**A picture containing circuit, electronics

Description automatically generatedBASIC COMPUTER SIMULATOR DOCUMENTATION:**

**Muhammad Omer 2020335**

**Khizar Ali Shah 2020196**

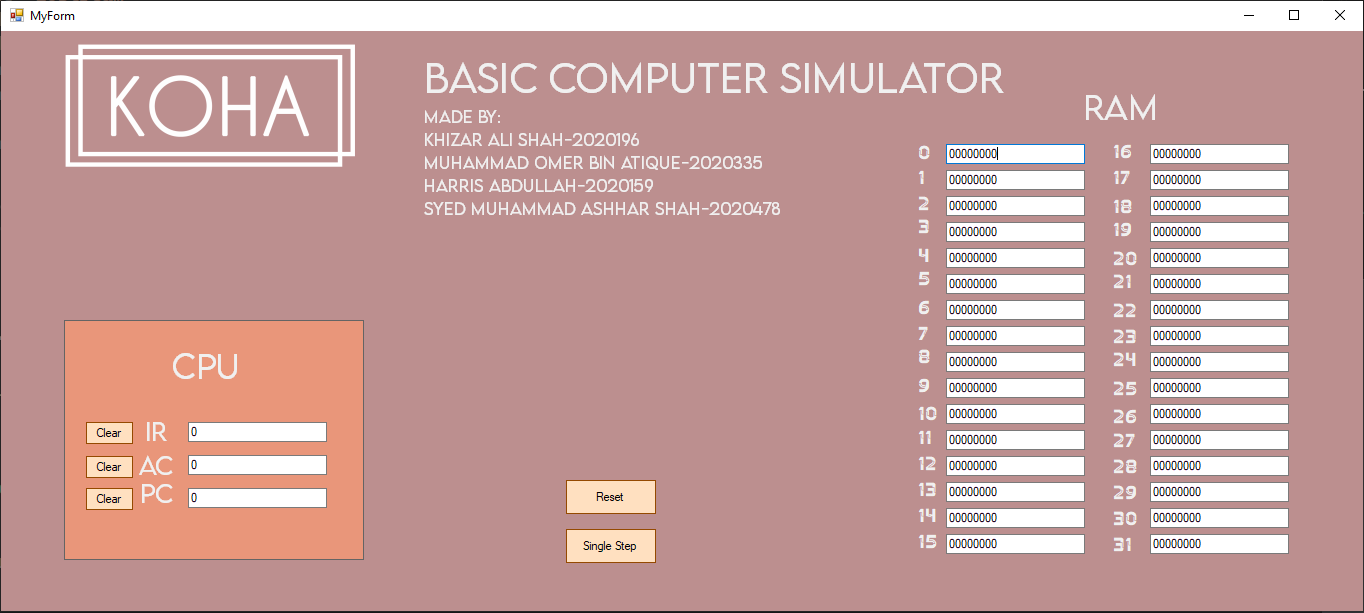
**Muhammad Ashhar 2020478**

**Harris Abdullah 2020159**

# Introduction:

This simulator is the basic representation of happens inside a basic computer. This simulator can accept Assembly language code and can execute the code line by line.

Essentially, the assembly language code is written inside the RAM area. The simulator starts executing the program form memory location 0. It iterates from one memory location to the other step by step executing each line.



# COMPONENTS:

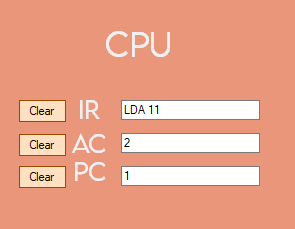
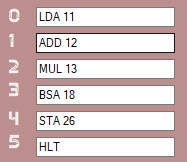
The simulator has the following components:

## CPU:

The CPU area showcases the internal workings of a CPU. While it does not show every internal component of the CPU, it does show the essential components including IR (Instruction Register), AC (Accumulator), PC (Program Counter).

