

**Project Report**

Project Title: Assembly Language Interpreter Written in C Language

Course Code: CSE360

Course Title: Computer Architecture

Section:

**Submitted By:**

**Submitted To:**

Dr. Ahmed Wasif Reza

Associate Professor

And Head of the department of CSE,

East West University

**Date of Submission:**

**Title**

An interpreter written in C language to interpret an assembly language based on the basic instructions for a machine having only one register.

**Objective**

The objective of our project is to simulate assembly language to provide a better understanding how a computer works. Assembly language is as close to the processor as a person can get as a programmer so assembly language is great for speed optimization. Our project is intended to:

* Create an interpreter for assembly language that can do simple tasks with one register.
* To give a better understanding of how CPU, register and memory works together to execute a task.

**Theory**

Assembly language is a low-level programming language for a computer or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

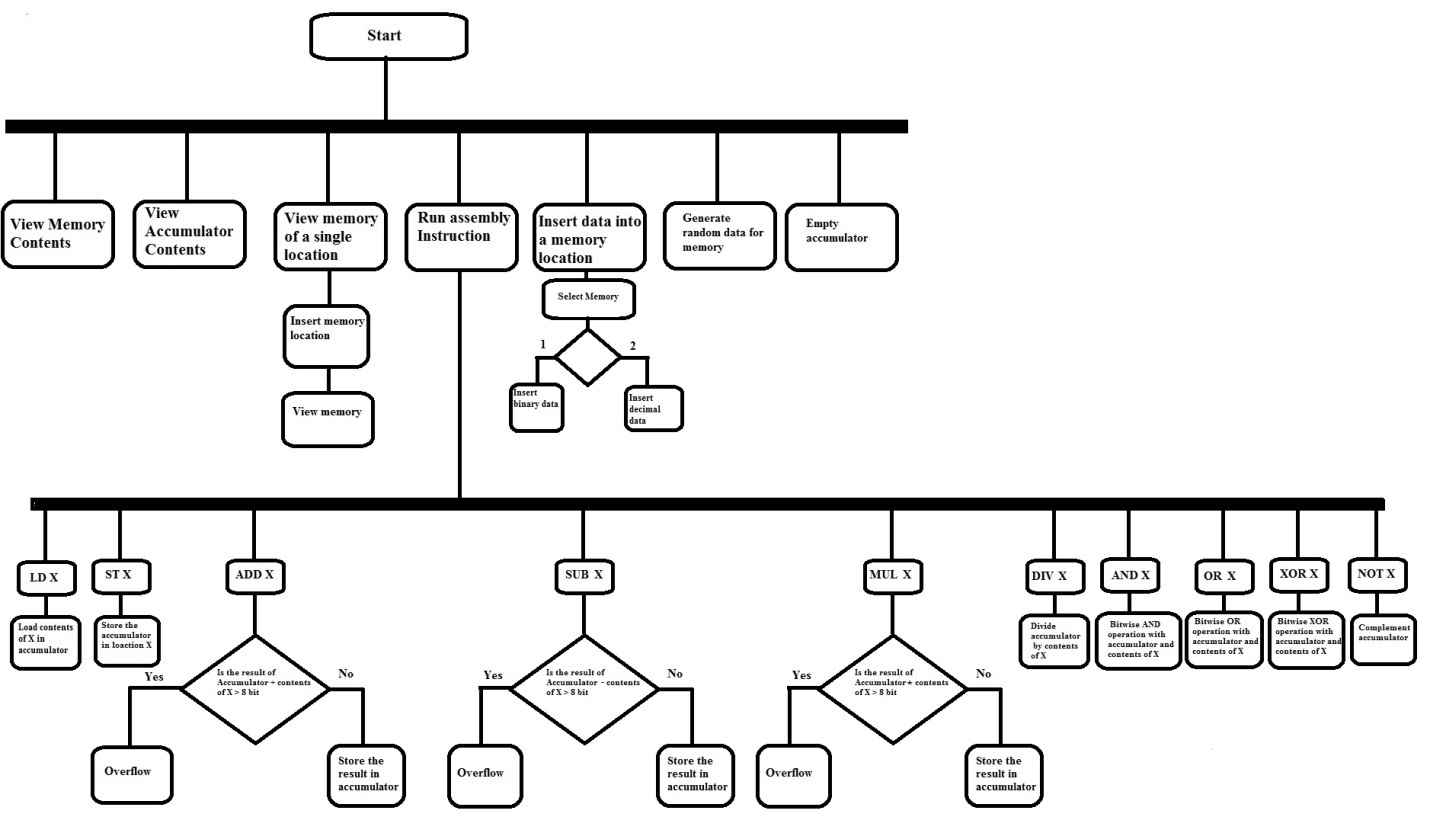
A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

