CIS-11 Project Documentation

Team CSS

Aurelia Juan Sindhunirmala
Aurelisa Juan Sindhunirmala
Brittany Hailee Chan

Bubble Sort
May 21, 2025

Advisor: Kasey Nguyen, PhD

Part I – Application Overview

- Implement the Bubble Sort Algorithm: Translate the logic of the bubble sort algorithm into the instruction set of the LC-3 architecture. The Bubble Sort algorithm is a comparison-based algorithm where adjacent elements are compared and swapped if they are not in order. It has an average time complexity of $O(n^2)$.
- Demonstrate understanding of low-level programming by creating input and output functionalities using registers and TRAP instructions, saving and restoring registers, using pointers to indirectly access variables, and utilizing control structures to decide which code to execute.
- Provide a basic sorting utility within the LC-3 environment.

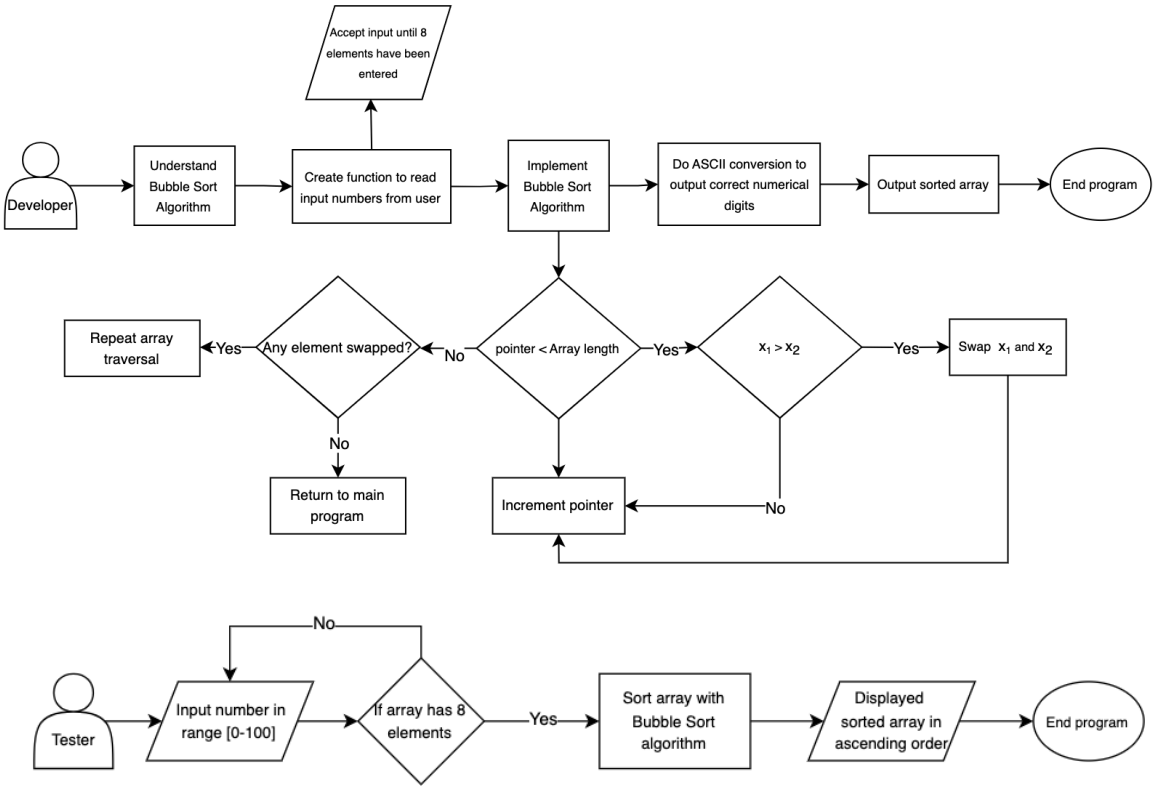
Objectives

- Implement the Bubble Sort Algorithm in LC-3 assembly to sort eight user-input integers in ascending order.
- Provide a foundation for sorting algorithms and realizing the importance of execution time when selecting sorting processes for large-scale implementations.
- Understand the business costs of using certain algorithms for fulfilling processes and the trade-off between runtime and memory usage.
- Conducting the project now will help the business identify whether the Bubble Sort algorithm is suitable in terms of costs and business workflow.
- Demonstrate Low-Level Programming Skills
    - Highlight the subroutine of each functionality—for instance, the swap method in C++.
- Businesses and their software engineers will benefit from this project as they can both identify the best online processes to handle the sorting of data in their databases or other infrastructures, which will also impact the experience of their users.

