

Assembler using C++

1. Introduction:

Brief overview of the project:

Our project is an assembler written using C++ programming language. The primary objective is to create a tool that translates human-readable assembly code into machine code, enabling computers to execute the specified instructions.

Objectives and significance of implementing an assembler:

An assembler eases the way programmers can write and debug codes. They use the human-friendly assembly code instead of the more complex machine code that results in more errors. They can also implement better optimization using low-level languages when working with systems like embedded systems than when using high-level languages.

2. Technical Background:

Basic concepts of assemblers and their role in computer organization:

Assembler converts assembly code to machine code for computers to understand and execute the provided instructions. It acts as an intermediary between the programmers using assembly code and the complexities of machine code.

Overview of assembly language and machine code relevant to the project:

Assembly language is human readable structure of instructions. Machine language is a numerical language that consists of 0s and 1s that tell a computer to directly execute an instruction. An assembler converts the instructions written in assembly code to machine language to be understood and executed by the computer.

3. Methodology:

Detailed description of the programming language used:

For the implementation of our assembler, we chose the C++ programming language.

C++ is efficient, adaptable, and provides low-level programming tools. We chose C++ as it is one of the languages that we are most skilled at making it much easier and more efficient when producing a better outcome for the project.

Explanation of the development environment and tools utilized:

We used Visual Studio 2022 as our main development platform. Visual Studio is a robust tool for developers, allowing all aspects of the development cycle in one place. This IDE equipped us with features like highlighting errors and debugging assistance, all of which boosted our efficiency during the coding process. We put it to work compiling C++ source code which guaranteed language compatibility and helped produce the final executable file.

4. Implementation Details:

We start off by including libraries and initializing used variables, strings, and arrays.

```
#include <iostream>
#include <stdio.h>
#include <string>
#include <fstream>
#include <cctype>
using namespace std;
```

```
string input, counter, sub;
string lines[100]; // array of strings to store all lines in the file
for (int i = 0; i < 100; i++)
{
    lines[i].clear();
}

ifstream inputFile;
int x = 0, check = 0, org = 0, j = 0, ct = 0, mempos = 0, g = 0, data = 0;
int space, found = 0;
string label, mode;
```

We created an array of strings to include the instruction and its hexadecimal code. The instructions were divided as memory-reference, register-reference and input/output reference.

```
int main()
{
    string MemoryInstruction[7][3] = {
        {"AND", "0", "8"},
        {"ADD", "1", "9"},
        {"LDA", "2", "A"},
        {"STA", "3", "B"},
        {"BUN", "4", "C"},
        {"BSA", "5", "D"},
        {"ISZ", "6", "E"}
    };

    string RegisterInstruction[12][2] = {
        {"CLA", "7800"},
        {"CLE", "7400"},
        {"CMA", "7200"},
        {"CME", "7100"},
        {"CIR", "7080"},
        {"CIL", "7040"},
        {"INC", "7020"},
        {"SPA", "7010"},
        {"SNA", "7008"},
        {"SZA", "7004"},
        {"SZE", "7002"},
        {"HLT", "7001"}
    };

    string InOutInstruction[6][2] = {
        {"INP", "F800"},
        {"OUT", "F400"},
        {"SKI", "F200"},
        {"SKO", "F100"},
        {"ION", "F080"},
        {"IOF", "F040"}
    };
}
```

The text file including input was read and each line was copied into an array of strings. This helped in the iteration and searching for labels with their memory locations.

Also, the pseudo instruction ORG was looked for to know how to create the symbol address table. The counter string takes the number along the ORG (if org was present) and converts it to integer into the ct variable.

The ct was decreased to be able to start the next instruction in the code with the correct memory location. If ORG was not present, we would start our ct from 100.

```
inputFile.open("C:/Users/ASUS/OneDrive/Desktop/C0 Project/input.txt");
if (inputFile.is_open())
{
    //cout << "File is Open" << endl;

    while (getline(inputFile, input))
    {
        if (x == 0)
        {
            if (input[0] == 'O' && input[1] == 'R' && input[2] == 'G') // looking for org
            {
                org = 1; // ORG is found
            }

            if (org == 1)
            {
                for (j = 4; j < input.length(); j++)
                {
                    counter += input[j];
                }

                ct = stoi(counter);
                ct--;
            }
            else
            {
                ct = 100;
            }
        }

        lines[x] = input; // Store each line in the array
        x++;
    }
}
```

The program starts iterating through the created array.

```
g = 0;
for (int f = ct; f < x; f++, g++)
{
    string line = lines[g];

    check = 0, data = 0;
```

If the length of the current line was more than 4 meaning it includes an address of some kind or a label.

If the character right after the instruction code is an integer, this means that it indicates a memory location. The instruction hexadecimal code is searched for in the made strings and output along with the memory location.

Also, the presence of I(indirect) is taken into consideration, and this is done by counting the number of spaces in the line. If there are two spaces, three words present so I is included.

