How to Use this Template

1. Make a copy [File → Make a copy...]
2. Rename this file: "**Capstone_Stage1**"
3. Replace the text in green

Submission Instructions

1. After you've completed all the sections, download this document as a PDF [File → Download as PDF]
2. Create a new GitHub repo for the capstone. Name it "**Capstone Project**"
3. Add this document to your repo. Make sure it's named "**Capstone_Stage1.pdf**"

GitHub Username: Izodine

iAssembly

Description

iAssembly (or iASM for short) is an x86 assembly simulator designed to teach the basics of 32-bit Assembly. It uses NASM (Netwide Assembler) as a basis. One of the main points of the app is to show that while assembly looks very intimidating, it is not that difficult to learn the basics of how it works. It is also designed to demonstrate how low level computer functions work to aid in designing better software. It contains a tutorial section that outlines various assembly and low level concepts. It provides exercises and allows you to “link” the code editor to it. It’ll provide the instructions for the exercise as well as a button to check if the user was correct.

Intended User

The intended user would be anyone learning or wanting to learn how to program. It can even be useful for users who are experienced in programming but unaware of how low level operations work.

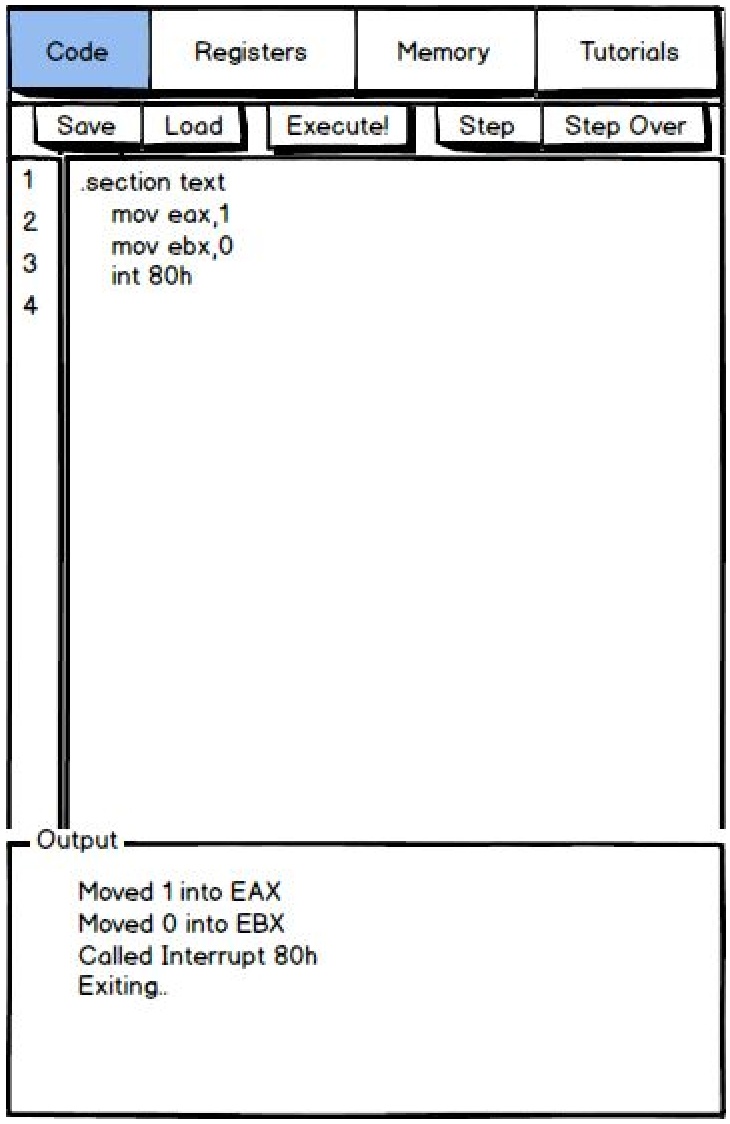
Features

- A simulated environment for basic x86 Assembly using Netwide Assembler as a basis for syntax.
- Basic Syntax Highlighting
- Ability to save and load programs.
- Detailed explanations and tutorials that link to the code editor.
- Ability to see what values are in each register.
- Ability to observe the stack and heap.

