



East West University

Department of Computer Science and Engineering

Project Report

Course Title : Computer Architecture
Course Code : CSE360
Section : 2
Semester : Summer 2020
Project No : 27
Project Title : Construct an interpreter written in C language to interpret an assembly language based on a given set of basic instructions for a machine having only one register, which is accumulator, and all the operands are in memory.

Submitted To:

Dr. Ahmed Wasif Reza
Associate Professor
Department of Computer Science & Engineering
East West University

Submitted By:

Mayisha Tasneem	(2017-2-60-010)
Sabbir Ahmed	(2017-2-60-063)
Sayed Atique Newaz	(2017-2-60-067)

Date of Submission: 18th September, 2020

Title:

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

Objective:

We were given the following set of assembly code instructions:

	Opcode, operand	comment	
	ADD X	Add memory location x into acc.	
	SUB X	Subtract X from Acc.	
	MUL X	Multiply X with Acc.	
	DIV X	Divide acc. by X.	
	AND X	And X with acc.	
	NOT X	Complement acc.	
	OR X	Or X with acc.	
	LD X	Load memory location X at acc.	
	ST X	Store acc. at memory location X	

Our task was to design a assembly language interpreter with the above mentioned instructions for a machine having only one register using C language.

Design:

The main points about Single Accumulator based CPU Organization are:

1. In this CPU Organization, the first ALU operand is always stored into the Accumulator and the second operand is present in the Memory.
2. Accumulator is the default address thus after data manipulation the results are stored into the accumulator.
3. One address instruction is used in this type of organization.

In this type of CPU organization, the accumulator register is used implicitly for processing all instructions of a program and store the results into the accumulator. The instruction format that is used by this CPU Organization is '**one address field**'. Due to this the CPU is known as **One Address Machine**.

The format of instruction is: Opcode + Address

Opcode indicates the type of operation to be performed.

Mainly two types of operation are performed in single accumulator based CPU organization:

1. **Data transfer operation –**

In this type of operation, the data is transferred from a source to a destination.

For ex: LOAD X, STORE Y

Here LOAD is memory read operation that is data is transfer from memory to accumulator and STORE is memory write operation that is data is transfer from accumulator to memory.

2. **ALU operation –**

In this type of operation, arithmetic operations are performed on the data.

For ex: MULT X

where X is the address of the operand. The MULT instruction in this example performs the operation,

$$AC \leftarrow AC * M[X]$$

AC is the Accumulator and M[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 –

- One of the operands is always held by the accumulator register. This results in short instructions and less memory space.
- Instruction cycle takes less time because it saves time in instruction fetching from memory.

Disadvantages –

- When complex expressions are computed, program size increases due to the usage of many short instructions to execute it. Thus memory size increases.
- As the number of instructions increases for a program, the execution time increases.

Implementation:

As per instructions, we used C programming language to implement an assembly language interpreter based on the given set of basic instructions assuming only one (accumulator) register.

Functional modules of source code:

Two most important modules in the code include `takeInput()`; and `parseCommand()`;

`takeInput()`;

The `takeInput()`; module takes input from the user in the form of assembly code, separates and stores the tokens in an array.

`parseCommand()`;

The `parseCommand()`; module takes the assembly code commands (tokens) stored in the array and performs the instructed operations and shows output.

```
void takeInput()
{
    fgets(input, sizeof(input), stdin);
    int len = strlen(input);
    input[len-1] = '\0';
    int words = 0, i;
```

```

int r = 0, k = 0;
for(i=0; input[i]!='\0'; i++)
{
    tokens[k][r] = '\0';
    if(input[i] == ' '){ k++; r = 0; }
    else
    {
        tokens[k][r++] = input[i];
        tokens[k][r] = '\0';
    }
}
}

```

```

void parseCommand()
{
    int x;
    if(!strcmp(tokens[0], "ADD"))
    {
        x = atoi(tokens[1]);
        acc = acc + x;

        printf("\naccumulator = %d\n", acc);
    }else if(!strcmp(tokens[0], "SUB"))
    {
        x = atoi(tokens[1]);
        acc = acc - x;

        printf("\naccumulator = %d\n", acc);
    }
    else if(!strcmp(tokens[0], "MUL"))
    {
        x = atoi(tokens[1]);
        acc = acc * x;
        printf("\naccumulator = %d\n", acc);
    }else if(!strcmp(tokens[0], "DIV"))
    {
        x = atoi(tokens[1]);
        acc = acc / x;
        if(x == 0)
        {
            printf("Math error! Cannot divide by 0!\n");
        }else
        {

```

```

        printf("\naccumulator = %d\n", acc);
    }
} else if (!strcmp(tokens[0], "AND"))
{
    x = atoi(tokens[1]);
    acc = acc & x;
    printf("\naccumulator = %d\n", acc);
} else if (!strcmp(tokens[0], "NOT"))
{
    acc = ~acc;
    printf("\naccumulator = %d\n", ~acc);
} else if (!strcmp(tokens[0], "OR"))
{
    x = atoi(tokens[1]);
    acc = acc | x;
    printf("\naccumulator = %d\n", acc);
} else if (!strcmp(tokens[0], "LD"))
{
    x = atoi(tokens[1]);
    acc = x;
    printf("\naccumulator = %d\n", acc);
} else if (!strcmp(tokens[0], "ST"))
{
    printf("\naccumulator = %d\n", acc);
    printf("\nstored in memory location: %p", &acc);
}
printf("\n\n");
}

```

Debugging-Test-run:

