“ASM IDE: Making Assembly Programming

Easier with Machine Learning”

Special Assignment Report

Submitted in Partial Fulfillment of the Requirements for

the Subject

EC 622: MODERN PROCESSOR ARCHITECTURE

Submitted By

Anant Raina: (17BEC010)

Yash Patel: (17BEC070)



Department of Electronics & Communication

Engineering

Institute of Technology

NIRMA UNIVERSITY

Ahmedabad 382 481

**Acknowledgement**

*29th March 2020*

*We would like to acknowledge the efforts of Dr. Dhaval Shah and Dr. Sachin Gajjar for guiding us through the project even under trying times. This project would not have seen the light of day without their warranted efforts in the form of valuable inputs*

*We thank them for their continued trust in our work ethic and we look forward to doing projects with them in the future.*

*Thank you*

*Anant Raina 17bec010*

*Yashkumar Patel 17bec070*

# **INDEX**

## **1.1 Introduction**

## **1.2 Pre-requisites**

### **1.2.1 Manual Installation**

## **1.2.2 Automatic Installation**

**1.3 Getting Started**

### **1.3.1 The Basics**

### **1.3.2 The GUI**

### **1.3.3 Simulator**

### **1.3.4 UI Manipulation**

**1.4 Internal Mechanics**

### **1.4.1 Flow Diagram**

### **1.4.2 Flow Working**

### **1.4.3 Simulator**

### **1.4.4 History**

## **1.5 Machine Learning**

## **1.6 Dataset**

## **1.7 Conclusion**

### **1.7.1 Future Scope**

## **1.8 References**

**1.1 Introduction**

In the early days of computers, programmers wrote programs in machine code, but then for the ease of the programmers and better readability, programming assembler languages were developed. Assembly programming requires knowledge of an instruction set. An assembler is a program that converts assembly language into machine code. It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor.

In computer programming, assembly language is a low-level programming language. Assembly language is denoted as ‘asm’ in which there is strong correspondence among the instruction in the language and machine code instruction. Assembly depends on machine code instructions and every assembler has its own assembly language which depends on the architecture of a computer.

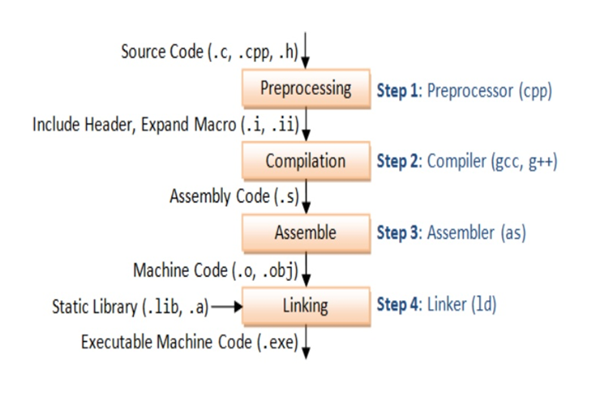


Fig 1: Data Flow diagram of building a source code

The programmer can write a program using a sequence of these assembler instructions. This sequence of assembler instructions, known as the source code, is specified to the assembler when a program is started and generates a corresponding bit stream. The output of the assembler program is called the object program relative to the input source program.

**1.2 Pre-requisites**

**Github Link:** https://github.com/KillingJoke42/ASM\_IDE

**1.2.1 Manual Installation**

1. python >= 3.6.9

Windows Users:

* Get Anaconda: Follow rest of the install as ubuntu

<https://www.anaconda.com/distribution/>

* Install Python3 Standalone

<https://www.anaconda.com/distribution/>

Install pip3

<https://pypi.org/project/pip/>

Ubuntu Users:

sudo apt-get update

sudo apt-get install python3

sudo apt-get install python3-pip

1. yaml >= 3.12

Ubuntu Users:

(non-pip) sudo apt-get install python3-yaml

python3 -m pip install yaml

Windows Users:

* Anaconda Users

pip install yaml

* Standalone (in cmd)

python3 -m pip install yaml

1. json >= 2.0.9

(embedded in python by default)

1. numpy >= 1.18.0

Ubuntu Users:

(non-pip) sudo apt-get install python3-numpy

python3 -m pip install numpy

Windows Users:

* Anaconda Users

pip install numpy

* Standalone (in cmd)

python3 -m pip install numpy

1. sklearn >= 0.22.2.post1

Ubuntu Users:

(non-pip) sudo apt-get install python3-sklearn

python3 -m pip install sklearn

Windows Users:

* Anaconda Users

python -m pip install sklearn

* Standalone (in cmd)

python3 -m pip install sklearn

1. pyqt5

Ubuntu Users:

(non-pip) sudo apt-get install python3-pyqt5

python3 -m pip install yaml

Windows Users:

* Anaconda Users

pip install pyqt5

* Standalone (in cmd)

python3 -m pip install pyqt5

1. pyaudio >= 0.2.11

Ubuntu Users:

(non-pip) sudo apt-get install python3-pyaudio

python3 -m pip install pyaudio

Windows Users:

* Anaconda Users

python -m pip install pyaudio

* Standalone (in cmd)

python3 -m pip install pyaudio

1. SpeechRecognition >= 3.8.1

Ubuntu Users:

python3 -m pip install SpeechRecognition

Windows Users:

* Anaconda Users

pip install SpeechRecognition

* Standalone (in cmd)

python3 -m pip install SpeechRecognition

**1.2.2 Automatic Installation**

Ubuntu Users can directly use the “install.sh” file with the

cloned repository and run in the terminal (in the repo directory):

sudo apt-get update

git clone <https://github.com/KillingJoke42/ASM_IDE>

cd ASM\_IDE

sudo bash install.sh

An executable for Windows is in development: keep referring to the repository for updates on the installer

**1.3 Getting Started**

**1.3.1 The Basics**

The application, if run via the python interpreter, must be run using "python3 asm\_ide.py" The following files exist in the repository that comprises of the source code:

1. asm\_ide.py -- The main GUI application
2. classify.py -- houses the Machine Learning backend of the application
3. CompilerForMachineCode.py -- Converts given assembly instruction into machine code
4. interpreter1.py -- responsible for converting user input in english to assembly code
5. simulator.py -- simulator that executes the instructions, assuming the role of a virtual RISC processor
6. Speech\_recog.py -- convert speech to text via Google's Speech Recognition API
7. styles.py -- theme settings for the GUI
8. trainlabel.py -- houses the training labels for the Machine Learning model

**1.3.2 The GUI**

|  |  |
| --- | --- |
|  |  |
| Figure 2: Different themes available for ASM\_IDE | |

The GUI is extremely user-friendly, that is to say, it is very customizable and is intuitive, allowing for ease of usage and understanding. To begin writing code, simply start typing in the text box at the bottom, and after writing your first instruction, press “Enter” to submit or the “submit” button to the right of the text box.

|  |
| --- |
|  |
| Figure 3: The text box with the submit button |

Let us try to add an instruction “add $s1, $s2, $s3”.

In the submit box, type “add s1 and s2 and store the result in s3”. Like so:



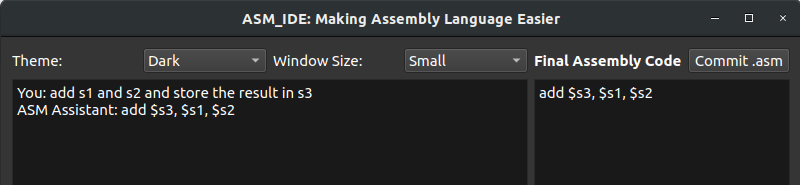
Press “Submit” to compute the instruction.



You may also use the “mic” icon and use the speech feature to submit the instruction you wish to execute.



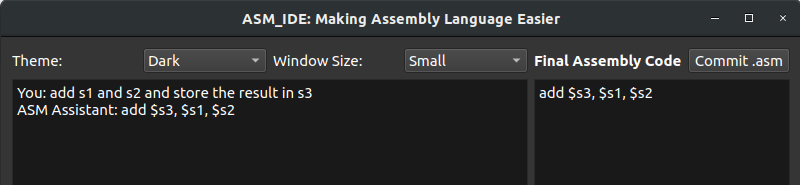
Violà! Your instruction is ready! Observe output in the ASM window on the right.



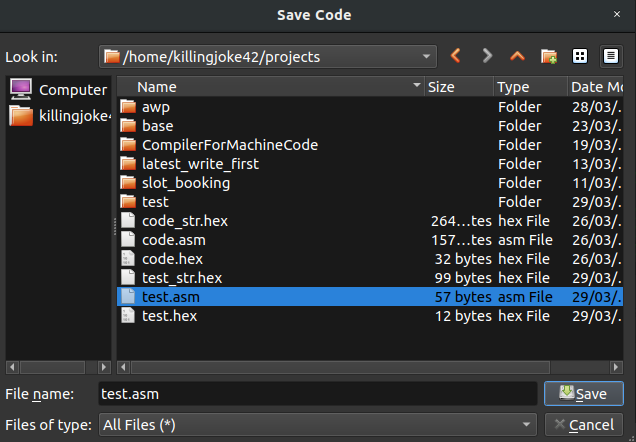
ASM\_IDE supports a total of 40 instructions, excluding procedures. While procedures are supported by the hexadecimal converter, they are not as of this date (29th March, 2020) executable in the simulator.

