Department of CSE

CSE360 **Computer Architecture**	
Project No: 58	
Project Name: Construct an Interpreter for Single Accumulator CPU Organization.	
Submitted Date: 26-05-2024	
Semester & Year: Spring 2024	**Group No:** 06
Under the Guidance of: Dr. Md. Nawab Yousuf Ali Professor Department of Computer Science and Engineering East West University, Bangladesh	**Submitted by:** Suraiya Nusrat Tanha 2021-2-60-030 Fardin Islam 2021-2-60-041 Jubayer Alam Likhon 2021-2-60-071 Umme Habiba Fariha 2021-2-60-079

1. Title

Construct an interpreter written in C language to interpret an assembly language based on the following basic instructions for a machine having only one register, which is an accumulator, and all the operands are in memory:

Opcode	Operand	Comment
ADD	X	Add memory location X into accumulator
SUB	X	Subtract X from accumulator
MUL	X	Multiply X with accumulator
DIV	X	Divide accumulator by X
AND	X	And X with accumulator
NOT	X	Complement accumulator
OR	X	Or X with accumulator
LD	X	Load memory location X at accumulator
ST	X	Store accumulator at memory location X

2. Objective

The objective of this project is to design and implement an interpreter in C language that can interpret an assembly language for a hypothetical machine. This machine has only one register, which is an accumulator, and all the operands are in memory. The assembly language includes basic instructions such as ADD, SUB, MUL, DIV, AND, NOT, OR, LD, and ST.
The goal is to enhance understanding of how low-level programming and machine languages work, and how high-level languages can be used to interpret them. This project will also help in understanding the concepts of registers, memory operations, and basic arithmetic and logical operations in the context of assembly languages.

3. Theory

We know the interpreter and assembly language are based on the principles of computer architecture and low-level programming. Such as:

- **Machine Language and Assembly Language:** Machine language is a low-level programming language that is specific to a particular computer architecture. It consists of binary or hexadecimal instructions that a computer can execute directly. On the other hand, assembly language is a more human-readable version of machine language. Each assembly instruction corresponds to a specific machine language instruction.
- **Accumulator:** The accumulator is a special register in the CPU where intermediate arithmetic and logic results are stored. In our hypothetical machine, the accumulator is the only register, and all operations are performed on it.
- **Memory:** Memory is where data is stored. In our machine, all operands are in memory, and each memory location is identified by a unique address.
- **Instructions:** Instructions tell the machine what operation to perform. Each instruction consists of an opcode, which specifies the operation, and an operand, which specifies the data on which the operation is performed.

Here are the mathematical formulas and derivations for the operations:

- **ADD X:** This operation adds the value at memory location X to the accumulator. Mathematically, if acc is the accumulator and mem[X] is the memory location X, the operation can be represented as:

$$acc = acc + \text{mem[x]}$$

- **SUB X:** This operation subtracts the value at memory location X from the accumulator. Mathematically, it can be represented as:

$$acc = acc - \text{mem[x]}$$

- **MUL X:** This operation multiplies the accumulator by the value at memory location X. Mathematically, it can be represented as:

$$acc = acc * \text{mem[x]}$$

- **DIV X:** This operation divides the accumulator by the value at memory location X. Mathematically, it can be represented as:

$$acc = acc \div \text{mem[x]}$$

- **AND X, OR X:** These operations perform bitwise AND and OR operations on the accumulator and the value at memory location X. Mathematically, they can be represented as:

$$acc = acc \wedge \text{mem[x]}$$

and

$$acc = acc \mid \text{mem[x]}$$

- **NOT X:** This operation performs bitwise NOT operation on the accumulator. Mathematically, it can be represented as:

$$acc = \neg acc$$

- **LD X, ST X:** These operations load the value from memory location X into the accumulator and store the value from the accumulator into memory location X. Mathematically, they can be represented as:

$$acc = \text{mem[x]}$$

and

$$\text{mem[x]} = acc$$

4. Design

The main features of single accumulator-based CPU organizations are:

1. The First ALU (Arithmetic-logic Unit) operand is always stored in the accumulators and the second operand is present in the memory.
2. Accumulator is the default address. After manipulating the data, the results are stored in the accumulators.
3. Single-address instruction is used, in this Organization.

In this type of CPU association, the accumulator register is used implicitly for recycling all instructions of a program and storing the results into the accumulator. The instruction format that's used by this CPU association is ‘single address field’. Due to this, the CPU is known as the “Single Address Machine.”

