



East West University

Project Report

Semester: Spring-2025

Course Title: Computer Architecture

Course Code: CSE360

Sec: 01

Project Name: Stack Machine ISA

Group No: 07

Group Members:

Student Name	Student Id
Md.Rabbi	2022-2-60-067
Saheba Akter	2022-2-60-023
Mohua Akter	2022-3-60-112
Shahriar Khan	2022-3-60-016

Submitted to

Dr. Md. Nawab Yousuf Ali

Professor

Department of Computer Science & Engineering

East West University

Date of Submission: 21.05.2025

Title

Stack Machine ISA.

Objective

It is necessary to design and create a working stack machine simulator written in the C programming language as the project's key goal. With this simulator, the ISA will function by processing data saved on a stack, without requiring separate CPU registers. The project has these objectives:

1. Use a linked list to represent a stack data structure (LIFO) when storing tokens (operands or operators) as strings.
2. Support key stack actions: using PUSH to add a value at the top and POP to remove value from the top.
3. Create algorithms that allow you to convert expression types.
4. Turning symbols with operators from infix to postfix.
5. Stepwise conversion from Postfix to Infix form.
6. Develop an algorithm that looks at postfix expressions using only numeric operands and common arithmetic operators.
7. Be able to carry out $+$, $-$, \times and \div on the top two entries in the stack. The methods must process both number values and formula expression if input operands are not numbers.
8. Design a user interface that allows console interaction, showing the stack information, menus and different messages or traces using ncurses.

Formal Design, Mathematics, Formulas,

Derivations

In this section, we discuss the main theories behind the stack machine such as how it works, what data is used and the rules for converting and evaluating expressions.

Stack Machine System:

In a stack machine, the processor or model relies on a stack to hold all temporary data and control information. Stack machines differ from register-based types in that they mainly handle data items kept on the top of the stack. Among the important features are:

1. Implied use of the top elements in the stack means that most instructions don't need the address of where the operands are stored in the instruction. A good example is an ADD instruction which already understands to grab the top two values, add them together and store the answer.
2. The LIFO principle means that during memory access, data on top gets worked on before the data that was put in earlier. Whatever was placed last onto the stack will be what you pop out first.

Stack Data Structure

The stack in this project is implemented as a singly linked list of nodes.

Node: Each node stores a token (a string up to 31 characters representing an operand or operator) and a pointer next to the subsequent node in the stack.

```
typedef struct Node {  
    char token[32];  
    struct Node* next;  
} Node;
```

Stack: The stack structure itself contains a pointer top to the topmost node and an integer size representing the current number of elements in the stack.

```
typedef struct Stack {  
    Node* top;  
    int size;  
}Stack;
```

Core Operations:

1. `push(Stack* s, const char* token)`: Adds a new node containing the token to the top of the stack. If memory allocation fails, the program exits .
2. `pop(Stack* s, int* error)`: Removes the top node from the stack and returns its token. Sets an error flag if the stack is empty. The returned token is a dynamically allocated string that must be freed by the caller . .
3. `peek(Stack* s, int* error)`: Returns the token from the top node without removing it. Sets an error flag if the stack is empty . .
4. `is_empty(Stack* s)`: Checks if the stack is empty .

Expression Notations:

1. **Infix Notation**, uses parentheses around multiplication and addition between the numbers and includes them in between $(A+B \setminus C)$. Sometimes, you need parentheses to make sure operations are done in the order you intended.
2. **In Postfix Notation**, the operators are positioned following the operands (you write $AB \setminus +$). Because postfix notation doesn't require parentheses or special operator precedence, it works well on a stack.

Shunting-Yard Algorithm (Modified for Infix to Postfix Conversion):

This implemented `infix_to_postfix_stepwise` function is based on the Shunting-Yard algorithm:

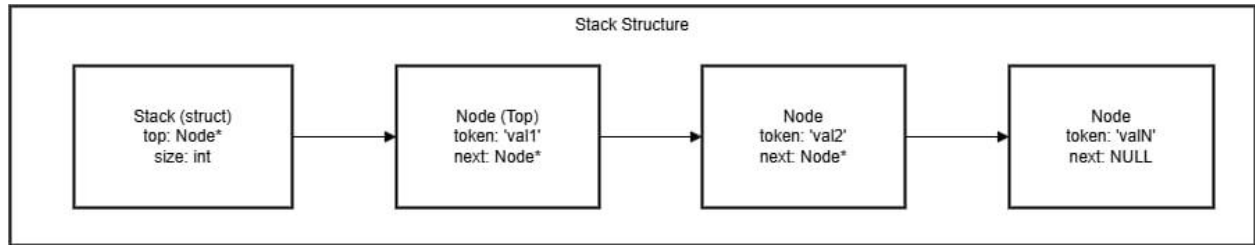
1. Make sure the postfix string is empty and the operator stack is empty as well.
2. Look at the expression from the left, reviewing each symbol as you go. You can have a token that is a number, a variable, an operator or a parenthesis.
3. When the token is one of the alpha or digit types, just add it to the postfix string, making sure to place a space after it.
4. Operator Stack.
5. Right Parenthesis `)`: Pop the operators off the stack and move them to the postfix string till a left parenthesis is spotted as the top most element. Pop and pitch the plastic case and insert the new one. If you get no answer, it means you have used unmatched parentheses.
6. If the line contains an operator keyword (`op`):
7. If the top of the operator stack is not empty and (the operator on top of it (`top_op`) has either greater or equal precedence than `op` AND `op` is not `^`), then the operator stack is never empty.
8. Pick a `top_op` from the stack and add it to the end of the postfix string.
9. Place `op` onto the stack.

Design

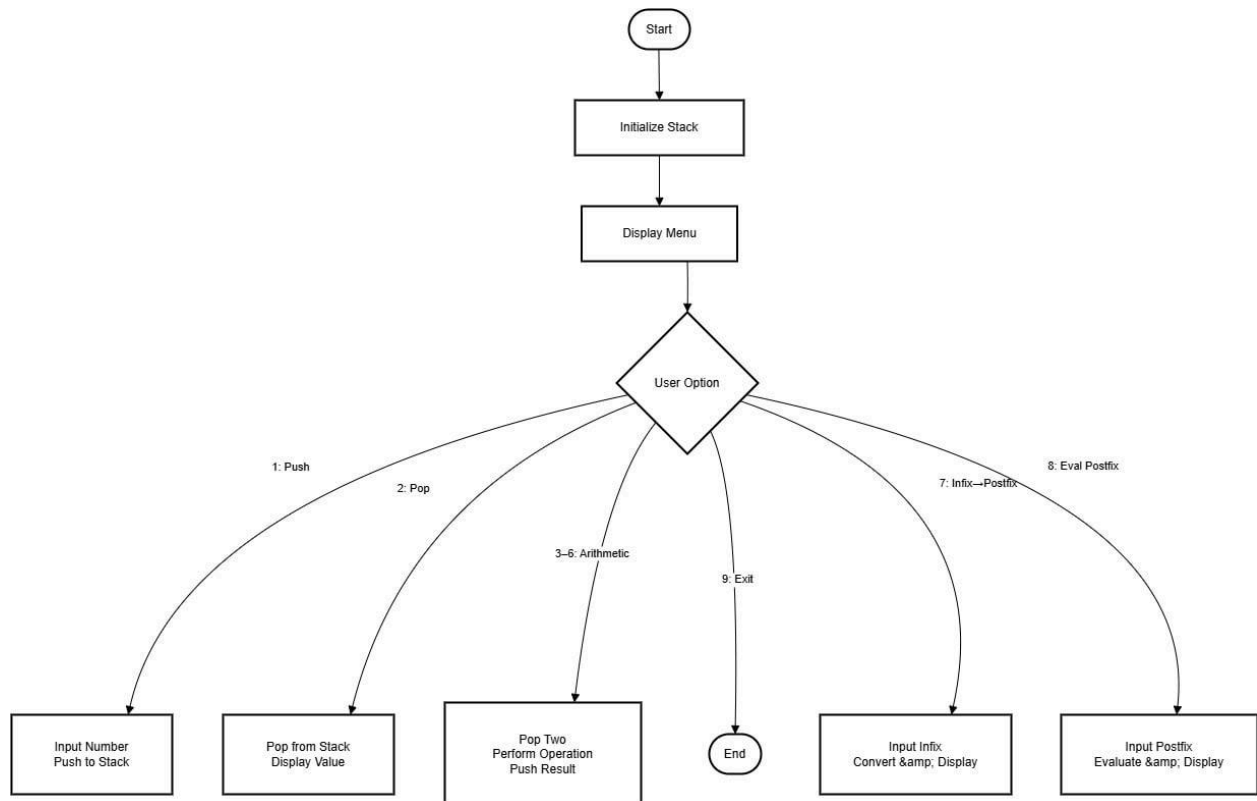
The design of the stack machine, including its data structures, flowchart , and an overview of its components.