To try all of the supported english content, familiarize yourself with the training dataset provided in the repository under the name “traindata.txt”.

To continue you must save your ‘.asm’ file. Press the “Commit .asm” button at the top right corner of the screen



A window will open. Write the name of the file you want to save as with the ‘.asm’ extension. Then, press ‘save’.



**1.3.3 Simulator**

Let us now try to simulate the code that we just published. For the purpose of the discussion, we will use pre-made code from asm\_ide.

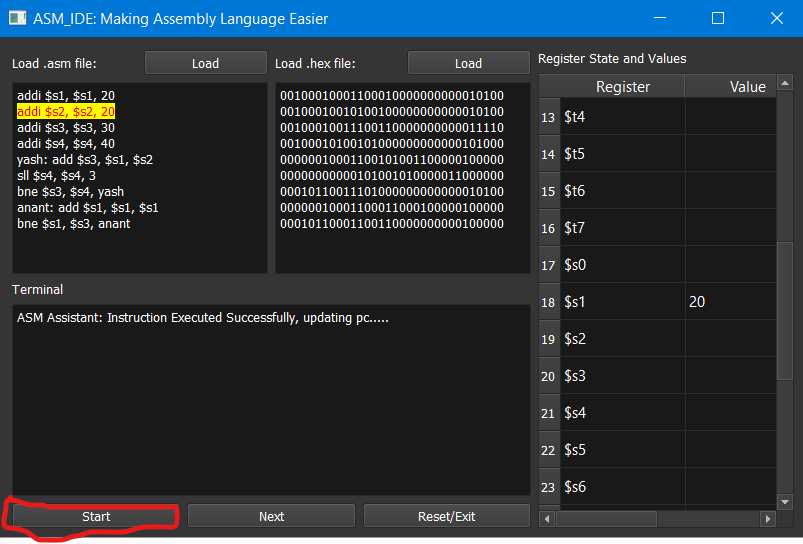
First, click on the “Load” button next to the code area to import the ‘.asm’ file.



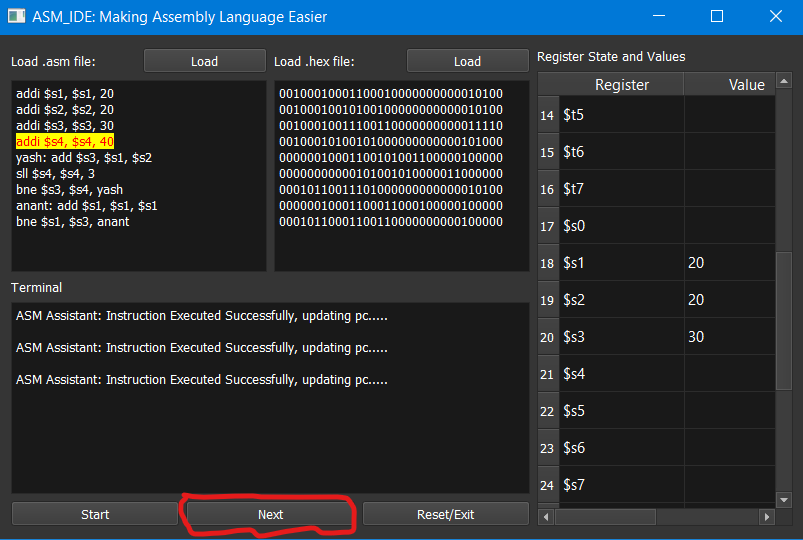
Next, import the ‘.hex’ file generated in the previous tutorial. (NOTE: Do not import the ‘\_str.hex’ file. It is a system file generated by the code. Also, do NOT delete this file. It is being used by asm\_ide.)



