SYSTEM PROGRAMMING: 8085 ASSEMBLER

An X-Window based Graphical User Interface for 8085 microprocessor code Assembly, Simulation and Debugging.

DEVELOPPED BY

BCSE-III (2016-17) A2 Group I

001410501044 Soumya Kanti Naskar
001410501057 Sahil Pandey
001410501066 Chinmoy S. Mahapatra
001410501071 Abhirup Mondal

OVERVIEW

The software includes a fully functional inbuilt text editor with syntax highlighting, automatic indents and unlimited undo/redo. The source code may to be typed here. Alternatively, the code may be prepared in any suitable external editor and the saved file opened by the application. The default extension for 8085 assembly source code is ".asm"; however other formats (e.g. ".txt") may also be used.

Basic file editor options are provided. Additionally for the simulator, the following:

- Object Code generation using a 2-pass Assembler, with an insiders perspective into the workings of the Symbol Table and Object code display that mirrors the structure of the source code, for a deeper insight into the Assembly process.
- Execution of the Assembled Code for quick testing, with a FULL 65536 bytes of available memory.
- Step-wise execution of Assembled Code for debugging, with register contents updated in real-time and filtered display of modified memory contents.
- Instant error reports to quickly locate and fix bugs.

We draw your attention to the available keyboard accelerators. These are available for ALL the simulator's functions, including the usual shortcuts for the text editor's functions, Assembly, Execution and Debug. Though these functions can also be accessed through the Menu bar and buttons in the Debug window, the performance of the software is significantly increased through using the keyboard based alternatives.

BUILD DEPENDENCIES

Required to compile:

- gtkmm-3.0-dev
- pangomm
- gtksourceviewmm-3.0-dev
- g++

To compile:

```
g++ -o simulator src/*.cpp src/*/*.cpp `pkg-config gtkmm-3.0 pango gtksourceview-3.0 --cflags --libs` -std=c++11
```

To execute:

```
./simulator
```

DATA FILES

- opcodes_table.txt - Contains the 8085 Instruction Set with OPCODES, MNEMONICS and instruction sizes
- assembler-gui.glade - XML layout file for the main GUI text-editor/simulator application window
- object-code.glade - XML layout file for GUI Object Code display window.
- symtab.glade - XML layout file for GUI Symbol Table display window
- debugger.glade - XML layout file for GUI Debug window.
- 8085.lang - XML file for Regex Syntax definition for 8085 Assembly Language Code
- jucse_1.xml - XML file for syntax highlighting color scheme

SOURCE FILE SUMMARIES

opcode.cpp and opcode.hpp

- Contains a helper class that is used to load the 8085 Instruction Set from file.
- The class performs multiple consistency checks.
- The OPCODE Table is built with the OPCODE's binary value, MNEMONIC and the size of the complete instruction. E.g. 'MOV A B' and 'INX H' are 1-byte instructions; 'ADI 20H' and 'MVI 31H' are 2-byte instructions; 'LXI H 2000H' and 'JNZ LOOP' are 3-byte instructions.

assembler.cpp and assembler.hpp

- A line of an assembly program consists of 3 parts.
  - The first field represents a label or symbol. We require that the label be terminated by a colon ':'. Unlike many readily available 8085 assemblers, it is possible to have only a label on a line.
  - The second field is the OPCODE of the instruction. This OPCODE must belong to the 8085 instruction set.
  - The remaining fields are the operands/arguments to the instruction. In many assemblers, for instructions like 'LXI H 2000H', 'LXI H' as a whole is treated as the OPCODE and only the '2000H' as the operand. This is because the 8085 instruction set treats it as such. Our assembler too takes this into account while generating the Object Code. However, internally while simulating, we treat both the register 'H' and the immediate value '2000H' as arguments.

- The Source Code is Assembled into Object Code with a 2-pass Assembly procedure.
  - Pass 1 :
    - ✓ Assigns Addresses to each line of the Source Program.
    - ✓ Stores the Addresses of Symbolic Labels in the Symbol Table.
    - ✓ Produces an Intermediate format, with each line stored separately in a vector of strings.
  - Pass 2 :
    - ✓ Converts the Opcode and Symbolic operands to their binary values, using the Opcode Table and Symbol Table.
    - ✓ Produces the Object Code in binary format.
    - ✓ Any invalid syntax (Symbol redefinition, Invalid OPCODE, etc.) are detected and displayed for quick localization.

string_cleaner.cpp and string_cleaner.hpp

- This source contains helper functions to process the input string of the Source Code line by line.
- Unlike other 8085 Assemblers that force a strict, restricted instruction format…

  LABEL: *<1 space - no more no less>* OPCODE *<1 space - no more no less>* OPERAND *<new line>*

  …our input may be free form. Our processing is able to handle an arbitrary number of whitespaces (spaces, tabs, newlines, etc.). Further, unlike other editors, we do not require commas ',' to separate operands. E.g. 'MOV A, B' is unnecessary. 'MOV A B' is sufficient.

simulator.cpp and simulator.hpp

- This is the primary class of the internal implementation. The simulator class itself contains an instance of assembler class. The assembler class in turn uses the opcode class and the string cleaner functions to generate the Object Code and Symbol Table.