User Interface Mocks

These can be created by hand (take a photo of your drawings and insert them in this flow), or using a program like Photoshop or Balsamiq.

Code Editor

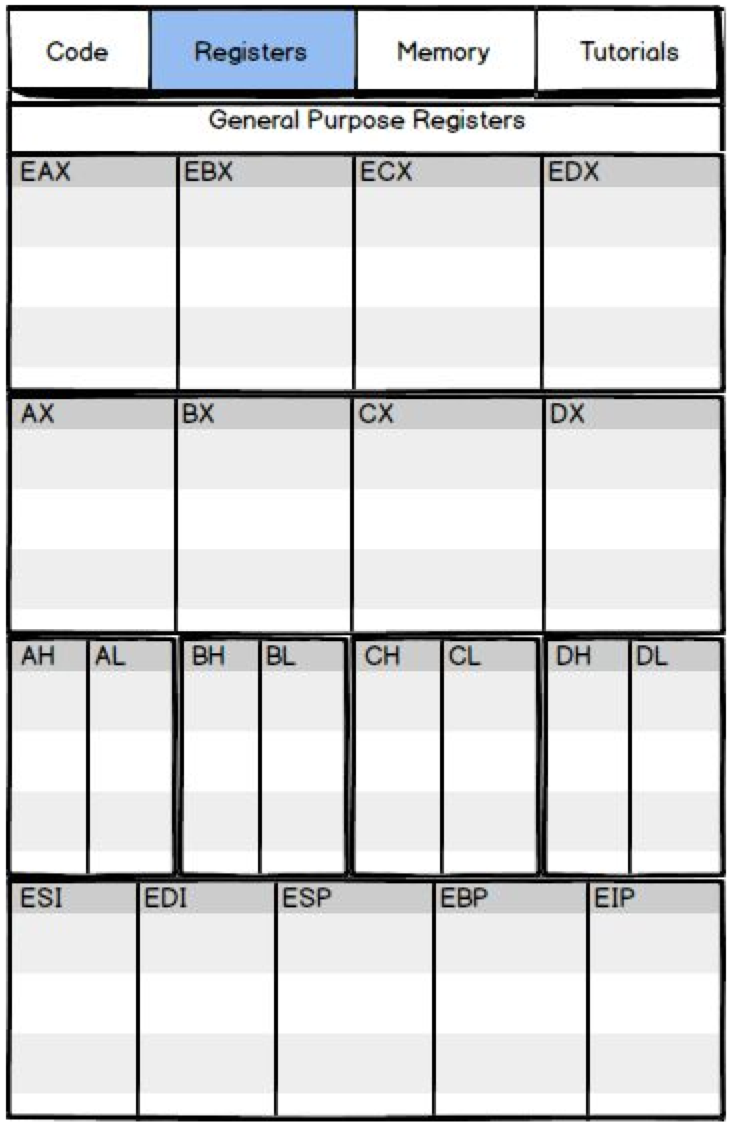


```
.section text
    mov eax,1
    mov ebx,0
    int 80h
```

```
Moved 1 into EAX
Moved 0 into EBX
Called Interrupt 80h
Exiting..
```