Data Structures:

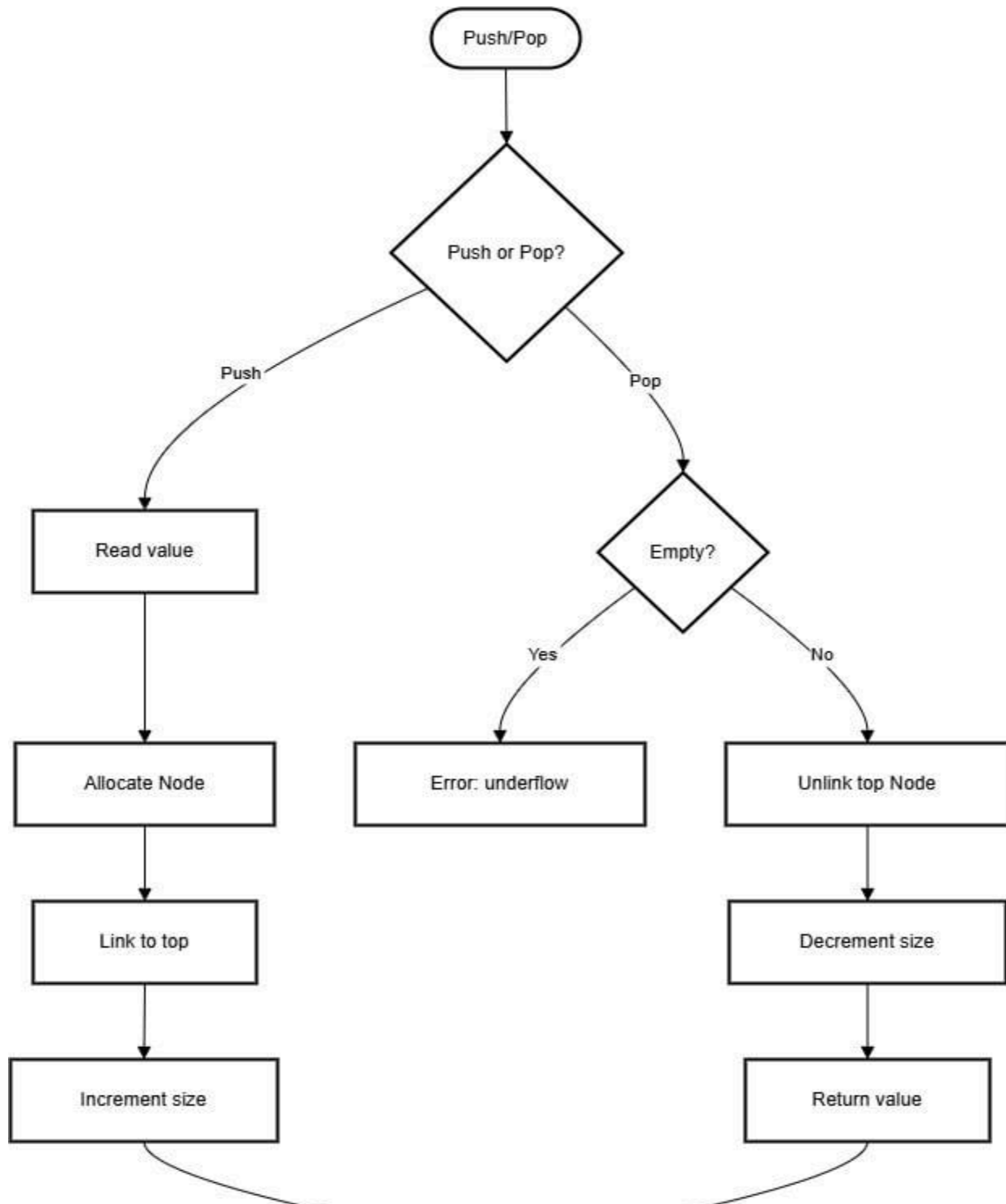
Stack implementation (Node, Stack): A Linked-list is used for stack.



FlowChart:



Push/Pop Machine Flow Chart:



Implementation

The implementation of the Stack Machine ISA was carried out in C, conforming to a stack-oriented design without the use of a general-purpose register. All computations are performed using a Last-In-First-Out(LIFO) stack. The key components include opcode interpretation, stack operations, memory handling, and control flow management.