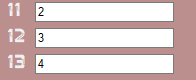
## IR (Instruction Register):

The instruction register is the part of the CPU which stores the Instruction being executed. For example, if the instruction being executed is LDA 11 then IR will show LDA 11. (See diagram)



## AC (Accumulator):

The accumulator is a register in CPU where all the operations are performed by default. The accumulator is essentially where the result is calculated. For example, in the example above operand stored at memory location 11, in this case 2, is loaded into the AC. (See diagram above)



## PC (Program Counter):

The program counter is the register which keeps track of the address of the next instruction to be executed. For example, in this case it is 1 as the instruction being executed is at memory location 0, so the address of the next instruction to be executed is 1.

## Clear:

The clear buttons in this simulator reset the values of IR, AC and PC to 0.

## RAM:

The RAM basically, represents the 32 contiguous memory locations which is the memory of the basic computer where the program in Assembly language is stored.

## Reset Button:

The reset button clears all the memory locations in RAM, IR, AC, and the PC to their default values which is ‘0’.

## Single Step:

This button basically executes one step on being pressed.

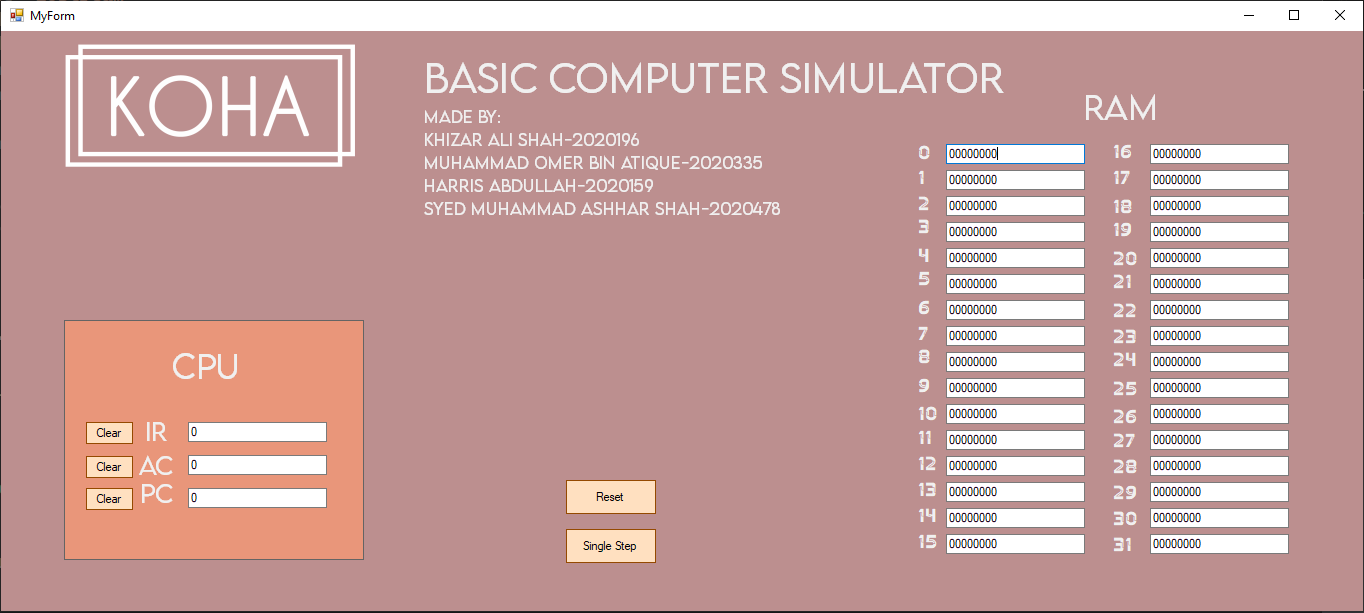
# INSTRUCTIONS:

The simulator accepts the following instructions:

|  |  |
| --- | --- |
| INSTRUCTIONS | DESCRIPTION |
| LDA | Loads into AC from memory |
| MUL | Multiplies memory word to AC |
| ADD | Add memory word to AC |
| STA | Store content of AC into memory |
| BUN | Branch Unconditionally |
| BSA | Branch and save return Address |
| ISZ | Increment and skip if zero |
| HLT | Halt Simulator/Computer |

# FETCH:

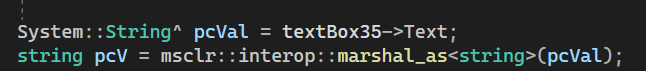
The simulator was implemented in the .NET C++ framework. So first a basic level front end was made as shown below:



Once this was accomplished, the backend was done in C++. Initially we decided to write all the code in the function made for the Single Step button. When ever the single step button is pressed the compiler executes the code in its function. We start off by getting the value in the PC register as it tells us the address of the instruction to be executed next. When using a windows form for the front end the values in the text boxes are stored as type “String ^”. To use this data in our code we have to convert it to “string”. For this we use the C++ library:

#include <msclr/marshal\_cppstd.h>

For String^ to string conversion we use the following code:



//pcVal is the variable that stores from the textbox of type String^

//pcV is of type string which stores value of pcVal after conversion //from

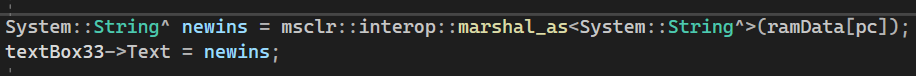
//String^ to string

Once we have the value of PC in string we can convert it to integer as well which makes it easier for us to increment and check.

Similarly, we also get the value of AC and store it in and integer variable.

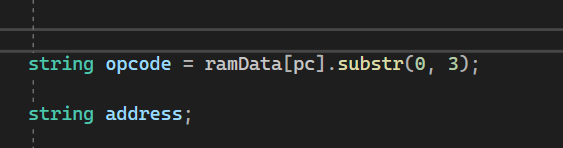
The next step involves an array of size 32 (0-31), ramData[]. This array will basically store the data in the RAM once the program starts or at PC = 0. This makes it easier for us to get data from the RAM and updating it in the future. Now as the PC is incremented we can check values in ramData[PC] to execute the program. Also the instruction at ramData[PC] is then put into the IR

For string to String^ conversion (For updating values in text box):



// The code above is used to convert type string variable into type // String^ variables.

Now once we write an assembly instruction, we have the basic format of the first 3 characters telling us the instruction, the next 1 character is the space and the next 2 are telling us the address. So, we use the string library and sub string function to get the first 3 letters and store them into a variable called opcode of type string. Similarly, we store the 2 address characters in string called address.



# EXECUTE:

Once we have instruction in opcode we can then match it with the instructions that we want to simulate.

## HLT:

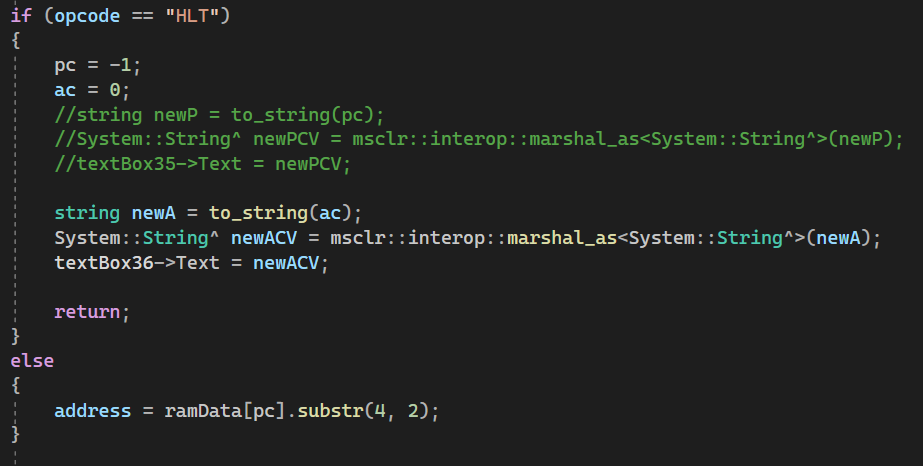
We first make the pc = -1 (as it is incremented later to zero) and we also reset the AC.

We update the AC within the if statement of HLT, yet PC is updated in the code later on.

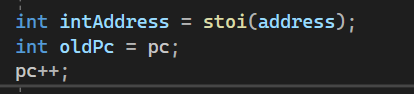
And then we just return

This will effectively stop the execution of the code.