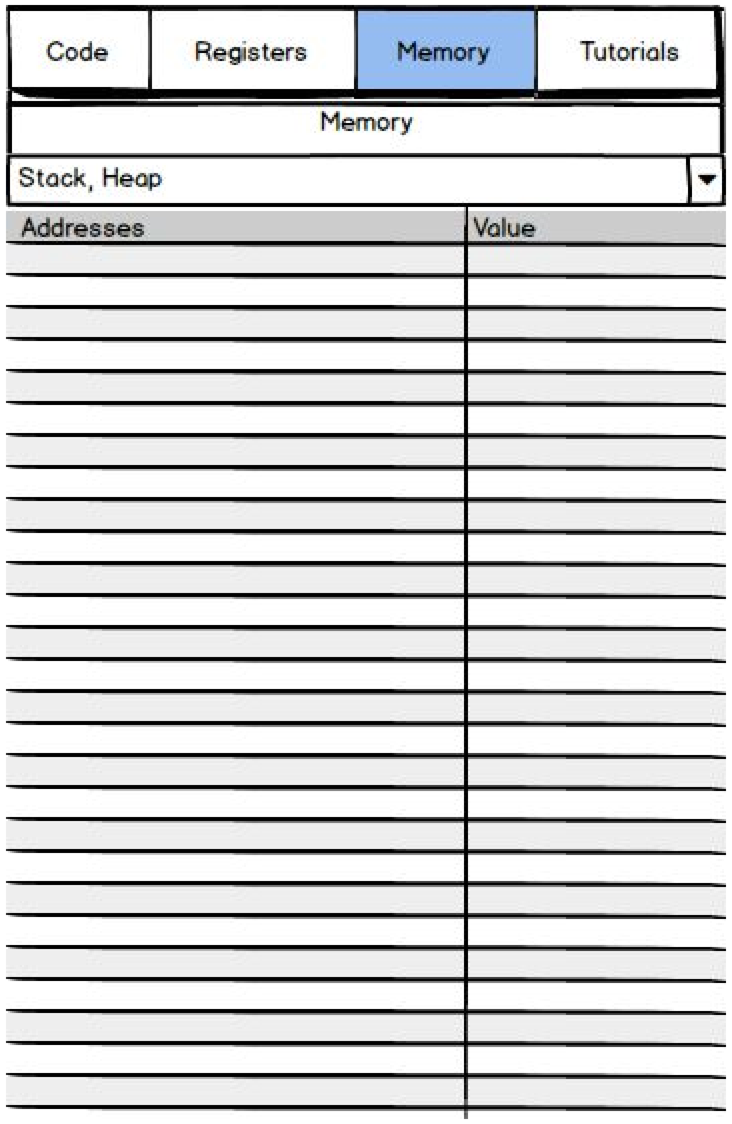
The app is separated into a couple different screens using tabs. This is the code editor. It allows the user to save/load programs, as well as execute, step, and step over. There will be basic syntax highlighting as well as line numbers. At the bottom there is an output window that displays the result of each line.

Register Screen



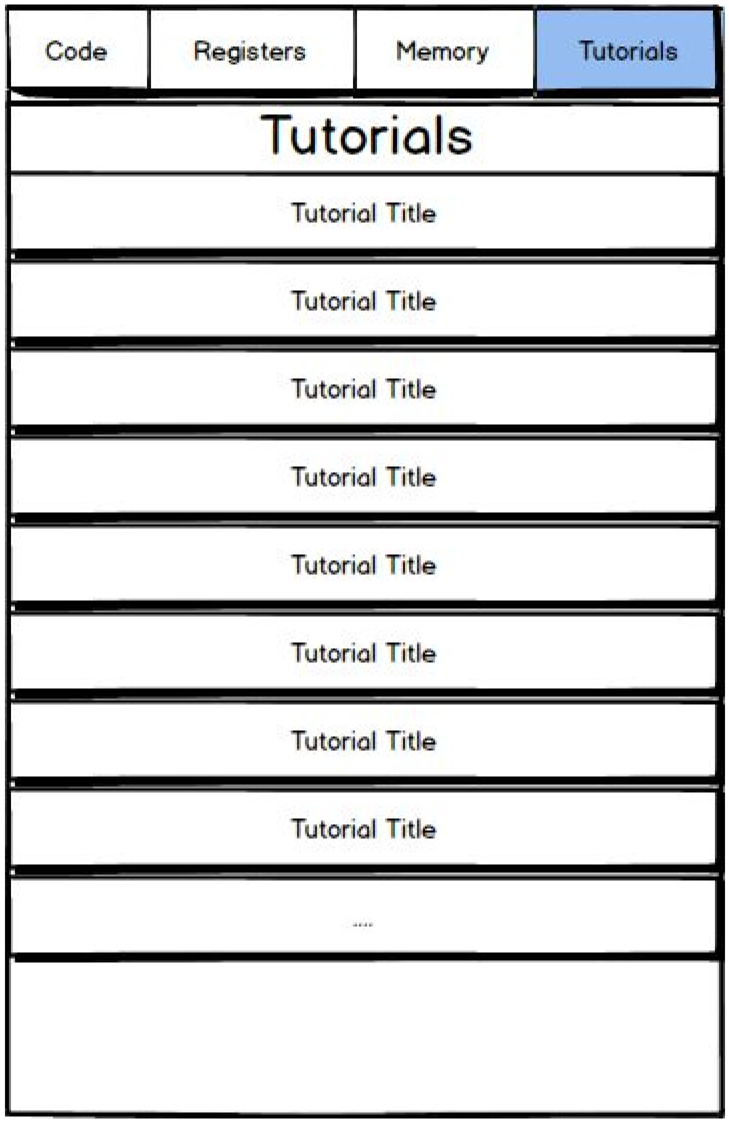
This screen shows what is currently in each register. It also breaks down the extended standard registers into their 16 bit sub parts.