Functional Modules:

- ❖ Main control loop(main):
 - It initializes the ncurses interface and stack.
 - It displays a menu to the user in a loop.
- ❖ Stack Operations:
 - PUSH: Adds a token to the top of the Stack.
 - POP: Removes the top token; returns error if the stack is empty.
 - PEEP:Retrieves the top token without removing it.
 - Is_empty: Checks whether the stack is empty.
- ❖ Expression Handling :
 - Infix to Postfix Conversion: Implements the Shunting Yard Algorithm.
 - Postfix to Infix Conversion: Uses a stack to reconstruct infix expressions from postfix tokens. Handles operator precedence and parenthesis explicitly.
- ❖ User Interface: This application features a real-time, interactive menu-driven interface using the ncurses library. It includes:
 - A menu window for selecting operations.
 - A stack display window to visualize current contents.
 - A message window for step-by-step operation tracing and error messages.

System Requirements:

- Compiler: GCC

- Language: C(ANSI C standard)
- Operating System: Linux
- Interface Library: **ncurses.h** - **used** for creating a text-based graphical user interface.

Debugging-Test-Run

This section describes the methodical procedure used to evaluate the implemented stack machine's functioning and guarantee its accuracy, resilience, and memory safety. Modular testing, interactive validation, and controlled test cases were used to thoroughly debug the project, guaranteeing dependable performance in both expected and edge scenarios.

Testing Strategy:

1. Unit Testing:

- Push, pop, and peek actions were tested using both legitimate and illegitimate inputs.
- Both valid algebraic expressions and incorrect formats were used as inputs for the verification of the expression modules (infix \leftrightarrow postfix, postfix evaluation).

2. Tests of Integration:

- The ncurses interface was utilized to test whole interaction flows.
- Analyses, conversions, and combined mathematical operations were performed in a methodical manner.
- Errors such as incorrect tokens, empty stack pop, and division by zero were purposefully triggered and handled compassionately.

3. Memory Management:

- Tools like Valgrind were used to monitor all dynamic allocations in order to make sure there were no incorrect accesses or memory leaks.

Validation :

- Every anticipated outcome was in line with the output displayed in the interface.
- No undefined behavior or software crashes were noticed.
- Errors in user input were regularly identified and reported.
- All tests showed clean, leak-free memory utilization.

Result Analysis

There are different types of operation in Stack Machine ISA. In our project we implemented stack operation, arithmetic operation , infix to postfix conversion, postfix to infix conversion.

1.Algorithmic Complexity:

Operation	AverageTime Complexity	Worst Time Complexity	Space Complexity
Push/Pop (Stack Operation)	$O(1)$	$O(1)$	$O(n)$
atof() conversion Arithmetic Operation	$O(k)$	$O(n^2)$	$O(n)$
Infix to Postfix Conversion	$O(n)$	$O(n)$	$O(n)$
Postfix to Infix	$O(n)$	$O(n)$	$O(n)$

Time Complexity:

k represents the string length.

n represents the length of expression

Space Complexity:

n represents the number of elements stored on the stack.

2. Robustness of Approach:

In our project, we can take string and numeric value as input and stack can store numeric and symbolic value and handle it. This flexibility allows us to work both values simultaneously. To prevent runtime errors, we avoid division by zero operation. We check stack underflow to prevent invalid operations with a friendly message and input validation ensures only alphanumeric token or valid characters are processed.

3. UI Features:

We can visualize the stack with highlighting the top element and we observed the result is green colour, error message is red colour, instruction is cyan colour.

Future Improvements

A number of potential enhancements might be taken into consideration to increase the present stack machine's usefulness and adaptability. More complicated program logic might be made possible by expanding the instruction set to include logical operations, conditional branching, and function call simulation. By enabling users to save and load phrases or instruction sequences, file input/output capabilities might improve usability and practicality. A graphical user interface like GUI version that makes use of frameworks like Qt or SDL may also improve the application's accessibility and usability. Last but not least, adding support for a basic scripting language that resembles assembly will enable users to create and decipher stack machine instructions in a more organized and legible manner.

The present version provides a strong basis for future research into stack machines and expression evaluation systems, despite time and scope constraints.

Conclusion

This project effectively illustrates how to use C to design and construct a stack-based machine architecture. All calculations and operations were carried out without the usage of registers by using a stack-oriented method, closely following the stack machine concepts. By offering a real-time, interactive terminal interface that graphically depicts stack operations and expression conversions, the ncurses library improved the user experience. Accurate step-by-step tracing was used to provide important features including postfix evaluation, infix-to-postfix conversion, and handling of symbolic expressions. The system's accuracy, stability, and robustness were validated through extensive testing, which made it appropriate for instructional demonstrations and laying the groundwork for comprehending stack-based computation models.