If I is present, the third column in the memory instruction string is used, else the second column is used.

```
check = 0, data = 0;

if (line.length() > 4)
{
    if (is_integer(line[4]))
    {
        space = 0;
        for (j = 0; j < line.length(); j++)
        {
            if (line[j] == ' ')
            {
                space++; //count no.of spaces
            }
            if (space == 2)
            {
                break;
            }
        }
        if (space == 2) //2 spaces means there is an I present
        {
            mode = line.substr(j, 1);
            for (int i = 0; i < 7; i++)
            {
                if (MemoryInstruction[i][0] == line.substr(0, 3))
                {
                    if (mode == "I")
                    {
                        //cout << "Hex Code: " << MemoryInstruction[i][2] << line.substr(4, j - 4) << endl;

                        cout << MemoryInstruction[i][2] << line.substr(4, j - 4) << endl;

                        sub = line.substr(4, j - 4);

                        //cout << "Binary Code: " << hexToBinary(MemoryInstruction[i][2]) + hexToBinary(sub) << endl << endl;
                        check = 1;
                        sub.clear();

                        break;
                    }
                }
            }
        }
    }
    else
```

```

}
else
{
    mode = "";
    for (int i = 0; i < 7; i++)
    {
        if (MemoryInstruction[i][0] == line.substr(0, 3))
        {
            if (mode == "")
            {
                //cout << "Hex Code: " << MemoryInstruction[i][1] << line.substr(4) << endl;

                cout << MemoryInstruction[i][1] << line.substr(4) << endl;

                sub = line.substr(4);

                //cout << "Binary Code: " << hexToBinary(MemoryInstruction[i][1]) + hexToBinary(sub) << endl << endl;
                check = 1;
                sub.clear();

                break;
            }
        }
    }
}
```

If the character right after the instruction code is NOT an integer, this means that it indicates the presence of a label.

The label is searched for in the rest of the program to identify their location (symbol memory location). If their location exceeds 109, the hexadecimal code for values 10 and above is displayed instead.

The instruction hexadecimal code is searched for in the made strings and output along with the memory location of the label.

```
}  
else  
{  
    space = 0;  
    for (j = 0; j < line.length(); j++)  
    {  
        if (line[j] == ' ')  
        {  
            space++;  
        }  
        if (space == 2)  
        {  
            break;  
        }  
    }  
    if (space == 2)  
    {  
        label = line.substr(4, j - 4);  
        mode = line.substr(j + 1);  
    }  
    else  
    {  
        label = line.substr(4);  
        mode = "";  
    }  
    if (label.length() == 3)  
    {  
        for (int i = 0; i < 7; i++)  
        {  
            if (MemoryInstruction[i][0] == line.substr(0, 3))  
            {  
                mempos = i;  
                check = 1;  
                break;  
            }  
        }  
        found = 0;  
    }  
}
```



```

}
found = 0;
string temp;
string location;

for (int z = 0; z < x - ct; z++)
{
    temp = lines[z];

    if ((temp.substr(0, 3) == label) && temp[3] == ',')
    {
        found = 1;
        if (ct + z > ct + 9)
        {
            if (ct + z == ct + 10) { location = "10A"; }
            if (ct + z == ct + 11) { location = "10B"; }
            if (ct + z == ct + 12) { location = "10C"; }
            if (ct + z == ct + 13) { location = "10D"; }
            if (ct + z == ct + 14) { location = "10E"; }
            if (ct + z == ct + 15) { location = "10F"; }

            if (mode == "")
            {
                //cout << "Hex code: " << MemoryInstruction[mempos][1] << location << endl;
                cout << MemoryInstruction[mempos][1] << location << endl;

                //cout << "Binary Code: " << hexToBinary(MemoryInstruction[mempos][1]) << hexToBinary(location) << endl << endl;
            }
        }
        else
        {
            //cout << "Hex code: " << MemoryInstruction[mempos][2] << location << endl;
            cout << MemoryInstruction[mempos][2] << location << endl;

            //cout << "Binary Code: " << hexToBinary(MemoryInstruction[mempos][2]) << hexToBinary(location) << endl << endl;
        }
    }
}
else
{

```

```
    }
    else
    {
        if (mode == "")
        {
            //cout << "Hex code: " << MemoryInstruction[mempos][1] << ct + z << endl;
            sub = ct + z;
            cout << MemoryInstruction[mempos][1] << ct + z << endl;

            //cout << "Binary Code: " << hexToBinary(MemoryInstruction[mempos][1]) << hexToBinary(sub) << endl<<endl;
            sub.clear();
        }
        else
        {
            //cout << "Hex code: " << MemoryInstruction[mempos][2] << ct + z << endl;
            sub = ct + z;

            cout << MemoryInstruction[mempos][2] << ct + z << endl;

            //cout << "Binary Code: " << hexToBinary(MemoryInstruction[mempos][2]) << hexToBinary(sub) << endl << endl;
            sub.clear();
        }
    }

    location.clear();
}
if (found == 0)
{
    cout << "ERROR!! undeclared variable: " << label << endl << endl;
    break;
}

location.clear();
label.clear();
}
else
{
    else
    {
        if (label.length() >= 3 && line[3] == ',') //cout data after variable
        {
            //cout << "Hex code: " << line.substr(5) << endl;
            sub = line.substr(5);
            //cout << "Binary Code: " << hexToBinary(sub) << endl << endl;
            cout << line.substr(5) << endl;

            sub.clear();

            data = 1;
        }
        else {
            cout << "ERROR!! label has to be 3 or less characters and ends in a comma: " << line << endl << endl;
            check = 2;
            break;
        }
    }

    if (check == 0 && data == 0 && g > 0)
    {
        cout << "ERROR!! unknown instruction: " << line << endl << endl;
    }
}

if (check == 2)
{
    break;
}
```

The strings are cleared after usage to avoid over accumulation of data.