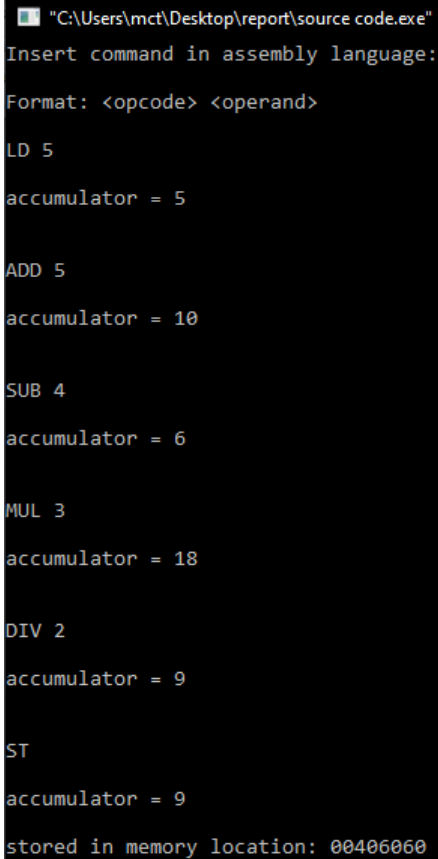
Debugging and test run yielded satisfactory results as the code was able to correctly interpret assembly language inputs by the user. Screen shots and truth-test of results are given below:

LD, ADD, SUB, MUL, DIV, ST

Input (assembly code):

```
LD 5
ADD 5
SUB 4
MUL 3
DIV 2
ST
```

Output:



```
"C:\Users\mct\Desktop\report\source code.exe"
Insert command in assembly language:
Format: <opcode> <operand>
LD 5
accumulator = 5

ADD 5
accumulator = 10

SUB 4
accumulator = 6

MUL 3
accumulator = 18

DIV 2
accumulator = 9

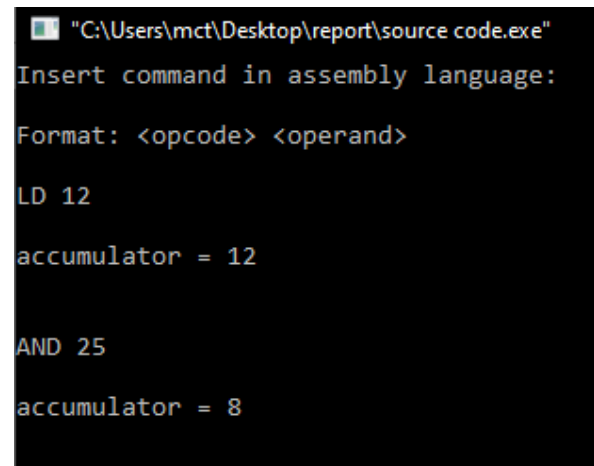
ST
accumulator = 9
stored in memory location: 00406060
```

Bitwise AND

Input (assembly code):

```
LD 12
AND 25
```

Output:



```
"C:\Users\mct\Desktop\report\source code.exe"
Insert command in assembly language:
Format: <opcode> <operand>
LD 12
accumulator = 12
AND 25
accumulator = 8
```

Truth-test:

```
12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

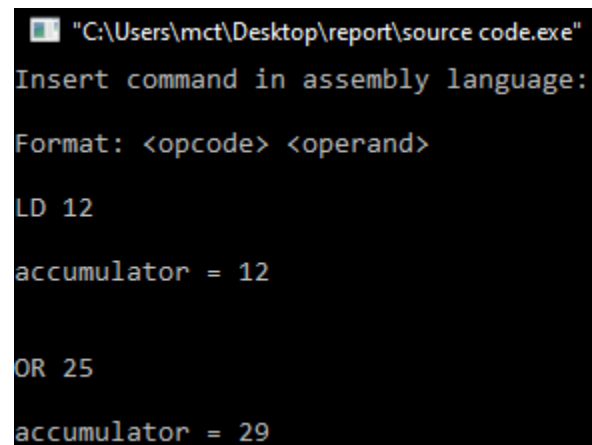
00001100
& 00011001
-----
00001000 = 8 (In decimal)
```


Bitwise OR

Input (assembly code):

```
LD 12
OR 25
```

Output:



"C:\Users\mct\Desktop\report\source code.exe"
Insert command in assembly language:
Format: <opcode> <operand>
LD 12
accumulator = 12
OR 25
accumulator = 29

Truth-test:

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
```

Bitwise OR Operation of 12 and 25

```
00001100
| 00011001
```

```
00011101 = 29 (In decimal)
```

Bitwise OR

Input (assembly code):

```
LD 35
NOT
```

Output:

```
"C:\Users\mct\Desktop\report\source code.exe"
Insert command in assembly language:
Format: <opcode> <operand>
LD 35
accumulator = 35
NOT
accumulator = -36
```

Truth-test:

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

11011100 = 220 (In decimal)

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

Success Rate:

Time Attempts: 33

Error Finds: 2

In this case we have: $2 \div 33 = 0.0606$

So, the success rate is $0.0606 \times 100 = 6.06\%$

Conclusion and Future Improvements:

In conclusion, we were able to accomplish our project's given objectives as our code was able to correctly parse and interpret the assembly language input given by the user. We have fixed the bugs encountered while developing this program and at its current state it serves its function without error. However, future improvements should include exhaustive bug-testing and bug-fixes.

Bibliography:

- Computer Organization and Architecture: Designing for Performance (8th Edition) - William Stallings - Prentice-Hall, Inc. Division of Simon and Schuster One Lake Street Upper Saddle River, NJ, United States