The format of instruction is: Opcode + Address
What type of operation should be performed was shown by Opcode.
There are two types of operations are performed in single accumulators-based CPU organization:

1. Data Transfer Operation
In this type of operation, the data is transferred from a source to a destination.
For example: LOAD X (LD X), STORE Y (ST Y)
Here LOAD is memory read operation that is data is transferred from memory to accumulator and STORE is memory write operation that is data is transferred from accumulator to memory.

2. ALU (Arithmetic-Logic Unit) Operation
In this type of operation, arithmetic operations are performed on the data.
For example: MUL X
Here X is the address of the operand. The MUL instruction in this example performs the operation,

$$AC = AC * MEM[X]$$

AC is the Accumulator and MEM[X] is the memory word located at location X. This type of CPU organization is first used in PDP-8 processor and is used for process control and laboratory applications. It has been totally replaced by the introduction of the new general register-based CPU.

Advantages

- Simplicity: Easy to understand due to one register (the accumulator).
- Efficiency: Faster execution by using the accumulator directly.
- Uniformity: The uniform approach to processing instructions through one accumulator can simplify programming and machine language structure.

Disadvantages

- Limited Register Set: Only one register for computations.
- Memory Access: Slower due to frequent memory access.
- Limited Instruction Set: Fewer instructions available.
- No Error Handling: Lacks built-in error handling.

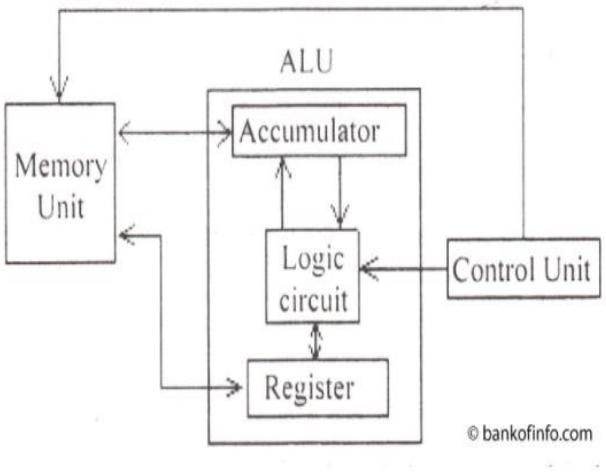


Figure 01: Single Accumulator ALU

5. Implementation

As instructed, we used C language to design an interpreter and the implementation of the source code is described below: -
Two important modules of the code are

i) getValues();

ii) Accumulator();

getValues() takes input from the user in the form of assembly code, separates and stores the tokens in an array.
Accumulator() takes the assembly code commands (tokens) stored in the array and performs the instructed operations and shows the output.

getValues_Function

```
void getValues()
{
   // Taking input from user
   printf("\n\nEnter a string (e.g., ADD 500): ");
   fgets(input, sizeof(input), stdin);

   for (int i = 0; input[i]; i++)
   {
      input[i] = toupper(input[i]);
   }

   sscanf(input, "%s %d", opcode, &operand);
}
```

Accumulator_Function

```
void Accumulator()
{

   if (!strcmp(opcode, "ADD"))
   {
      acc1 = acc1 + operand;
      printf("\nAccumulator = %d", acc1);
   }
   else if (!strcmp(opcode, "SUB"))
   {
      acc1 = acc1 - operand;
      printf("\nAccumulator = %d", acc1);
   }
   else if (!strcmp(opcode, "MUL"))
   {

      acc1 = acc1 * operand;

      printf("\nAccumulator = %d", acc1);
   }
   else if (!strcmp(opcode, "DIV"))
```

```
{

   acc1 = acc1 / operand;

   if (operand == 0)
   {
      printf("Math error! Cannot divide by 0!\n");
   }
   else
   {
      printf("\nAccumulator = %d", acc1);
   }
}
else if (!strcmp(opcode, "AND"))
{
   acc1 = acc1 & operand;
   printf("\nAccumulator = %d", acc1);
}
else if (!strcmp(opcode, "NOT"))
{
   acc1 = ~acc1;
   printf("\nAccumulator = %d", acc1);
}
else if (!strcmp(opcode, "OR"))
{
   acc1 = acc1 | operand;
   printf("\nAccumulator = %d", acc1);
}
else if (!strcmp(opcode, "LD"))
{
   acc1 = operand;
   printf("\nAccumulator = %d", acc1);
}
else if (!strcmp(opcode, "ST"))
{
   printf("\nAccumulator1 = %d", acc1);
   printf("\nstored in memory location: %p", &acc1);
}
else if (!strcmp(opcode, "END"))
{
   printf("\nThe session has ended!\nThank you :)\n");
   printf("This software is developed by\n\n");
   displayDev("Name", "Student ID");
   printf("-----------------------------------------\n");
   displayDev("Fardin Islam", "2021-2-60-041");
   displayDev("Jubayer Alam Likhon", "2021-2-60-071");
```

```
        displayDev("Umme Habiba Fariha", "2021-2-60-079");
        displayDev("Suraiya Nusrat Tanha", "2021-2-60-030");
        printf("\n\n\n");
        exit(0);
    }
    printf("\n");
}
```