If an instruction was not found in memory, the check remains 0 and it is looked for inside the register reference string and the input/output reference string.

```
//check=0 not memory
//check=1 memory
//check=2 label error

if (check == 0)
{
    for (int i = 0; i < 12; i++)
    {
        if (RegisterInstruction[i][0] == line.substr(0, 3))
        {
            //cout << "Hex code: " << RegisterInstruction[i][1] << endl;
            cout << RegisterInstruction[i][1] << endl;

            //cout << "Binary Code: " << hexToBinary(RegisterInstruction[i][1]) << endl << endl;

            check = 1;
            break;
        }
    }

    for (int i = 0; i < 6; i++)
    {
        if (InOutInstruction[i][0] == line.substr(0, 3))
        {
            //cout << "Hex code: " << InOutInstruction[i][1] << endl;
            cout << InOutInstruction[i][1] << endl;

            //cout << "Binary Code: " << hexToBinary(InOutInstruction[i][1]) << endl << endl;

            check = 1;
            break;
        }
    }
}
```

```
if (check == 0 && g > 0 && data == 0)
{
    //cout << "Hex code: " << line << endl;
    if (line.length() == 3)
    {
        cout << "instruction not found " << endl;
    }
    else {
        cout << line << endl;
    }

    //cout << "Binary Code: " << hexToBinary(line) << endl << endl;
}

return 0;

else
    cout << "Unable to open the file" << endl;
    return 1;
```

Errors:

The presence of errors regarding the user entry were taken into consideration.

Labels for variables must be 3 or less characters and must be terminated with a comma. The programs display an error if the labels exceed 3 characters.

If an instruction was not found at all (invalid instruction), an error also gets displayed.

An error gets displayed if a variable was called and its label was not declared in the program.

Functions:

is_integer was used to determine if the character is a digit.

hexToBinary was used to convert the hexadecimal string to binary string.

5. Challenges and Solutions:

Challenge 1:

- Working with memory addresses and labels was difficult, particularly when figuring out addresses using the ORG value and handling labels in memory commands.

Solution:

- Two phases were implemented to overcome this challenge. On phase one, details about labels were collected and found the matching memory addresses. On phase two, this data was used to work out addresses when creating code.

Challenge 2:

- Difficulty in spotting and flagging undeclared variables or unfamiliar instructions, especially when dealing with labels and memory commands.

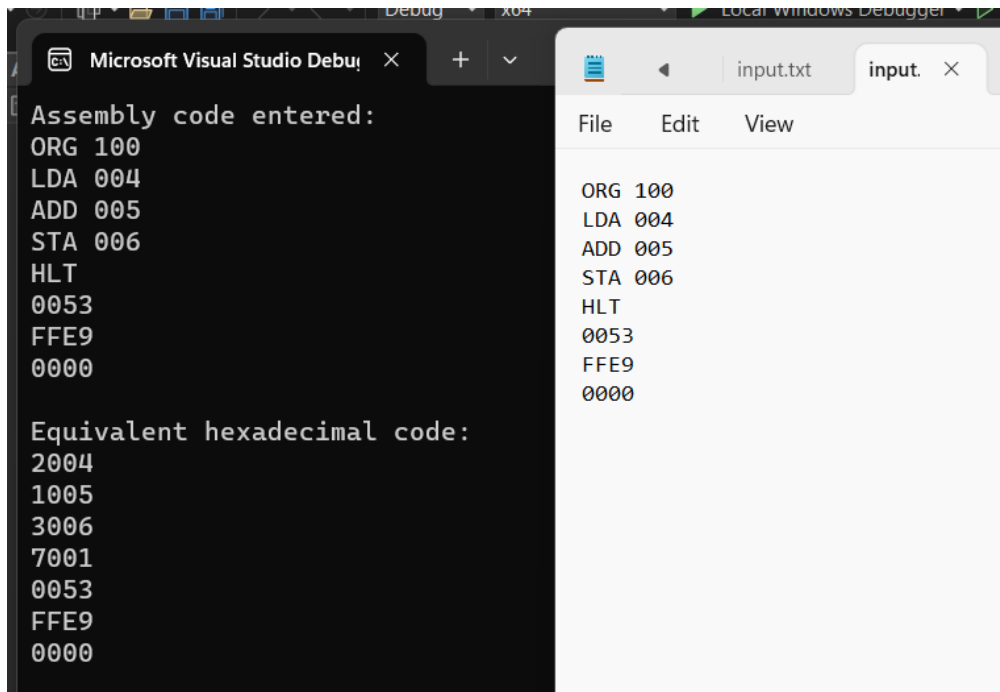
Solution:

- A detailed error-check system was created to spot and flag issues, like undeclared variables or unfamiliar instructions. The error messages were clear to help with debugging.

