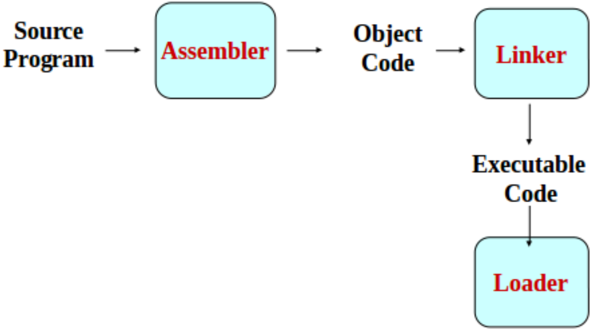
# SIC Assembler and Simulator


Hari Bhushan , Roll : 001410501006
Vineet Kumar , Roll : 001410501018
Sushant Gupta , Roll : 001410501027
Shubham Kumar Ranu , Roll: 001410501028

System Programming Lab
BCSE 3rd Year 2016
Jadavpur University
Kolkata


November 28, 2016

# Basic structure of ASSEMBLER



# Simplified Instructional Computer

- Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines. It has two models:
- Sic standard model
- SIC/XE (Extra Equipment) model.
- Upward compatible
  A program written in SIC should run on SIC/XE

# SIC machine Architecture

- Memory
  - 8-bit bytes
  - 3 consecutive bytes form a word, addressed by the lowest byte
  - Memory size is $2^{15} = (32768)$ bytes
- Registers : Total five registers / 24- bits each
  - A : Accumulator : 0 : used for arithmetic operations
  - X : Index register : 1 : used for addressing
  - L : Linkage register : 2 : the jump to subroutine (JSUB) instruction stores the return address in this register
  - PC :Program counter : 8 : contain the address of the next instruction to be fetched for execution
  - SW : Status word : 9 : contain a variety of information, including a condition code.
- Data Formats:
  - Integers are stored as 24-bit binary numbers; 2's complement representation is used for negative values
  - No floating-point hardware

# Machine architecture

- Addressing Modes

  `Mode  Indication Target address calculation`

  - Direct **X=0** TA = address
  - Indexed **X=1** TA= address + (X)

# Instruction Set

- **Instruction Set:**
- load and store: LDA, LDX, STA, STX, etc.
- integer arithmetic operations: ADD, SUB, MUL, DIV, etc.
- **All arithmetic operations involve register A and a word in memory, with the result being left in the register**
- comparison: COMP
- **COMP compares the value in register A with a word in memory, this instruction sets a condition code CC to indicate the result**
- conditional jump instructions: JLT, JEQ, JGT
- **these instructions test the setting of CC and jump accordingly**
- subroutine linkage: JSUB, RSUB
- JSUB jumps to the subroutine, placing the return address in register L
- RSUB returns by jumping to address contained in register L

# Instruction Set

- Input and Output: :
- Input and output are performed by transferring 1 byte at a time to or form the rightmost 8 bits of register A
- Each device is assigned a unique 8-bit code 8
- Three I/O instructions: 1- Test device (TD): 2- Read Data (RD) 3- Write Data (WD)
- **Data movement**
- No memory-memory move instruction
- 3-byte word: LDA, STA, LDL, STL, LDX, STX
- 1-byte: LDCH, STCH
- Storage definition
  – WORD, RESW
  – BYTE, RESB

# Data Structures Used

OPTABLE - It stores the mapping between mnemonic and machine code.
The class diagram for Optable class is

| Class : Optable |
|---|
| • static private Hashtable<String,Integer> |
| • static public int getOpCode(String key)<br>• static void populate() |

# Data Structures Used

SYMTAB - It stores the label name and the value(address) for each label.
The class diagram for SymTab class is

| **Class: Symtab** |
|---|
| • private Hashtable<String,Integer> |
| • public SymTab()<br>• public void putVal(String name,int address)<br>• public int getAddr(String name)<br>• public String toString() |

# Development of Assembler

The class diagram for Assembler class is

Class : Assembler:

- private SymTab stab;
- private String program,name;
- private String[] lines;
- private String[] words;
- private int addrs[],last_addr;
- private ObjectProg myObj;