6. Debugging-Test-run

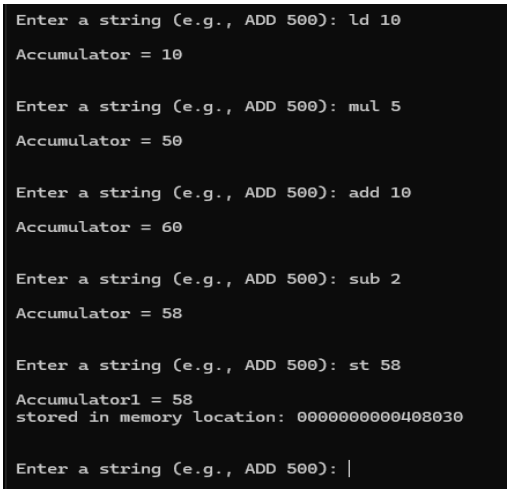
Debugging and test run yielded satisfactory results as the code was able to correctly interpret assembly language inputs by the user. Screen shots and truth-test of results are given below:

LD, ADD, SUB, MUL, ST

Input:

```
LD 10
MUL 5
ADD 10
SUB 2
ST 58
```

Output:

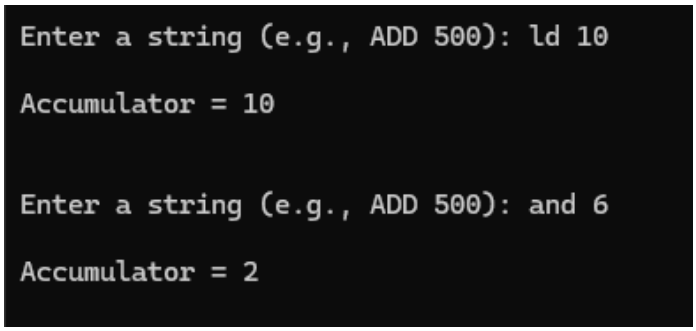


Bitwise AND

Input:

```
LD 10
AND 6
```

Output:



Truth Test:

```
10 = 00001010
6  = 00000110
AND
-------------------
2  = 00000010
```

Bitwise NOT

Input:
NOT 5

Output:

```
Enter a string (e.g., ADD 500): not 5

Accumulator = -6
```

Truth Test:
5 = 00000101
NOT

-6 = **1**1111010
(since the leftmost bit is the sign bit)

7. Results analysis:

Error Handling: The division by zero is handled, but the message is printed after performing the division, which is incorrect. It should be checked before performing the division to avoid runtime errors.
Invalid Opcode: If an invalid opcode is entered, no action is taken, and no error message is printed, which may confuse users.
Empty Input: The code does not handle empty input or cases where the input does not match the expected format (e.g., missing operand).

The code is relatively simple and straightforward. It uses clear function names and modularizes tasks into functions like clearValue, getValues, and Accumulator. Some improvements can be made, such as checking for errors before performing operations and handling more input edge cases.

8. Conclusion and Future Improvements

In our project, we diligently worked to rectify all bugs during the code development process. The design we implemented successfully delivers the desired output. To illustrate, it accurately performs operations such as Addition, Multiplication, Subtraction and Division, providing us with the precise results we anticipated. We managed to make the code fully operational without any glitches, signifying that our project was complete and ready for deployment. Looking ahead, we plan to enhance our design by incorporating more accumulators, which will also reduce the processing time. The memory location of these accumulators can be stored for future retrieval. We strongly advocate for comprehensive bug testing and timely resolution of any identified issues.

9. Bibliography:

- Computer Organization, CPU architecutre, ALU, Accumulator, Program Counter PC, Registers, IR, Decoder, Timing and Control unit, Flags, bus. (fullchipdesign.com)
- Computer Organization | Instruction Formats (Zero, One, Two and Three Address Instruction) - GeeksforGeeks
- Introduction of General Register based CPU Organization - GeeksforGeeks
- Computer Organization and Architecture: Designing for Performance (8th Edition) -William Stallings - Prentice-Hall, Inc. Division of Simon and Schuster One Lake StreetUpper Saddle River, NJ, United States