6. Results and Discussion:

Examples:

*Using ORG, no labels



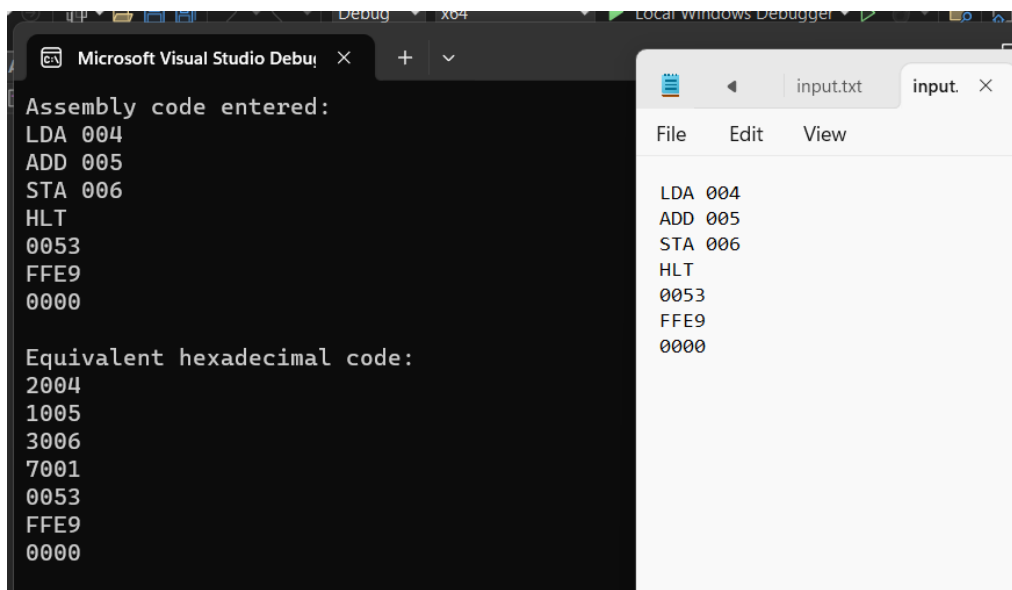
The screenshot shows the Microsoft Visual Studio Debugger interface. On the left, a black console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a white text editor window shows the same assembly code.

```
Assembly code entered:
ORG 100
LDA 004
ADD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
2004
1005
3006
7001
0053
FFE9
0000
```

```
File Edit View
ORG 100
LDA 004
ADD 005
STA 006
HLT
0053
FFE9
0000
```

*no ORG, no labels



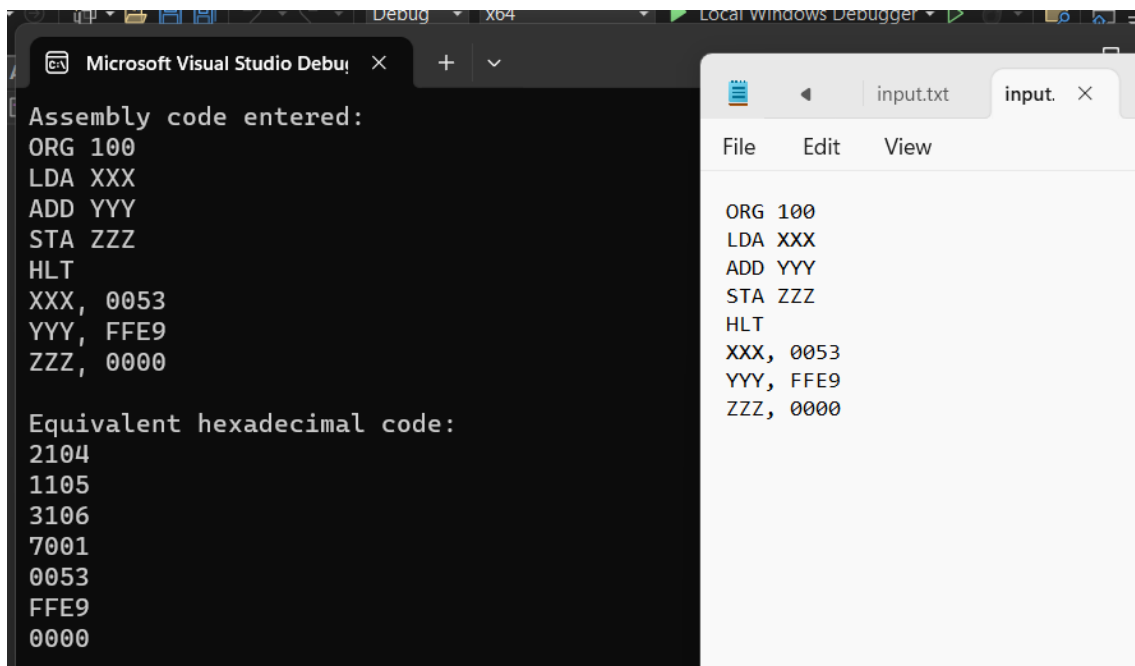
The screenshot shows the Microsoft Visual Studio Debugger interface. On the left, a black console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a white text editor window shows the same assembly code.

```
Assembly code entered:
LDA 004
ADD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
2004
1005
3006
7001
0053
FFE9
0000
```

```
File Edit View
LDA 004
ADD 005
STA 006
HLT
0053
FFE9
0000
```