- The simulator class contains 7 registers, each 8-bits in width, namely A, B, C, D, E, H and L. The Flags are not maintained as entries in a separate register as in the 8085 microprocessor, but as 5 separate Boolean variables.

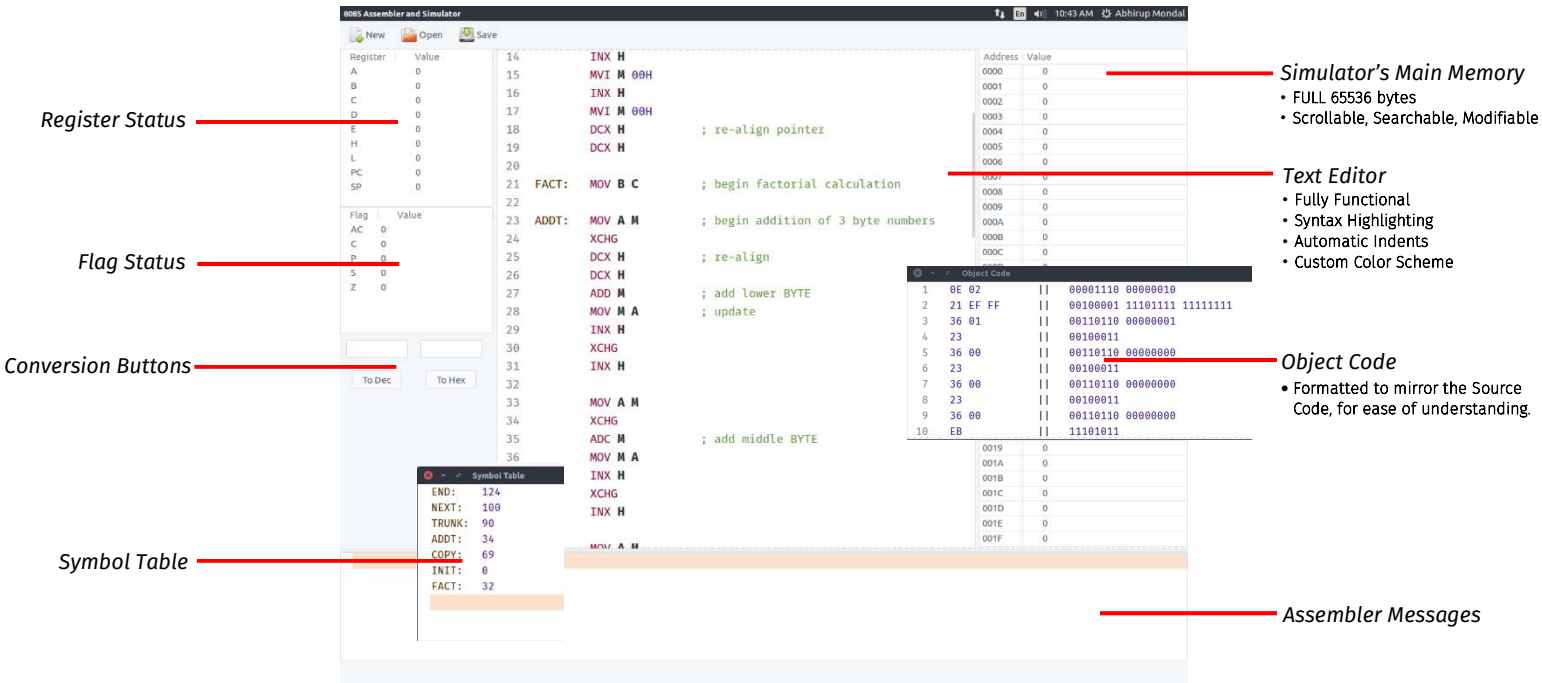
- The Object Code is not loaded into the simulator's main memory so as to provide the use with a FULL 65536 bytes of accessible, modifiable and observable memory. This can be used to provide input to the program. The memory view and the register/flags view together can be used to monitor the status of the executing Object Code. Coupled with the Debug features, this gives a strong sense of program flow.

- There are a total of 79 OPCODES in the 8085 Instruction Set, each of which has its own function, with careful attention paid to how flags are affected by different operations. This is vital as different instructions often work together in unexpected ways in real life programs to produce the desired result. Operations corresponding to port based input/output and interrupt handling have not been implemented; their usage in programs shall have no effect on the status of the simulator.

