly and get it working.

## Translates the plan into real LC-3 code.

## Handles branching, looping, memory storage, and user input/output.

## Tests each part using the LC-3 simulator to make sure it works as expected. Timing: Starts after planning and continues until the program is complete.

## 4. Tester & Debugger Objective: Make sure the program runs correctly and fixes any issues.

## Tries out different test cases and input values.

## Finds bugs and works with the programmer to fix them.

## Checks that all outputs are accurate and match what’s expected. Timing: Begins once the main code is written; repeated before final submission.

## 5. Documentation Lead Objective: Put together all the written parts of the project.

## Organizes everything into one final report, including goals, logic, flowchart, and code.

## Keeps documentation updated as the project progresses.

## Makes sure the final version is clean, clear, and ready to turn in. Timing: Ongoing throughout the project; final cleanup before submission.

## Workflow and Task Relationships

## The Program Designer and Flowchart Designer work together to plan out the structure.

## The Assembly Programmer uses the flowchart and pseudocode to build the actual program.

## The Tester & Debugger runs the program and helps fix any issues that come up.

## The Documentation Lead stays in the loop the whole time, collecting materials and preparing the final document.

## Production Rollout Considerations

## Rollout Strategy

## Final Simulation Testing: Before turning in the project, the team will run several full tests using the LC-3 simulator to make sure all inputs are handled properly and that the program produces the correct results in all test cases.

## Debugging and Logic Checks: Any issues that come up—such as incorrect output, input errors, or logic bugs—will be tracked and fixed by the tester and programmer working together.

## Team Code Review: Everyone on the team will go over the final version of the code to ensure it follows LC-3 rules, meets all class requirements, and is written in a clear, organized way.

## Final Documentation Review: The full documentation package—program goals, pseudocode, flowchart, role breakdowns, and final reflections—will be proofread and polished to make sure everything is complete and ready for submission.

## Terminology

| **Term** | **Definition** |
| --- | --- |
| **LC-3** | A simplified, educational computer architecture used to teach assembly language programming. |
| **Simulator** | A software tool that assembles and runs LC-3 programs, allowing users to test and debug code. |
| **Pseudocode** | A plain-English outline of a program's logic and steps, used for planning before actual coding. |
| **Flowchart** | A visual diagram that shows the logical flow and structure of a program. |
| **Register** | A small, fast memory location inside the CPU used to temporarily store data. |
| **Subroutine** | A reusable block of code that performs a specific task and can be called from multiple points in a program. |
| **Memory-mapped I/O** | A method where input/output devices are controlled by reading and writing to specific memory addresses. |
| **Instruction Set** | The full set of commands and operations supported by the LC-3 processor. |
| **Branching** | Changing the flow of a program by jumping to a different instruction based on a condition. |
| **Halt Instruction** | The LC-3 command (HALT) that stops the program from running. |

# Part II – Functional Requirements

This LC-3 program is designed to take five test scores from the user, store them in memory, and then calculate and display the maximum, minimum, average, 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 requirements.

1. **Input Phase**
   * The program will ask the user to enter five test scores, one at a time.
   * Each score will be entered as ASCII characters and converted to binary values.
   * The converted values will be stored sequentially in an array in memory using a pointer.
2. **Data Storage and Memory Allocation**
   * The program will reserve memory to store all five scores.
   * A pointer will keep track of the current position in the array.
   * .FILL and .BLKW directives will be used for memory allocation.
3. **Computation Phase**
   * The program will use subroutines to:
     + Find the maximum score.
     + Find the minimum score.
     + Calculate the average score as an integer.
   * Arithmetic operations will use LC-3 instructions like ADD, AND, NOT, etc.
4. **Letter Grade Assignment**
   * Based on the average score, the program will assign a letter grade:
     + 90–100 → A
     + 80–89 → B
     + 70–79 → C
     + 60–69 → D
     + 0–59 → F
   * Conditional branching will handle the grade assignment.
5. **Output Phase**
   * The program will display:
     + The maximum, minimum, and average scores.
     + The corresponding letter grade.
   * Outputs will use system calls like PUTS and OUT.
   * All text will be defined in memory using .STRINGZ.
6. **Subroutines**
   * The program will have at least two subroutines:
     + One for finding the maximum or minimum score.
     + One for calculating the average.
   * Subroutines will use the stack for saving and restoring registers.
7. **Stack Management**
   * The program will implement a stack using PUSH and POP macros or equivalent LC-3 instructions.
   * Registers will be saved before calling a subroutine and restored afterward.
8. **Branching and Looping**
   * The program will use loops (with BR and BRnzp) to:
     + Read multiple inputs.
     + Iterate through the array for computations.
   * Conditional branching will be used for grading and controlling flow.
9. **Overflow Management**
   * The program will ensure that inputs stay within the valid range (0–100).
   * Any invalid input will be rejected with an appropriate message (optional for extra credit).
10. **ASCII Conversion**