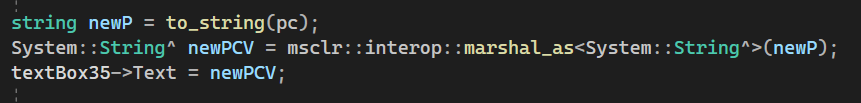
If the opcode is not “HLT” then we get the address as HLT is the only instruction which does not have an address.



Once we are done with HLT, we store the address (string) into and integer value we also store the value of pc into a new variable oldPc before incrementing the PC.



After incrementing the PC, we can then update the value onto front end.



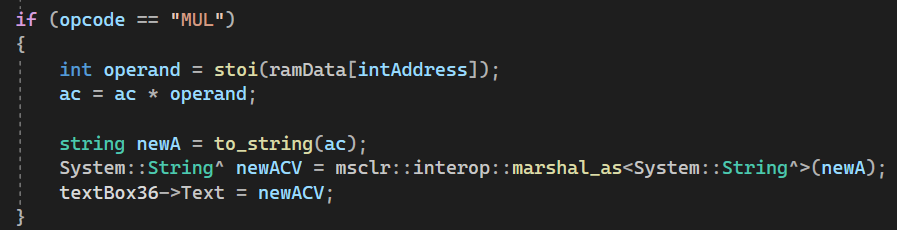
## MUL:

Multiply instruction has a very simple implementation.

First, we convert the value in ramData at the address specified with the instruction to an integer and store it in a variable operand.

Then we multiply it with the value of AC and store it in AC.

Then we update the AC textbox in the front end.



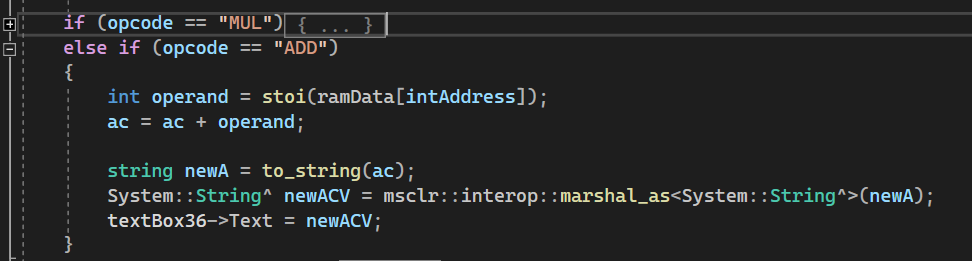
## ADD:

ADD instruction is very similar to MUL.

First, we convert the value in ramData at the address specified with the instruction to an integer and store it in a variable operand.

Then we add it with the value of AC and store it in AC.

Then we update the AC textbox in the front end.



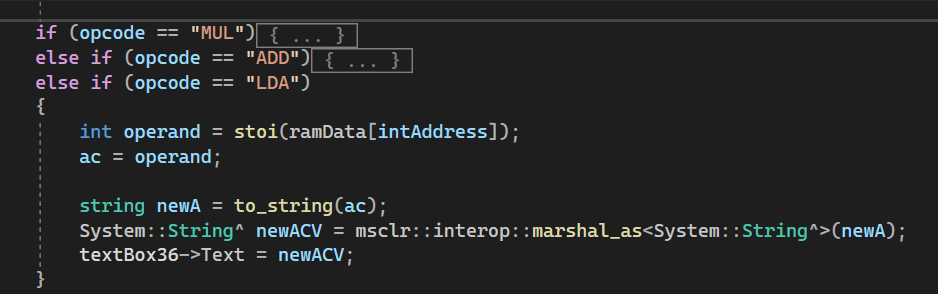
## LDA:

Load instruction is very similar to ADD and MUL.

First, we convert the value in ramData at the address specified with the instruction to an integer and store it in a variable operand.

Then we store it in AC.