***ORG, label**



The screenshot shows the Microsoft Visual Studio Debugger interface. On the left, a black console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a white text editor window shows the same assembly code.

```
Assembly code entered:
ORG 100
LDA XXX
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000

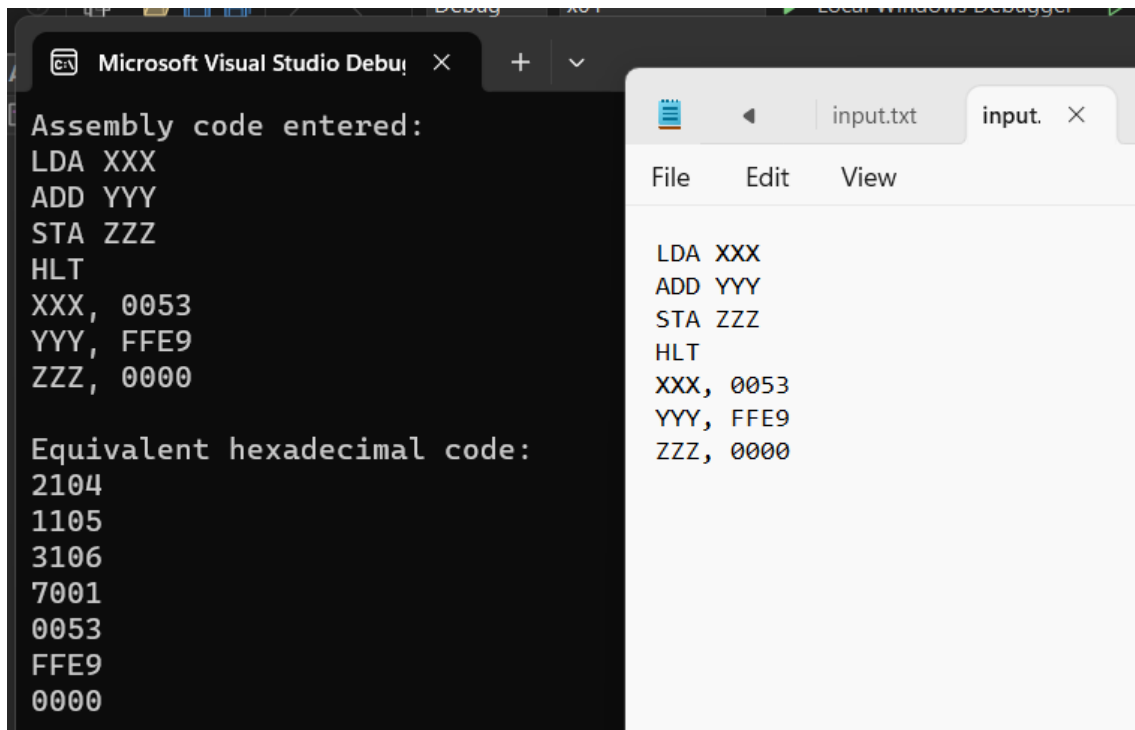
Equivalent hexadecimal code:
2104
1105
3106
7001
0053
FFE9
0000
```

input.txt

```
File Edit View

ORG 100
LDA XXX
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000
```

***no ORG, label**



The screenshot shows the Microsoft Visual Studio Debugger interface. On the left, a black console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a white text editor window shows the same assembly code.

```
Assembly code entered:
LDA XXX
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000

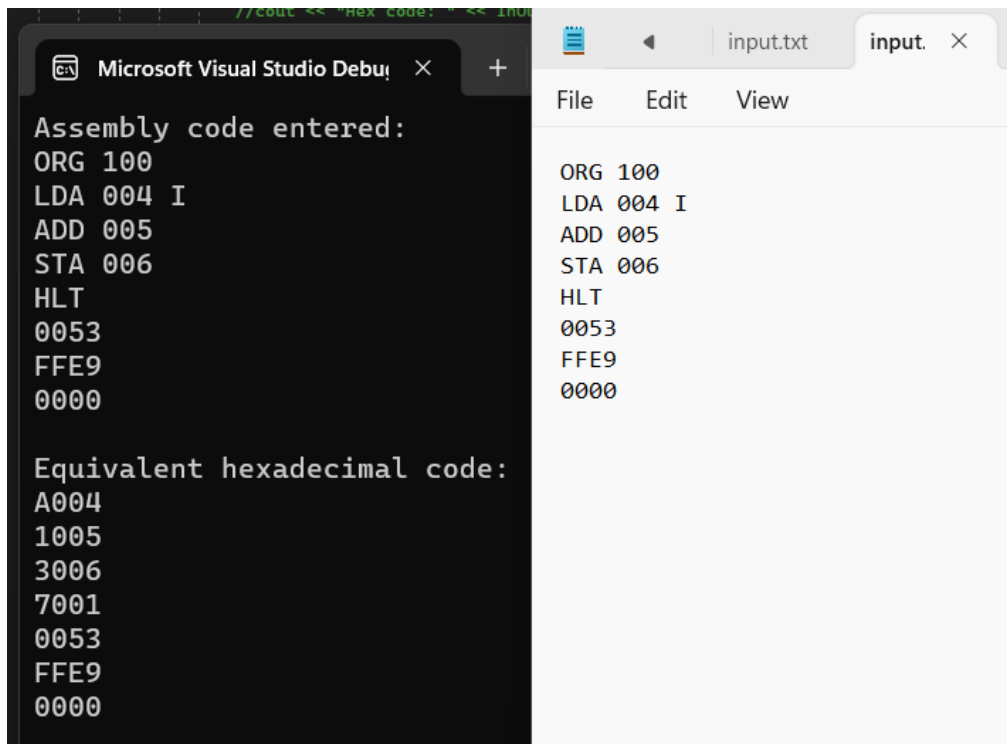
Equivalent hexadecimal code:
2104
1105
3106
7001
0053
FFE9
0000
```

input.txt

```
File Edit View

LDA XXX
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000
```