Memory Screen



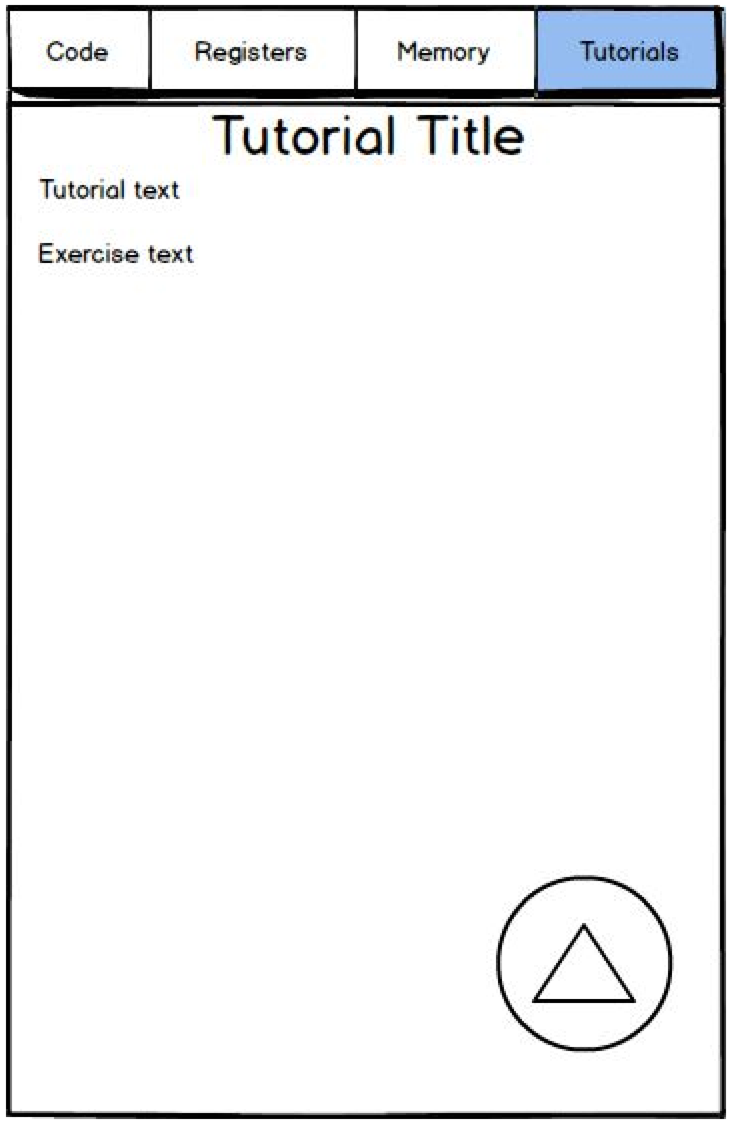
This screen shows a graph of what each address corresponds to value wise. It also has a box that lets the user select between stack and heap.