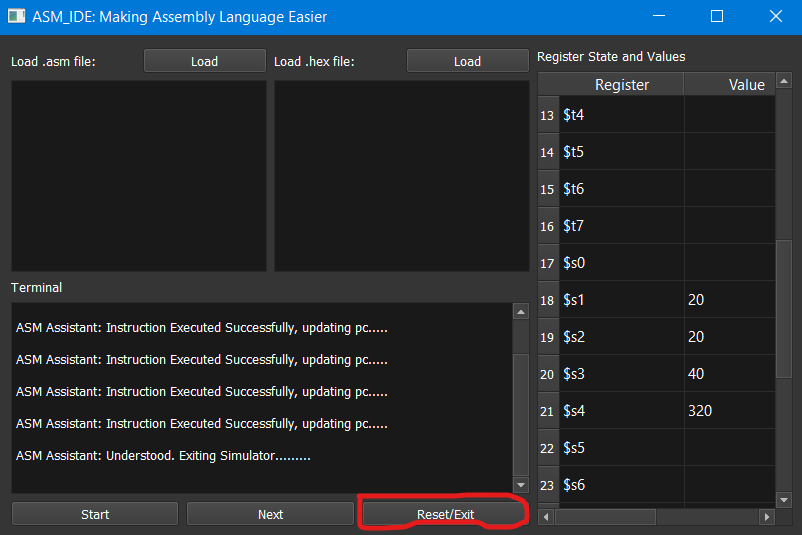
Now, press the “start” button. You will be informed by ASM-assistant that the execution has started. Also, the instruction that is next in the queue is highlighted.



To continue execution, press “next”. The next instruction will be executed. Do observe that the values of the registers also change with change in the instruction.



If you are done with simulation, or if the whole code has executed, press the “reset” button, to reset the simulator, or press “Start” to start the simulation all over again.



**1.3.4 UI manipulation**

There are many options available to change the UI elements of ASM\_IDE. At present, two are accessible to the user: “Change Theme” or “Change Window Size”. Try to change them to your convenience.

**1.4 Internal Mechanics**

**1.4.1 Flow Diagram**

|  |
| --- |
|  |
| Figure 3: Data flow Diagram |

**1.4.2 Flow Working**

In order to have more clarity while explanation, the paths have been color coded as per the following color scheme:

History, Speech-to-text data, user\_input to output dataflow, Training ML model

Data is first interpreted in the form of speech or in the form of text bypass and is returned back to the user via the GUI terminal. At the same time, the speech that is now converted to english text is sent to the Machine Learning Algorithm housed in ‘classify.py’ that makes a prediction based on its training. The classes are as follows:

[“arith”, “logic”, “copy”, “mem” (alias for memory), “control”, “shift”]

|  |
| --- |
|  |
| Figure 4: All instructions supported by asm\_ide |

Once this operation is performed, ‘interpreter1.py’, converts the given english text into assembly code, and relays it back to the GUI engine and to the ‘.asm’ file that the user saves. Once saved, ‘asmCall.py’ is tasked with performing to tasks:

1. Generate a symbol table
2. Convert all the assembly code to machine code and save a ‘.hex’ file

To generate the ‘.hex’ file, asmCall needs the help of ‘CompilerForMachineCode.py’ to convert an assembly instruction into machine code. asmCall compiles all the machine codes returned by CompilerForMachineCode and writes them to a ‘.hex’ file and to a ‘\_str.hex’ file. The ‘\_str.hex’ is different only in the sense that it houses the binary bits in ‘ASCII’ format. Hence all lines in this file are actually binary strings and not binary. The file is next written to the GUI engine for the user to see, and the assembling task is complete.

**1.4.3 Simulator**

The entire task of simulation is handled by ‘simulator.py’ and the GUI engine. The tasks given to ‘simulator.py’ is as follows:

1. Get the machine code from the HEX file
2. Get the registers used and the opcode and function
3. Keep track of a ‘register file’ with the values of all the variables
4. House all the functions that execute the action of an instruction

The GUI engine supplements these tasks by adding visual elements to the simulation process.

Simulator follows the following conventions when interpreting machine code:

|  |
| --- |
|  |
| Figure 4: The interpret convention |

**1.4.4 History**

As the english text is received from ‘Speech\_recog.py’, it is stored in a stack data structure so as to have a feel of the classic Linux terminal, where pressing the up and down arrow keys results in the user going back through the history of executed instructions. The user may also say “remove last” or sentences of similar annotations to remove the last instruction that was entered into the code area.