***indirect, no label**



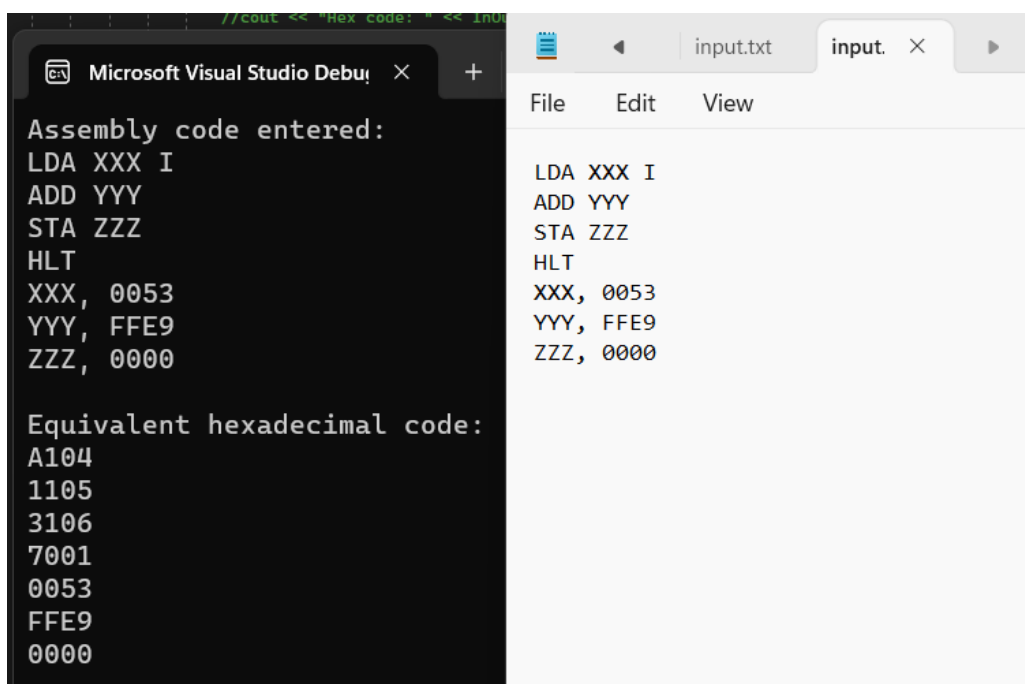
The screenshot shows the Visual Studio Debugger interface. On the left, a console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a file named 'input.txt' is open, showing the same assembly code.

```
Assembly code entered:
ORG 100
LDA 004 I
ADD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
A004
1005
3006
7001
0053
FFE9
0000
```

```
input.txt
ORG 100
LDA 004 I
ADD 005
STA 006
HLT
0053
FFE9
0000
```

***indirect, label**



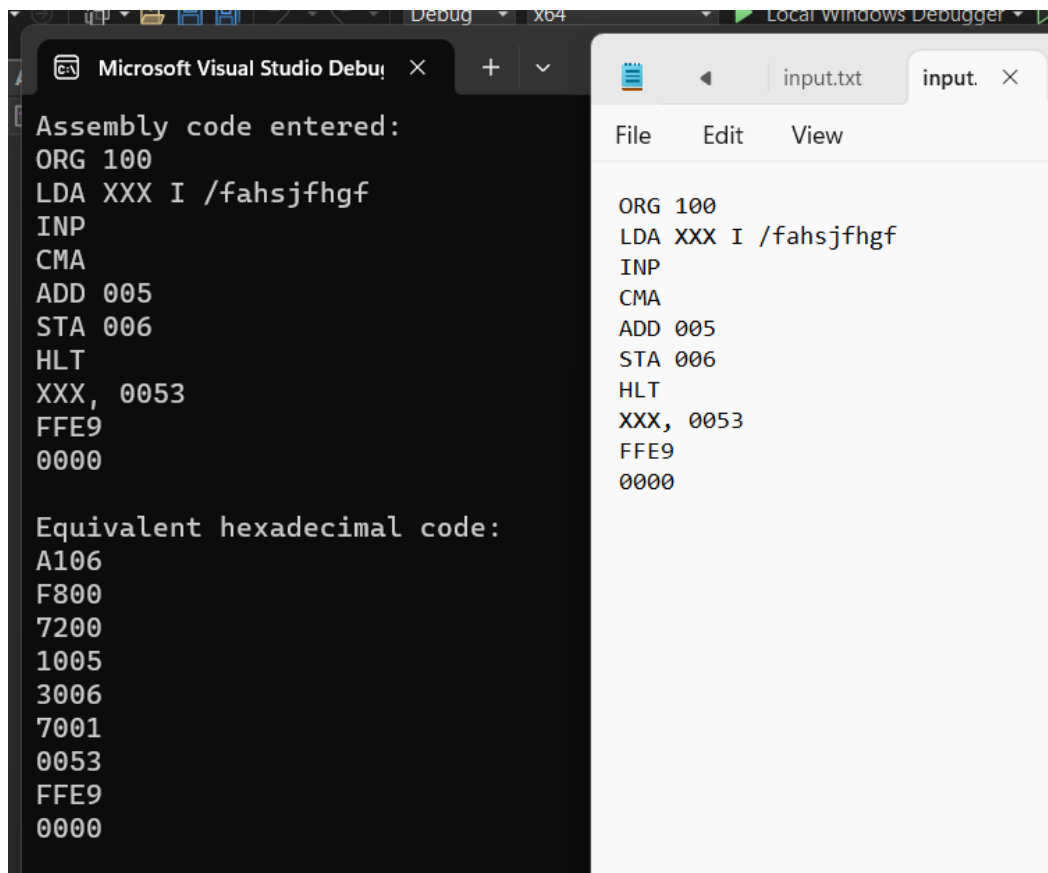
The screenshot shows the Visual Studio Debugger interface. On the left, a console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a file named 'input.txt' is open, showing the same assembly code.

```
Assembly code entered:
LDA XXX I
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000

Equivalent hexadecimal code:
A104
1105
3106
7001
0053
FFE9
0000
```

```
input.txt
LDA XXX I
ADD YYY
STA ZZZ
HLT
XXX, 0053
YYY, FFE9
ZZZ, 0000
```