Tutorial Screen



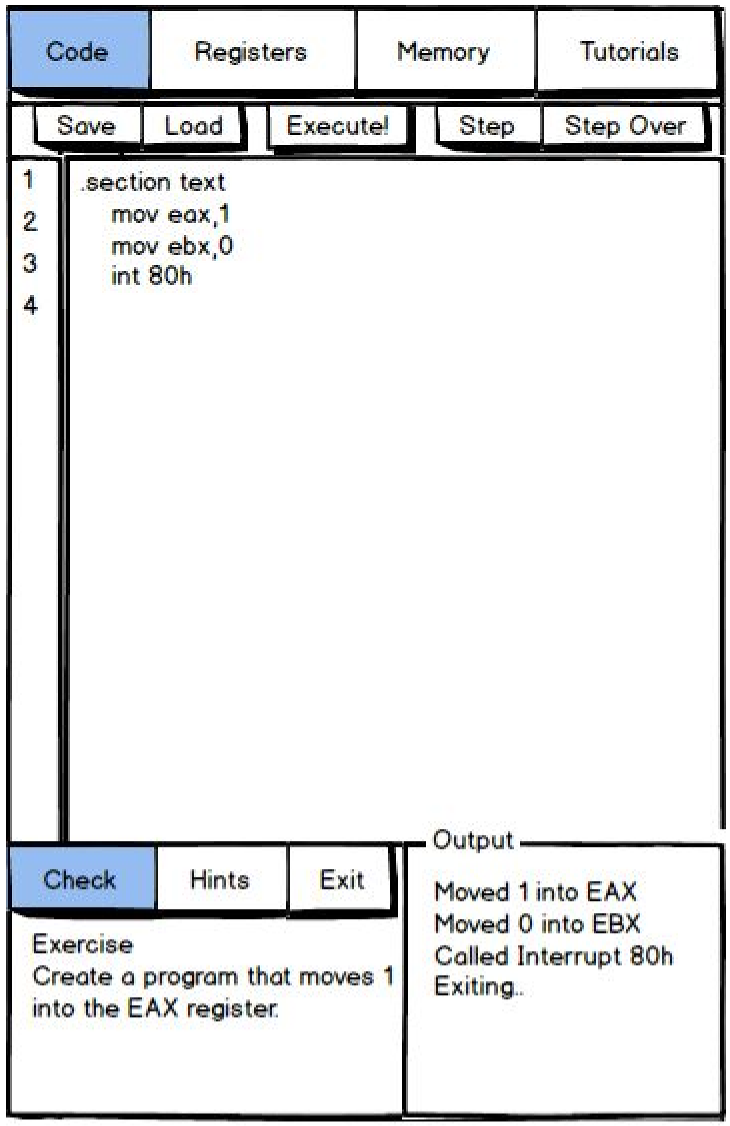
This screen is the listing for tutorials. Upon clicking on an entry it will flip to a detail fragment for that specific tutorial.

Tutorial Detail Screen



This screen shows the details of the tutorial/explanation. The FAB will start the alternate code editor that has the tutorial linked into it. The triangle is supposed to be a play button (I couldn’t figure out how to rotate the thing).

Code Editor (Tutorial Integrated Version)



```
.section text
    mov eax,1
    mov ebx,0
    int 80h
```

Exercise
Create a program that moves 1 into the EAX register.

```
Moved 1 into EAX
Moved 0 into EBX
Called Interrupt 80h
Exiting..
```

This is the alternate code editor view when the user wants to do an exercise defined in a tutorial.

Key Considerations

How will your app handle data persistence?

Most of the data will be user preferences. So this particular app will mostly be using SharedPreferences. For saving/loading programs, it will be using a basic file explorer and use an output stream to move the contents of the program into a file.

Describe any corner cases in the UX.

If the user wants to leave the alternate code editor linked to the corresponding tutorial, there is a clearly marked button for it.

Describe any libraries you'll be using and share your reasoning for including them.

This particular app doesn't need any libraries apart from the standard Material Design/Android libraries. Nothing crazy is being used here apart from my own Java code.

Next Steps: Required Tasks

This is the section where you can take the main features of your app (declared above) and decompose them into tangible technical tasks that you can complete incrementally until you have a finished app.

Task 1: Project Setup

- Configure libraries (Material design)
- Determine what the most suitable minSDK should be to target the most users but allow for gorgeous UI.
- Import pre-written Java code to form the base functionality of the app.

Task 2: Implement UI for Each Activity and Fragment

- Build UI for CodeActivity
- Add fragments, tabs, and toolbar buttons to CodeActivity.
- Add basic syntax highlighting to the code textview.
- The fragments mentioned above are the CodeFragment, and OutputFragment.
- Design alternate OutputFragment layout in accordance to design mock for tutorial integration. These will change via a Fragment transaction to make the "tutorial linked" code editor.
- Build UI for RegisterActivity, MemoryActivity, and its corresponding fragments.
- Build UI for TutorialActivity and its fragment, and the TutorialDetailActivityFragment. This will replace TutorialActivityFragment in a Fragment transaction.