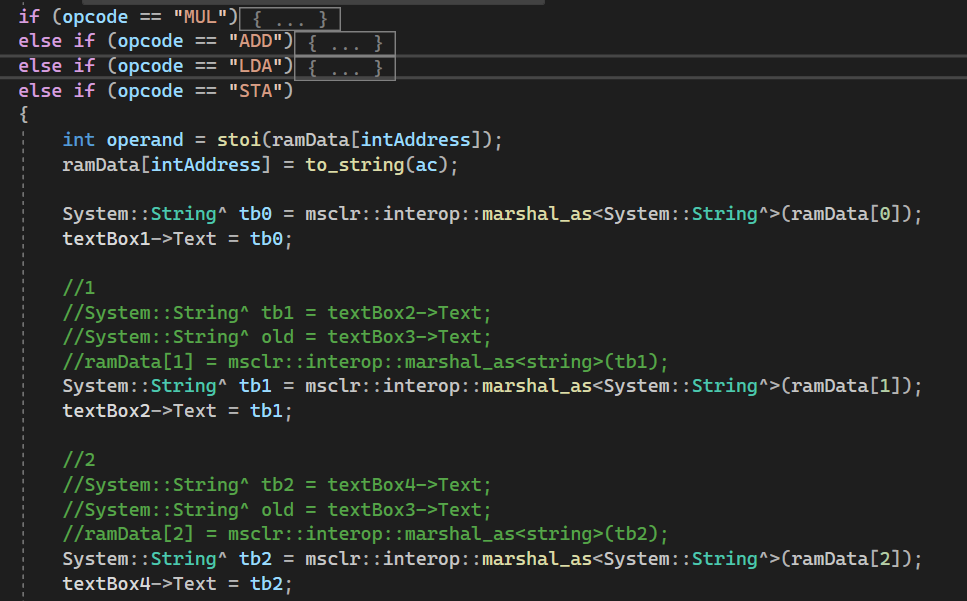
Then we update the AC textbox in the front end.



## STA:

STA has a bit of a complex implementation.

First, we covert value of AC (Int) to string and then update the ramData[] at the directed address with the value of ac. Then we update all the RAM text boxes in the front end which not only keeps the rest of the program there but will only update the required address.

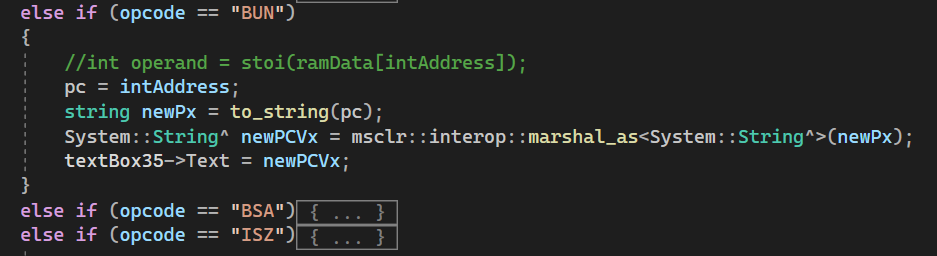


## BUN:

BUN has very simple implementation.

The value of pc gets updated the address specified in the instruction.

Then we update the value of PC for the front end.



BSA:

BSA is the most complex instruction to implement.

It combines both the concepts of STA and BUN.

The basic idea is that the instruction will branch off to the address specified but will store the address of the memory location where the BSA instruction was stored in the memory location it was branched to. It will then execute what ever instructions have been written below the memory location it was branched to. For returning back to the original assembly instruction we use the idea of Indirect addressing.

Our implementation of the BSA instruction eliminated the idea of indirect addressing for returning. Instead, we store a BUN instruction with the return address being its address part.

Now the return address could be a single digit memory location or 2 digit memory location, so when we store it with BUN we need to make sure that there are 2 bits present to minimize any error, hence, we write if statements to make sure that there is no error when it comes to reading the BUN instruction.

One important thing to remember is that we do not return exactly to memory location where BSA is stored at but rather the next memory location as we do not want BSA to be executed once again.Once we have updated the PC value to branched value we update the whole RAM same as with STA.



## ISZ:

The basic idea behind ISZ is that we increment the value stored at the memory location specified, then we check whether the value at the memory location is 0. If yes then we increment the PC effectively skipping the next instruction.

We start off by storing the value at ramData[intAddress] into an integer variable dr (data register). Then we increment the Data register.

Then we check if the value of dr is 0 or not. If yes then it will increment PC and update the new value in the front end.

Once the if statement is executed, we update the ramData[intAddress] with the new value of dr effectively updating the memory location with the new incremented value of dr.

Then same as STA we update every text box in the RAM to update the front end.