An assembly language uses a register which is accumulator to hold values during calculations and a memory that holds the data. Assembly language performs following tasks on the data in the memory:

|  |  |  |
| --- | --- | --- |
| **Opcode** | **Operand** | **Function** |
| ADD | X | Add memory location X into accumulator. |
| SUB | X | Subtract X from accumulator. |
| MUL | X | Multiply X with accumulator. |
| DIV | X | Divide accumulator with memory location X. |
| AND | X | Apply bitwise AND operation with memory location X and accumulator. |
| NOT | X | Apply bitwise NOT on accumulator. |
| OR | X | Apply bitwise OR operation with memory location X and accumulator. |
| XOR | X | Apply bitwise XOR operation with memory location X and accumulator. |
| LD | X | Load memory location X into accumulator. |
| ST | X | Store current accumulator to memory location X. |

**Design**

Flowchart:



Algorithms:

View all memory contents

showAccumulator()

print("Accumulator")

print(toDecimal(Accumulator))

print(Accumlator)

showMemory()

integer i=0,j=0

while(i<MemorySize)

print(Memory[i])

print(toDecimal(Memory[i]))

i++

while(j<Length)

print(Memory[i][j])

j++

View accumulator contents

showAccumulator()

print("Accumulator")

print(toDecimal(Accumulator))

print(Accumlator)

View memory contents of a location

viewMemoryLocation()

print("Enter memory location")

input(operand)

showMemory(operand)

showMemoryFor(operand)

integer j=0

print(Memory[operand])

print(toDecimal(Memory[operand]))

while j<Length

print(Memory[operand][j])

j++

Run Assembly Instruction

runInstruction()

input(Opcode,Operand)

Instruction(Opcode,Operand)

Instruction(opcode,operand)

integer i=0

integer j=0

if(Opcode=="ADD")

temp = Accumulator+Memory[Operand]

Accumulator=temp

else if(Opcode=="SUB")

temp = Accumulator-Memory[Operand]

Accumulator=temp

else if(Opcode=="MUL")

temp = Accumulator\*Memory[Operand]

Accumulator=temp

else if(Opcode=="DIV")

temp = Accumulator/Memory[Operand]

Accumulator=temp

else if(Opcode=="AND")

i=0

while i<Length

if(Accumulator[i]==1 && Memory[operand]==1)

Accumulator[i]=1

else

Accumulator[i]=0

i++

else if(Opcode=="NOT")

i=0

while i<Length

if(Accumulator[i]==0)

Accumulator[i]=1

else

accumulator[i]=0

i++

else if(Opcode=="OR")

i=0

while i<Length

if(Accumulator[i]==0 && Memory[operand]==0)

Accumulator[i]=0

else

Accumulator[i]=1

i++

else if(Opcode=="XOR")

i=0

while i<Length

if(Accumulator[i]!=Memory[operand])

Accumulator[i]=1

else

Accumulator[i]=0

i++

else if(Opcode=="LD")

j=0

while j<Length

Accumulator[j]=Memory[Operand][j]

j++

else if(Opcode=="ST")

j=0

while j<Length

Memory[Operand][j]=Accumulator[j]

j++

Insert data to a memory location

inputMemory()

String B[Length]

integer l,D,c,i,j

input(l)

input(c)

if(c==1)

input(B)

if(b.length!=Length)

return

else

i=0

while(i<Length)

if(B[i]!=0&&B[i]!=1)

return

i++

j=0

while(j<Length)