***adding a comment**



The screenshot shows the Microsoft Visual Studio Debugger interface. On the left, a black console window displays the assembly code entered and its equivalent hexadecimal code. On the right, a white text editor window shows the same assembly code with a comment added to the LDA instruction.

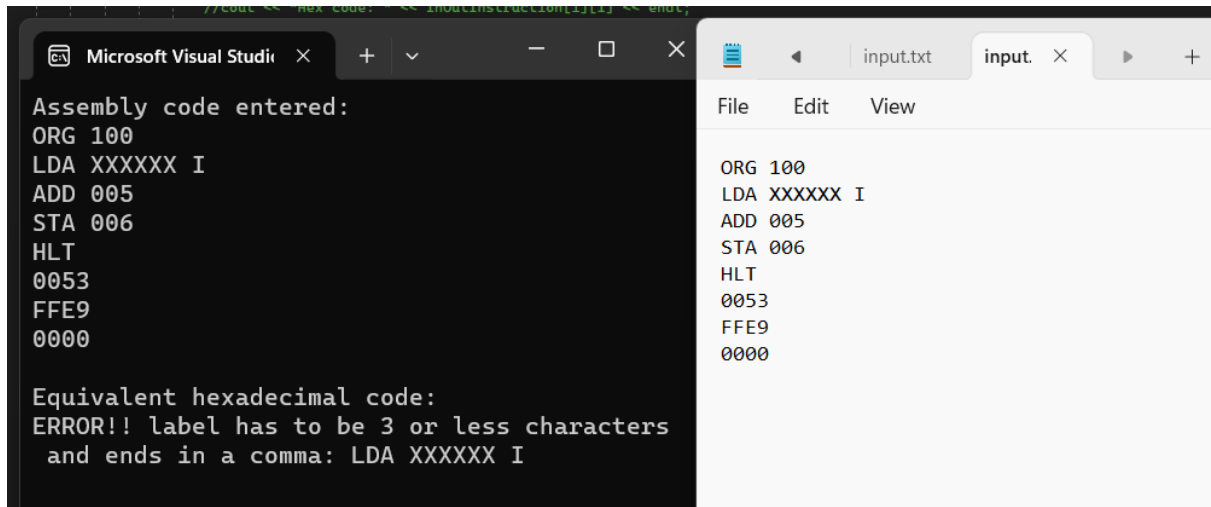
```
Assembly code entered:
ORG 100
LDA XXX I /fahsjfhgf
INP
CMA
ADD 005
STA 006
HLT
XXX, 0053
FFE9
0000

Equivalent hexadecimal code:
A106
F800
7200
1005
3006
7001
0053
FFE9
0000
```

```
input.txt
File Edit View
ORG 100
LDA XXX I /fahsjfhgf
INP
CMA
ADD 005
STA 006
HLT
XXX, 0053
FFE9
0000
```

ERRORS Examples:

*label > 3



The screenshot shows the Microsoft Visual Studio interface with two windows. The left window, titled 'Microsoft Visual Studio', contains the following assembly code:

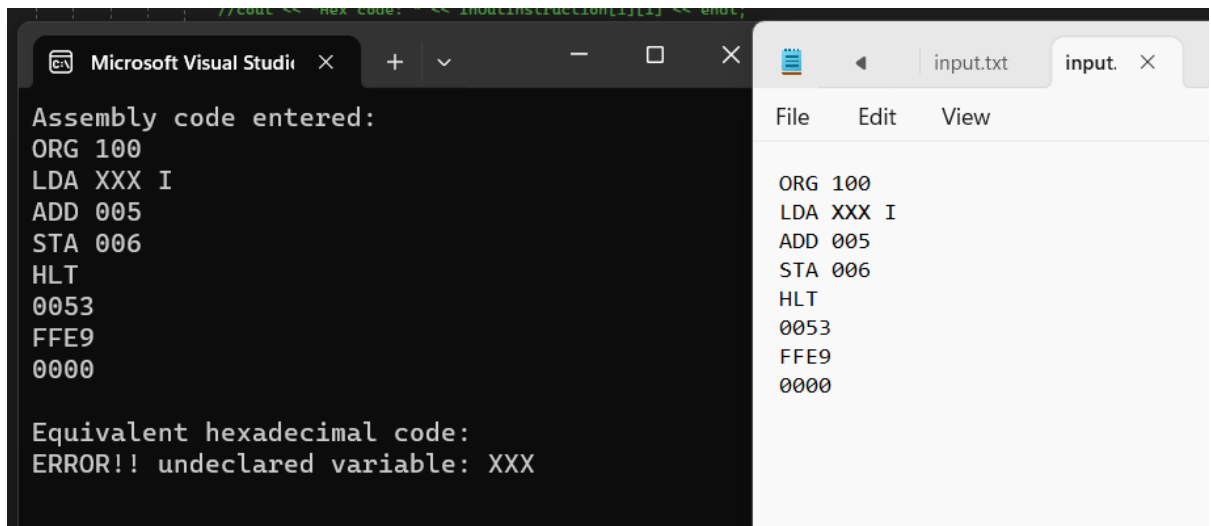
```
Assembly code entered:
ORG 100
LDA XXXXXX I
ADD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
ERROR!! label has to be 3 or less characters
and ends in a comma: LDA XXXXXX I
```

The right window, titled 'input.txt', shows the equivalent hexadecimal code:

```
ORG 100
LDA XXXXXX I
ADD 005
STA 006
HLT
0053
FFE9
0000
```

*undeclared label



The screenshot shows the Microsoft Visual Studio interface with two windows. The left window, titled 'Microsoft Visual Studio', contains the following assembly code:

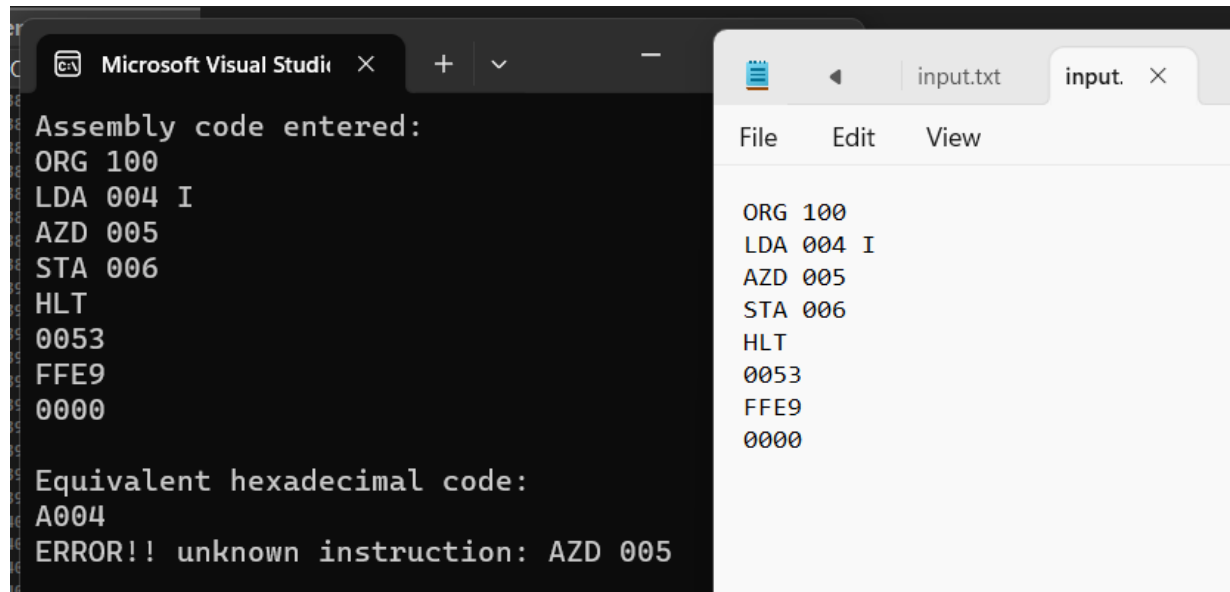
```
Assembly code entered:
ORG 100
LDA XXX I
ADD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
ERROR!! undeclared variable: XXX
```

The right window, titled 'input.txt', shows the equivalent hexadecimal code:

```
ORG 100
LDA XXX I
ADD 005
STA 006
HLT
0053
FFE9
0000
```

***unknown instruction**



The screenshot shows the Microsoft Visual Studio interface. On the left, a dark-themed console window displays the following text:

```
Assembly code entered:
ORG 100
LDA 004 I
AZD 005
STA 006
HLT
0053
FFE9
0000

Equivalent hexadecimal code:
A004
ERROR!! unknown instruction: AZD 005
```

On the right, a light-themed editor window titled 'input.txt' shows the same assembly code:

```
ORG 100
LDA 004 I
AZD 005
STA 006
HLT
0053
FFE9
0000
```

7. Conclusion:

The project provided a deep understanding of assembly language through the different types of instruction references. By making strong error-checking systems, we learned about handling undeclared variables and unknown commands.

The project experience was beneficial as it showed us the importance of following the rules for any type of application or program to ensure its successful execution.

The two-pass approach indicates how vital it is to plan and design your program to face its complex requirements.