* Scores will be typed in as ASCII digits.
* The program will convert ASCII to binary integers for computation.
* During output, the binary integers will be converted back to ASCII for display.

1. **System Calls**

* The program will use system calls for:
  + GETC to get input.
  + OUT to output a single character.
  + PUTS to display strings.
  + HALT to end the program.

## Statement of Functionality

The **Test Score Calculator** program will run with clear, consistent behavior. Everything it’s supposed to do is listed below. If it’s not here, it’s not part of this version of the program.

**1. Input Handling**

* The program will ask the user to enter five test scores, one at a time.
* Each score will be typed in using the keyboard as ASCII digits (can be two or three digits).
* These ASCII inputs will be converted into binary integers before any calculations happen.
* Only numbers from 0 to 100 will be accepted.
* *(Optional: if the input is out of range, the user might be asked to enter it again.)*

**2. Data Storage**

* Each score will be saved in a memory array using a pointer to keep track of where to store each one.
* These saved scores will be used later for the calculations.

**3. Score Calculations**

* The program will figure out:
  + The highest score (max).
  + The lowest score (min).
  + The average, calculated by adding all five and dividing by 5 (integer division).

**4. Letter Grade Assignment**

* Based on the average score, the program will assign a letter grade:
  + 90–100 = A
  + 80–89 = B
  + 70–79 = C
  + 60–69 = D
  + Below 60 = F
* This will be done using conditional branching.

**5. Output Display**

* After calculations, the program will display:
  + “Max Score: [value]”
  + “Min Score: [value]”
  + “Average Score: [value]”
  + “Letter Grade: [A–F]”
* All output values will be converted from binary back to ASCII before being shown.

**6. Subroutine Integration**

* The program will include at least two subroutines:
  + One to calculate the average.
  + One to find either the max or min value.
* These subroutines will save and restore registers using stack operations (PUSH and POP).

**7. Branching and Control Flow**

* Loops will be used to go through the scores for input and calculations.
* Conditional branches will handle score comparisons, grade decisions, and other logic.

**8. System Services**

* The program will use LC-3 system calls:
  + GETC for getting input
  + OUT for showing a single character
  + PUTS for displaying text
  + HALT to end the program

## Scope

## This project will be built and delivered in one single phase. Everything listed in the functionality section will be included in this one final version.

## Features included in this phase:

## Taking in five test scores from the user

## Storing values in binary and converting between ASCII and binary as needed

## Calculating the max, min, average, and letter grade

## Using stack-based subroutines for calculations

## Applying branching, pointers, and the correct LC-3 directives

## The program is designed to handle exactly five scores per run. There’s no plan to expand beyond that in the future.

## Performance

## The program must take all five test scores and complete all calculations (max, min, average, letter grade) in one session without crashing or getting stuck in an infinite loop.

## Assuming the user enters valid input without pausing too long, the program should finish everything in under 2 seconds of simulation time.

## All memory used must stay within the standard LC-3 address space and should not exceed 100 words total.

## Stack operations should be handled safely—they must not mess with program code or memory-mapped I/O areas.

## Usability

1. The program will show clear and simple prompts when asking for input, like: “Enter score 1:”
2. Results will be displayed in a clean, labeled format so it's easy to read things like the max, min, average, and letter grade.
3. There won’t be any menus or complex navigation—the input process is straight-forward and follows a step-by-step flow.
4. All numbers will be properly converted between ASCII and binary so the scores look right when typed in and shown on screen.