Business Process

1. Current Process:
    a. Students learn the sorting logic conceptually or through high-level language implementations.
    b. Assignments typically focus on isolated tasks rather than integrated algorithmic problem solving.
2. Enhanced Process:
    a. Incorporating the Bubble Sort project using LC-3 through experimental learning.
    b. Students engage with full program development: input handling, sorting logic, and output formatting.
    c. Practical application fosters better retention and encourages debugging/ problem-solving skills.

User Roles and Responsibilities



1. Student (Developer and Tester)
    a. Understanding the Algorithm: Students must first understand the Bubble Sort Algorithm and the basic concepts of the LC-3 architecture as a prerequisite to using the program effectively.
    b. Program Development: Students use the LC-3 assembly language to implement the Bubble Sort algorithm using appropriate LC-3 instructions for data manipulation, comparisons, and control flow.
    c. Validate sorting logic, handle edge cases, and ensure expected I/O behavior.

2. Instructor (Evaluator)
    a. Guide assignment scope, assess program structure, functionality, and documentation.

Production Rollout Considerations

1. Deliverables: LC-3 object file, source code, documentation, and test log.
2. Deployment: Upload files to the course repository; verify correct address allocation and runtime behavior.
3. Rollout Strategy: Share via course repository; demonstrate in lab session before deadline.

Terminology

1. LC-3: Educational 16-bit instruction set architecture and simulator
2. Bubble Sort: Comparison-based sorting algorithm with $O(n^2)$ average time complexity.
3. Subroutine: Modular block of code invoked through JSR/ JSRR, saved/ restored via stack
4. Stack Operations: PUSH (store register), POP (retrieve register) through R6 as stack pointer
5. ASCII Conversion: Translating between integer values and ASCII character codes for I/O console
6. System Call Directives: TRAP x23 (GETC), TRAP x21 (OUT), TRAP x25 (HALT), etc
7. Opcodes: A part of the instruction that indicates to the assembler what kind of instruction it will be and is mandatory for any instruction. Opcodes include instructions for the starting address of the program (.ORIG), the end of the program (.END), to allocate *n* words of storage (.BLKW), to allocate one word initialized with value *n* (.FILL), and to allocate *n+1* locations initialized with characters and null terminator (.STRINGZ).

Part II – Functional Requirements

Statement of Functionality

The LC-3 Bubble Sort program guides should:

1. Prompt the user to enter eight integers (0-100) via console input, converting ASCII to binary.
2. Store inputs in an array located in memory starting at label ARRAY.
3. Sort the array in ascending order using the Bubble Sort algorithm implemented in a dedicated subroutine.
4. Utilize at least two subroutines (e.g., ReadNumbers, BubbleSort, DisplayNumbers) with JSR/JSRR calls.
5. Implement conditional and iterative branching for loop control and element comparisons.
6. Manage the stack with PUSH/POP of registers around subroutine calls and save-restore of R7.
7. Use pointers to traverse the array and swap elements by address.
8. Handle overflow: ensure inputs outside 0-100 are clamped or re-prompted.
9. Convert sorted integer values to ASCII digits for console display.
10. Display sorted results on the console, one number at a time, separated by spaces.
11. Use appropriate LC-3 directives (.ORIG, .FILL, .BLKW, .END) and system calls (TRAP x21, x23, x25).
12. Test the program with provided values (11, 8, 2, 17, 6, 4, 3, 21) and confirm output: 2 3 4 6 8 11 17 21.

Scope

This project includes developing an LC-3 assembly language program that reads user input, sorts eight integers using the Bubble Sort algorithm, and displays the sorted output in ascending order using ASCII conversion.

I. Phase 1:

Implement input functionality to read and store eight user inputs into an array. This phase includes displaying prompts using Load Effective Address (LEA) and PUTS, capturing input with GETC, and converting from ASCII to binary. Additionally, input validation must ensure that the value is within the acceptable range using branch logic and subtraction-based comparison.