- To improve readability, these 79 functions have been segregated into source modules and relocated to a sub folder 'Operations'
  - arithmetic.cpp - addition, subtraction, increments/decrements
  - logical.cpp - AND, OR, XOR, rotations and other bitwise operations
  - control.cpp - jumps, calls, returns and stack pointer manipulations
  - movement.cpp - loads, stores and register to register moves

gui.cpp and gui.hpp

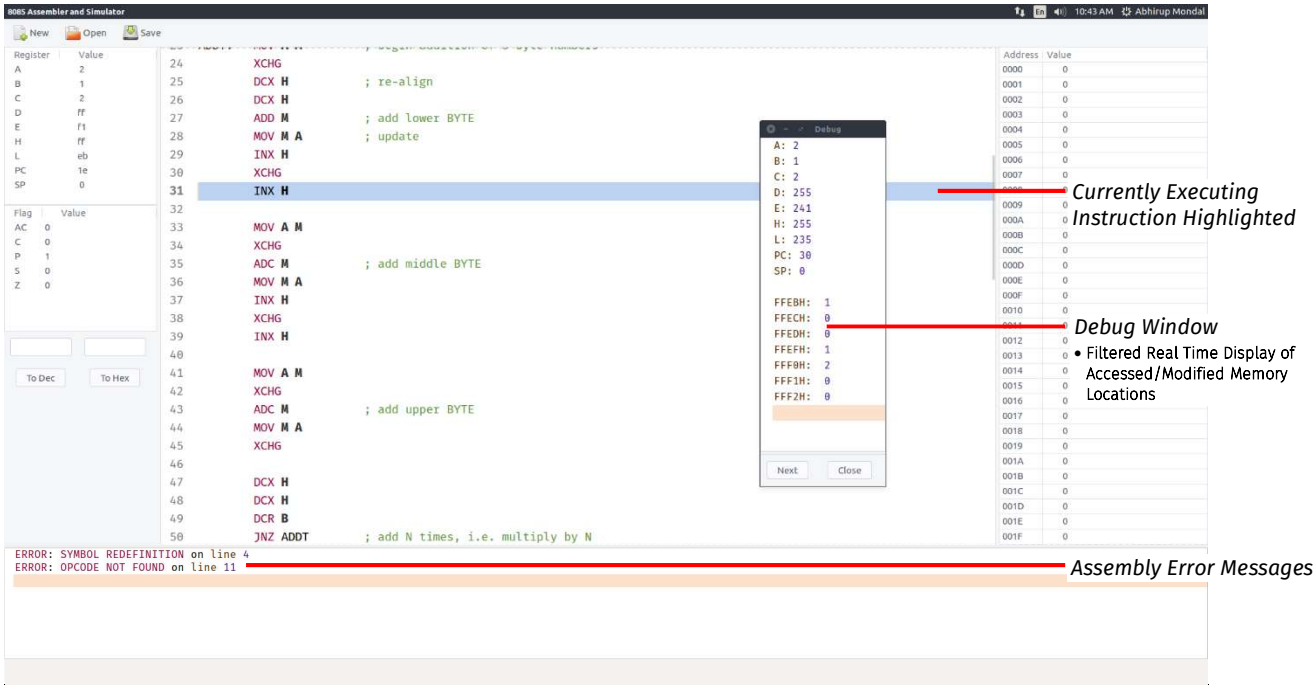
- This is the primary user interface source module. It contains the text-editor, register/flag view, memory view and error report facility. Additionally, the generated Object Code and Symbol Table can be accessed through the assembler object present in the local_simulator object.



- The text-editor provides all the basic functions of any editor, as well as advanced functions like Syntax Highlighting for 8085 Assembly Code, automatic indents and unlimited undo/redo. It also conspicuously highlights the current instruction during executed.

- All operations are controlled by the main GUI window. Calls to the Assemble and Execute functions from the Build menu percolate down to their respective counterparts in the assembler and simulator classes. Once Assembly has been completed, the Object Code and the Symbol Table can be viewed. This requires the creation of sub-GUI windows, handled by separate modules included in the main GUI module.
  - object.cpp and object.hpp
  - symtab.cpp and symtab.hpp

- The Execute option for the Build menu quickly simulates through the instruction of the program and updates the contents of both the Memory View and the Register/Flags View. If a more interactive interface is desired, the Next option from the Build menu executes instructions line by line, updating the displayed values after every operation. For debugging purposes the Debug option from the Build menu may be used. This opens the Debug sub-GUI window, handled by a separate module.
  - debug.cpp and debug.hpp



- In addition to executing instructions of the program line by line, Debug mode also provides a real time update of the status of registers and flags, as well as a filtered display of memory locations accessed/modified during the simulation. The currently executing instruction is also highlighted for ease of development.

- To improve readability, the GUI module has been segregated into source modules and relocated to a sub folder 'GUI'
  - actions.cpp - Contains the action handlers for the different menu options. Use of keyboard accelerators is recommended for improved performance.
  - setup.cpp - Initializes the main GUI window, loading the interface structure from XML files populating the space with the display objects. It also connects the different action signals to their respective handling functions.