# Documenting 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.  ***Note:***  This is possibly for a future update because our main goal is to release the application for 5 scores. | High | Possibly for a future update. |

# Part III – Appendices

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

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

### 65, 82, 91, 74, 88

### **Expected Output:**

### Max Score: 91

### Min Score: 65

### Average Score: 80

### Letter Grade: B

### This test case checks that the program correctly calculates the max, min, and average scores, and assigns the right letter grade based on the grading scale.

### **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 turn an ASCII digit into its actual number value, subtract the ASCII code for '0' (which is x30 in hex or 48 in decimal) from the input character.

### **Appendix C – Subroutine Plan**

### **Subroutine 1: FIND\_MAX**

### **Input:** Pointer to the array of scores

### **Output:** Highest score

### **What it does:** Loops through all five scores and keeps track of the biggest one.

### **Subroutine 2: FIND\_MIN**

### **Input:** Pointer to the array of scores

### **Output:** Lowest score

### **What it does:** Same idea as FIND\_MAX, but this one looks for the smallest score.

### **Subroutine 3: CALC\_AVG**

### **Input:** Pointer to the array of scores

### **Output:** Average score (integer value)

### **What it does:** Adds up all five scores and divides the total by 5.

### All subroutines will use PUSH and POP to save and restore any registers they change, so everything stays clean when jumping in and out of them.

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

| **Address** | **Label** | **Description** |
| --- | --- | --- |
| x3000 | START | Where the program starts running |
| x3100 | SCORE\_ARR | Beginning of the array that stores scores |
| x3200 | PROMPTS | Prompt messages for user input (.STRINGZ) |
| x3300 | OUTPUT\_STR | Templates for displaying the output |

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

**Purpose:**  
This subroutine stores an integer value at a specific index in the score array. It uses global labels for inputs instead of passing values through registers, making the code cleaner and more modular.

**How It Works:**

* **Inputs (via global labels):**
  + VAL\_SLOT: holds the value you want to store
  + IDX\_SLOT: holds the index where the value should go
* **Global dependency:**
  + ARRBASE: a .FILL pointing to the start of the score array (SCORE\_ARR)

**Steps:**

1. Save registers R0–R5 and R7 using fixed memory locations
2. Load VAL\_SLOT into R0 and IDX\_SLOT into R1
3. Load base address from ARRBASE into R4
4. Calculate the address: R4 + R1
5. Store the value at that memory location
6. Increment the index in IDX\_SLOT by 1
7. Restore all saved registers and return

Caller Usage Example (Inside Main Code)

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

ST R5, VAL\_SLOT ; Store value to insert

ST R0, IDX\_SLOT ; Store index

JSR STORE\_BY\_INDEX ; Insert value into array

LD R0, IDX\_SLOT ; Load updated index i = i + 1

**Why This Approach Works:**

* **No need to pass pointers:**  
  Since the base of the array is stored in ARRBASE, the subroutine can find the right spot without needing address math in the main program.
* **Why use global "mail slots" (VAL\_SLOT, IDX\_SLOT) instead of arguments in registers?**  
  This mimics passing by reference and keeps things modular. It’s also easier to debug and reuse.
* **Why not use the stack to save registers?**  
  Since this subroutine assumes no subroutine calls inside it, we use fixed memory slots to save/restore registers. It keeps things lightweight 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 lets the user input a 3-digit number (from 000 to 999) and stores it as a binary value.

**Steps:**

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

**Integration Pattern**

JSR READ\_3DIGIT\_NUMBER ; Result is stored in INPUT\_INTEGER

LD R5, INPUT\_INTEGER ; Load the result

ST R5, VAL\_SLOT ; Pass the value

ST R0, IDX\_SLOT ; Pass the index

JSR STORE\_BY\_INDEX ; Insert value into the array

This sequence reads the input, converts it, and stores it in the array at the correct index.

## Flow chart or pseudo-code.

### **General Flowchart**: This flowchart provides an overview of the main structure and flow of a grading utility.

### A diagram of a software AI-generated content may be incorrect.

The diagram outlines the life-cycle of a compact I/O helper routine, specifically the "Prompt for Scores" function.

A diagram of a software flow

AI-generated content may be incorrect.