Memory[l][j]=B[j]

ShowMemoryFor(l)

break

else if(c==2)

input(D)

if(D>Limit/2||D<Limit/2\*-1)

return

j=0

while(j<Length)

Memory[l][j]=B[j]

ShowMemoryFor(l)

j++

break

Generate new random data for memory

GenerateMemory()

integer i

srand(time(NULL))

i=0

while(i<MemorySize)

temp=toBinary((rand()%LIMIT)-LIMIT/2)

Memory[i]=temp

i++

Empty accumulator

EmptyAccumulator()

integer i

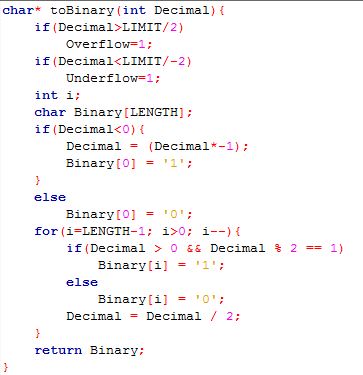
i=0

while(i<Length)

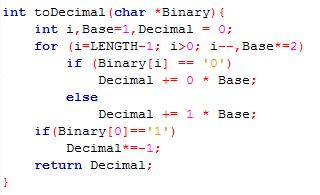
Accumulator[i]=0

i++

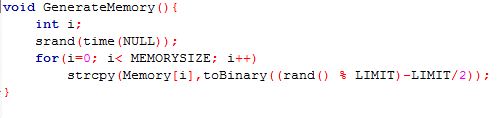
**Implementation**



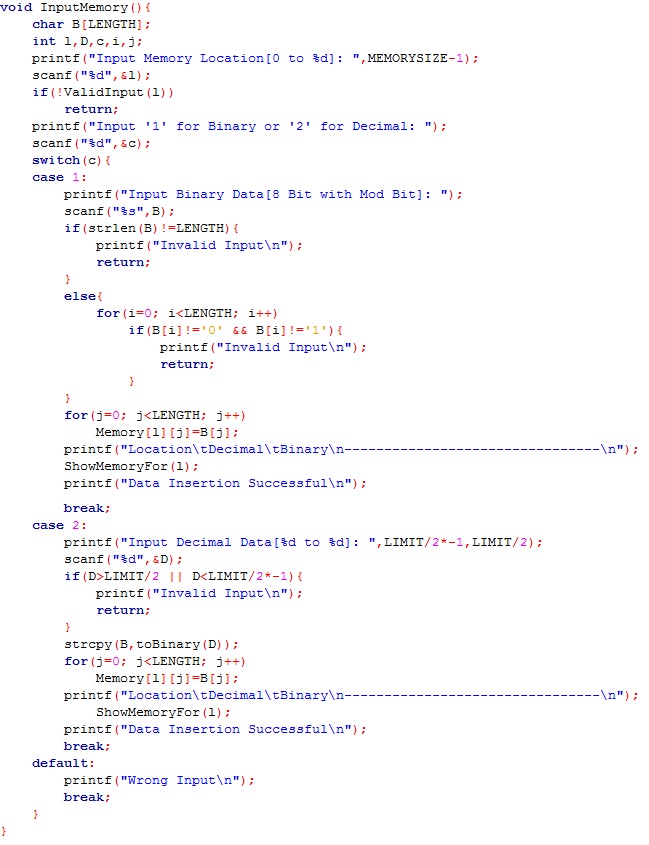
Above block of code convert a decimal number to binary. It is essential because we store binary values in memory.



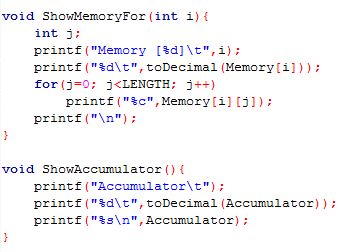
This function converts a binary number to decimal. It takes binary values stored in memory and covert into a decimal number. This function is important because it helps a user to understand easily what value is stored in the memory.