II. Phase 2:

Develop subroutines to handle sorting, one of which is the repeated passes required for the Bubble Sort algorithm to check whether there are still unsorted values. Another subroutine is to check the adjacent elements from one another and perform swapping if needed. The condition for checking adjacent elements (x_1 and x_2) would involve branching, whereby performing (x_2 - x_1) and using *BRn* to see whether the difference between the two values is negative. If the difference is negative, this means that x_1 is bigger than x_2, which means that they must be swapped.

III. Phase 3:

After performing the sorting, this phase focuses on displaying the sorted array, involving array traversal with pointers and ASCII conversion to convert from binary to ASCII. The TRAP x21 instruction (OUT) is used to output the character in R0 to the console. This phase will check whether the sorting algorithm was implemented correctly.

IV. Phase 4:

Test and debug the program with the provided test inputs. This is the final programming phase, and it will determine the final steps required to polish the coding aspect of the project.

Performance

- Algorithm Complexity: $O(n^2)$ for n = 8 (worst case: 196 comparisons)
- Memory usage: Minimal, 8 16-bit words plus stack space (<1 KB)
- Program execution time is negligible due to the small dataset size, but this implementation is not scalable for larger arrays

Usability

- Input prompts clearly indicate the acceptable range (0-100).
- Invalid inputs are detected and re-prompted.
- The final sorted output is displayed clearly, with spacing and a newline.
- Each number is followed by a space and ends with a newline for readability.

Documenting Requests for Enhancements

Date	Enhancement	Requested by	Notes	Priority	Release No/ Status
5/21/2025	Support dynamic array size input	Aurelia Sindhunirmala	Allow user to enter the number of values to sort (e.g., n)	Medium	v.1.1
5/21/2025	Add sorting order	Aurelisa Sindhunirmala	Provide a clear logic of sorting	Medium	v.1.1
5/22/2025	Display error message for input validation	Brittany Chan	Add friendly message to lead user for a seamless performance.	Low	v.1.2

Part III – Appendices

1. Define and reserve memory
    a. ORIG, ARRAY, variables, stack, and pointer
2. Initialize SP (R6) and save return address (R7)
3. ReadNumbers subroutine
    a. Loop 8 times: GETC
    b. Convert to ASCII to integer values
    c. Store validated integers in the array using a pointer offset
4. BubbleSort subroutine
    a. Use flags and loops to implement bubble sort passes
    b. Compare adjacent values, swap if needed
    c. Repeat passes until no swaps occur
5. Swap subroutine:
    a. Save registers using PUSH
    b. Exchange two memory values using load/ store
    c. Restore registers with POP; return
6. DisplayNumbers subroutine:
    a. Loop through array values
    b. Call subroutine for ASCII conversion
    c. Output characters via TRAP x21
7. ASCIIConversion subroutine:
    a. Extract tens and ones digits
    b. Add ASCII ‘0’ to digit values
    c. Output via TRAP x21
8. Finalize program
    a. Restore R7, clean up, HALT

Pseudo-code.

```
BEGIN
  SET R6 <- xFE00       ; initialize stack pointer
  JSR ReadNumbers       ; read 8 inputs
  JSR BubbleSort        ; sort array
  JSR DisplayNumbers    ; output sorted array
```

```
  TRAP x25              ; halt
END

ReadNumbers:
  FOR i = 0 TO 7
    TRAP x23            ; GETC
    JSR ASCIItoInt      ; convert
    STR R0, R1, #0      ; store result at current pointer location
  RETURN

BubbleSort:
  SET swapped = TRUE
  WHILE swapped
    SET swapped = FALSE
    FOR i = 0 TO 6
      LDR R0, R1, #i    ; load array[i]
      LDR R2, R1, #(i+1)
      BRn SKIP_SWAP     ; if R0 <= R2
      JSR Swap          ; swap array[i], array[i+1]
      SET swapped = TRUE
    SKIP_SWAP:
    END FOR
  END WHILE
  RETURN

Swap:
  PUSH R1, R2, R7
  ; swap logic here
  POP R7, R2, R1
  RET

DisplayNumbers:
  FOR i = 0 TO 7
    LDR R0, R1, #i
    JSR IntToASCII      ; convert
    TRAP x21            ; OUT
    TRAP x21            ; OUT space
  RETURN
```