**1.5 Machine Learning**

The Machine Learning pipeline comprised of three different elements:

1. **CountVectorizer**: This is used to transform a corpora of text to a vector of term / token counts. It also provides the capability to preprocess your text data prior to generating the vector representation making it a highly flexible feature representation module for text. Here it was used to get the frequency of certain tokens that occur frequently when talking of a particular class. For instance, the term “add” occurs very frequently in an “arith” type instruction
2. **TFIDF Transformer**: This is a common term weighting scheme in information retrieval, that has also found good use in document classification. The goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus. Here, we have used TFIDF to assign weights to each frequented token for a particular task, that directly influences what the instruction type is. For instance, the term “store” occurs frequently in both “arith” and “shift” instructions but is not a parameter that can distinguish between the two. So, it is weighted low by the TFIDF.
3. **OneVsRestClassifier:** When we want to do multiclass or multilabel classification and it's strategy consists of fitting one classifier per class, OneVsRestClassifier is ideal. For each classifier, the class is fitted against all the other classes. The class having probability distribution more than the rest is the predicted class. In our case, there are six total classes.

**1.6 Dataset**

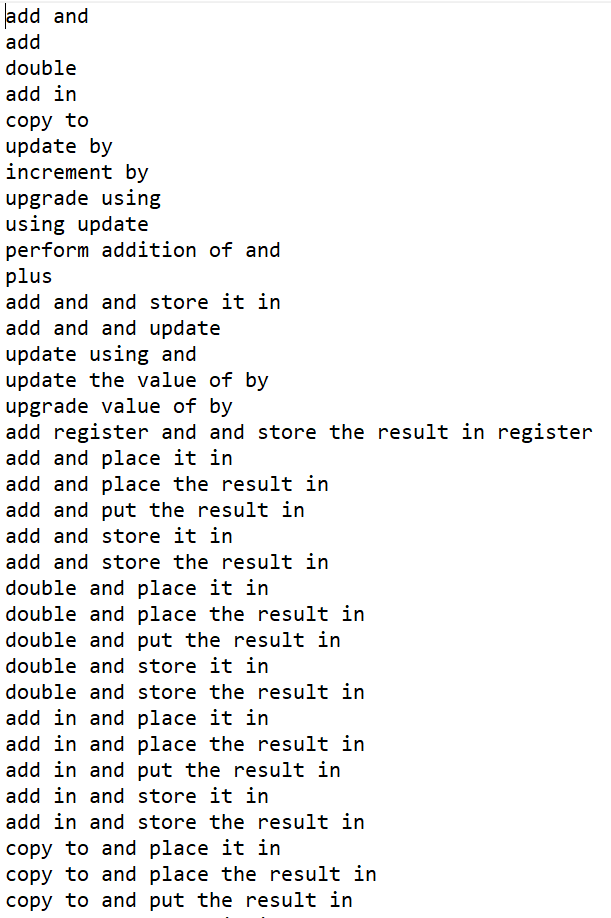
There are 494 samples in the dataset. In the dataset, there are 6 classes like arith, logic, copy, shift, memory and control.

Class Total Samples Class Total Samples

Arith 248 Shift 18

Logic 105 Memory 69

Copy 16 Control 45

****

Since no dataset was available for assembly language instructions to the best of our knowledge, we created our own dataset as stated above and will now upload it for researchers to forward it for use in miscellaneous AI applications.

**1.7 Conclusion**

It was the aim of this project to create a system that is capable of understanding human-friendly language and help convert it into assembly language code that can also be simulated for testing and further reviewed by the user. It has now become an all-in-one package for RISC enthusiasts

AI is jumping leaps and bounds and there is already research towards AI that writes code by itself. ASM\_IDE, as it currently stands, aims to be the first step to this milestone by accepting assembly language as its base

ASM\_IDE has its limitations, but most definitely serves as a starting step for learners new to RISC ISA. The dataset developed can also be used by researchers further to help fuel AI research. It can also be used to test Assembly programs to check reliability of code.

**1.7.1 Future Scope**

We now wish to fix a blaring issue that is incompatibility with procedures, i.e, the jal and jr instructions. Further, we wish to add a ‘dynamic dataset’ feature that will enable users to enter new data samples into the dataset if the ML prediction fails, that helps expand the reach of the ML algorithm. We also aim to introduce UI performance improvements and bug fixes as the git repository becomes public and is challenged by open-source enthusiasts around the globe.

**1.8 References**

1. *Remote-Intelligent-Assistant (REIA)* , aseem96, [online], available, <https://github.com/aseem96/Remote-Intelligent-Assistant-REIA>
2. *“Computer Architecture”, B. Parhami****,*** *Oxford University Press, New York, NY, USA, 2005.*

# *“Hands-On Machine Learning with Scikit-Learn and TensorFlow”,* AurÈlien GÈron, O’Reilly Publications, NY, USA, 2017