The above function generates a new set of memory. The memory is generated with random values. It written In a way that the values to be stored in the memory will not exceed the memory size.



This block of code do task of taking a value to store in the memory. The code provides the option to user to input both binary and decimal value. If a user input an illegal value such as a very large number or character then it will not take the input.



There are two functions in the given blocks of code above. The showMemory() function prints the memory block in the screen. The showAccumulator() function prints the register which is accumulator in the screen.



The above block of code is responsible for executing assembly language. It can do basic assembly language tasks such as ADD, SUB, MUL, DIV, AND, OR, XOR, LD, ST. The code takes an opcode to determine which operation to do and an operand with which value the operation should execute. If any operation creates an overflow then the program will print the error message.

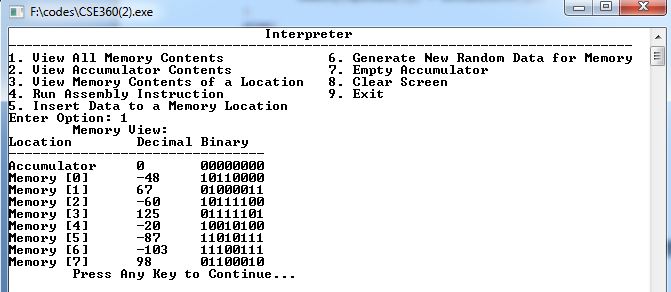
System Requirement:

OS: Windows, Linux, MacOS

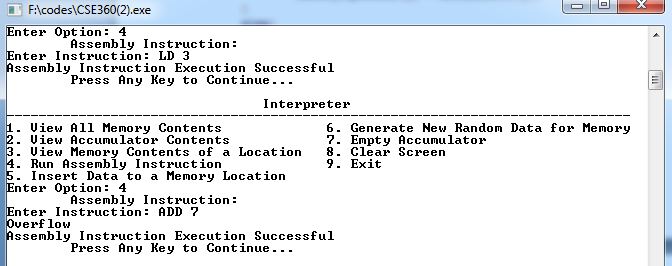
Compiler: Any C language compiler

**Debugging-Test Run**

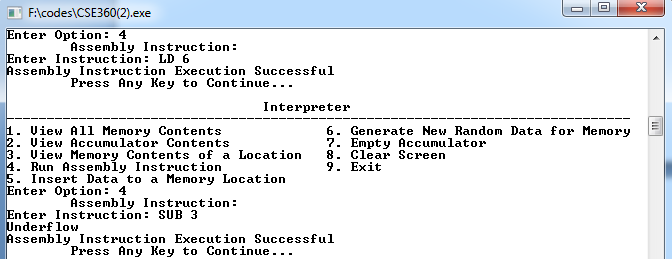
Our code will not allow any overflow of values in memory. So when any operation generate any overflow it will print error message.



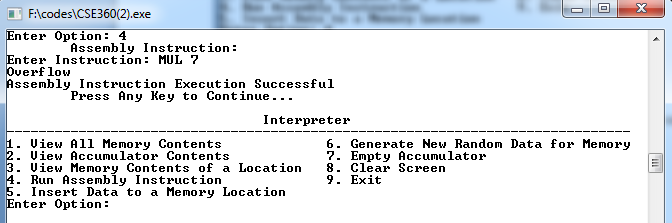
We can see the current memory.



Then we run assembly instruction to Load memory location 3 into the accumulator then run another assembly language to add memory location 7 to the accumulator. The result creates an overflow. So the console prints the message “overflow”.



We can see that the code generates an underflow message when the value is too small.

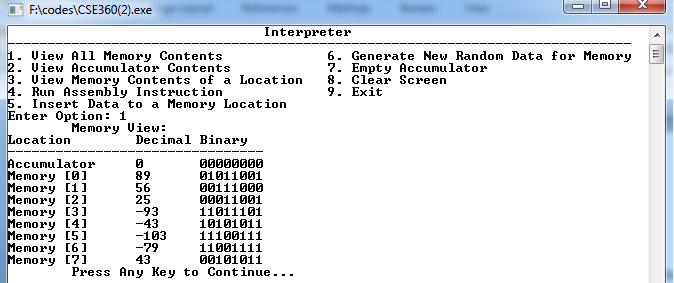


When the multiplication operation is applied if the result of multiplication between accumulator and a memory contents exceed expected length then the console will print overflow.