Task 3: Link Pre-Completed Simulation Code to Android

- Some of the code has already been finished, as it's been worked in outside of Android Studio in Eclipse to get essential functions working more easily. It was easier to do it this way in the context of this project.
- At this point in integration, the basic Bss/Data segments are functional.
- Link Bss/Data segments to the MemoryActivity. Use readBss() and readData() from the Memory class to achieve this. They both return a Map object, where the key is the address, and the value is the value sitting at that address.
- The text parser is almost complete, but it can't be linked yet due to...

Task 4: ..The Simulation Thread..*dun dun duuuun*

- This is a doozy. Because the ability of stepping through/over the code is an essential part of visualizing x86, this needs to be put in a separate thread. Gasp!
- In order to run the App and the Interpreter parallel to each other, the code has to be executed by the interpreter in a separate thread, and must be able to be controlled. By control, I mean it has to wait/continue on fixed intervals and on logical conditions. The thread is *not* stopped in a random fashion. I will be using wait() and notify() to achieve this.
- There is only one additional thread needed to make this all work, and will be done in accordance to the following resources:
  http://developer.android.com/training/multiple-threads/define-runnable.html
  http://developer.android.com/guide/components/processes-and-threads.html
  http://developer.android.com/training/multiple-threads/communicate-ui.html
  http://stackoverflow.com/questions/1036754/difference-between-wait-and-sleep
  http://www.javamex.com/tutorials/wait_notify_how_to.shtml

Task 5: Finish Building the Simulation Code

- Add an implementation of a basic EFLAGS register.
- Finish the Bss segment.
- Finish the Data segment.
- Implement simulated method/procedure calls.
- Add some system interrupts to the Instruction set.
- Add a decent selection of Instructions. Mov is already implemented. Add ADD, SUB, MUL, DIV, CALL, and some of the variations of the JMP instruction.

- Add the thread specific code to allow the interpreter to “step” through and “step over” the code.

Task 6: Link the CodeFragment UI to the Simulation code

- Link the step button to the newly formed step code in the Interpreter.
- Link the “step through” button to the newly formed code in the Interpreter.
- Link the Execute button to the Interpreter’s “load” method. The load method takes in an array list of strings, each string corresponding to a line. So, this will have to take the code in the text area and convert it to an array list. After this, invoke interpret() to actually begin executing the code.
- Add the functionality to the tutorial panel buttons to exit tutorial mode, give a hint, and check the code to see if it yields the right result.

Task 7: Implement Tutorials

- Write a method to modify OutputFragment to integrate the tutorial panel, and another method to switch it back. It needs to be able to transform into the tutorial integrated editor as defined in the above mock.
- Create an adapter and fill it with a new layout that corresponds to the correct tutorial.
- The FAB in the TutorialDetailFragment has to be able to 1) Switch the tab to the code editor, and 2) Execute the method defined in the first point to flip around the UI to the Tutorial integrated UI.
- Add the functionality to the tutorial panel buttons to exit tutorial mode, give a hint, and check the code to see if it yields the right result.

Task 8: Error cases

- Now that everything is linked up and ready to go. Test away! Make sure text entered by the user can’t break the parser or the interpreter.
- Test edge cases. Since this is a closely coupled x86 simulator, it also simulates each segment. Make sure that overflowing the stack doesn’t break anything, and vice versa with the heap.
- Test any other cases that have been thought of during development.
- The simulation has been coded to take error cases into consideration, but this is why every case should be tested to make sure it handles it gracefully. This is supposed to be a teaching/learning tool, not a headache and confusion inducing tool.

Task 7: Polish

- Make sure all the UI gracefully meets all Material Design guidelines.
- Make the app feel alive. Add animations where they may look nice and convey what the user is supposed to do/can do.

Add as many tasks as you need to complete your app.

Submission Instructions

1. After you’ve completed all the sections, download this document as a PDF [File → Download as PDF]
2. Create a new GitHub repo for the capstone. Name it “**Capstone Project**”
3. Add this document to your repo. Make sure it’s named “**Capstone_Stage1.pdf**”