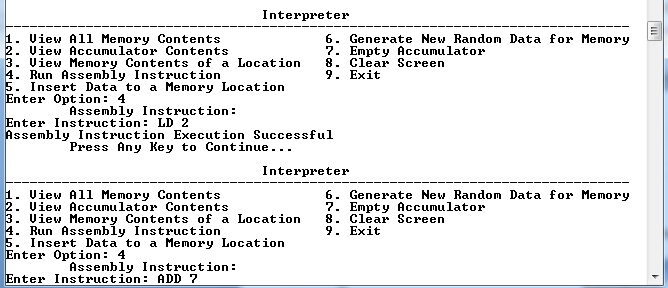
So we can see that above test runs prove our code can handle unexpected and invalid inputs and results.

**Result Analysis**

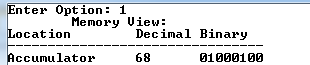
Now we run some test inputs to see if our application works.



We can see the contents of the memory.

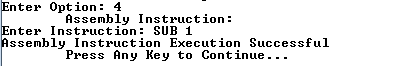


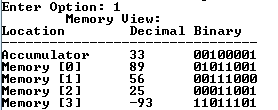
Then we loaded memory location 2 to accumulator and added location 7. After that the accumulator looks like this.



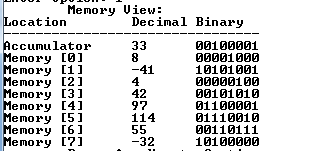
Then we applied subtraction with contents of location 0 and contents of location 1





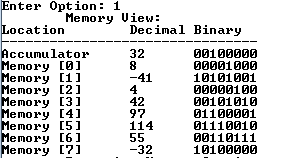


Now we apply multiplication operation. We generate new memory.

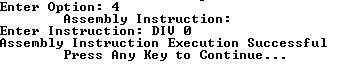


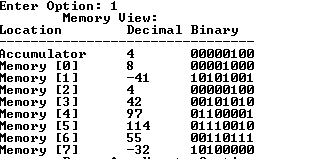




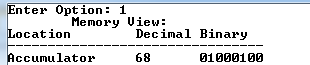


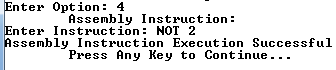
Now we apply division operation.





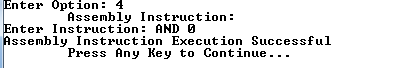
Then we applied not operation and the result given below.

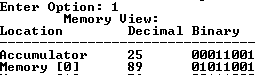




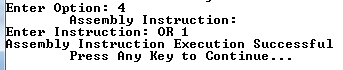


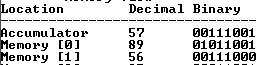
Then we applied AND operation with location 0.





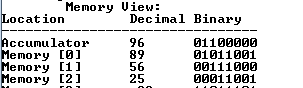
Then we applied OR operation with location 1. Now the result is:





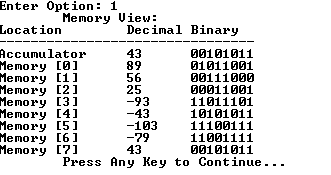
Now we apply XOR operation with location 0.





Now we store the value in location 7





**Future Work and Conclusion**

Currently our project only has one address instruction support. We can add some features in future to provide zero address instruction, two address instructions and three address instruction supports. We can add many other instructions to support conditions, loops, arrays etc.

Assembly language is very important to understand how CPU interacts with memory and registers. Our project will help people to learn basic assembly language instructions. We will improve our project in future so that people can get more benefits from our project.

**Bibliography**

1. [www.wikipedia.org](http://www.wikipedia.org)
2. [www.tutorialspoint.com](http://www.tutorialspoint.com)
3. [www.techopedia.com](http://www.techopedia.com)
4. Computer Organization and Architecture: Textbook by William Stallings