# Support for Assembler Class

`Pass1(): void`

It creates intermediate file and help in solving the problem of forward Refrence

Its output are Symbol table

`Pass2():void:`

It creates the final object code file or we say machine code

Its output is object code

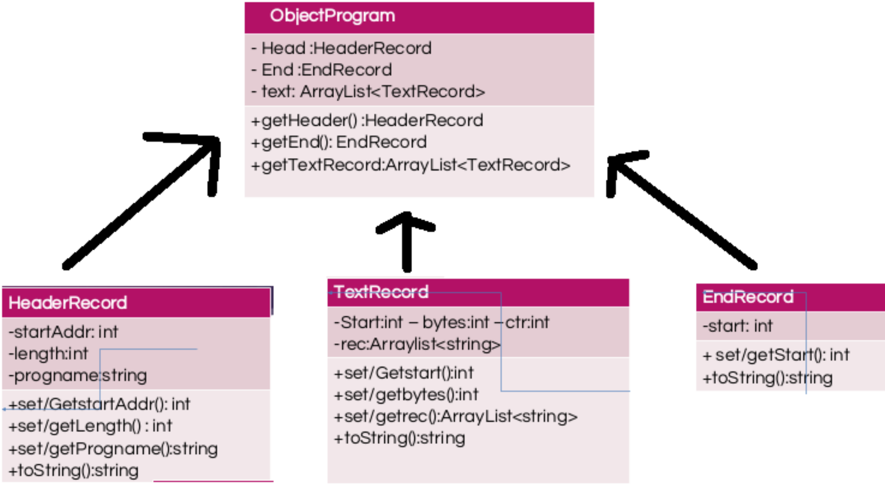
`assemble():void`

It helps in reading the program line by line and updates the location counter

# Object Program

- The object program is stored by the class ObjectProg.
- The ObjectProg uses the following classes to store the object program :
  - HeaderRecord : It contains the program name, starting address and length.
  - TextRecord : It contains the machine code instructions and data of the program, together with the indication where these are loaded.
  - EndRecord : It specifies the address in the program where execution is to begin.
- The class diagram showing the architecture of these classes is shown in the next page.

# Class diagram for object program



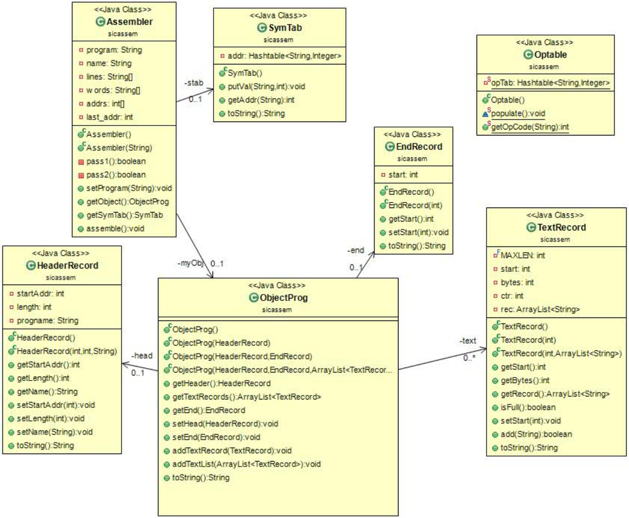



Figure: Class diagram

# Simulator

A simulator is a program enabling a computer to execute programs written for a different operating system.
It loads the program in the memory and executes the instructions


The class diagram is shown

**ExecEngine/simulator**

-reg_A:int
-reg_L:int
-reg_PC:int
-reg_X:int
-strt , end:int
-memory:Hashtable<Integer,Integer>
- myObj:ObjectProg

+set/getRegisters():void/int
+Init():void
+set/getMemory():void/int
+exec(): void
+perform(int opc,int addr):void

# GUI

GUI implemented class has output as : **symbol table**,**object code**,**register contents** and **memory dump**.
The UI class has the `static void main()` as well as the following classes :

- `panel`
- `actlist`
- `savelist`
- `MymenuBar`

# Final Product

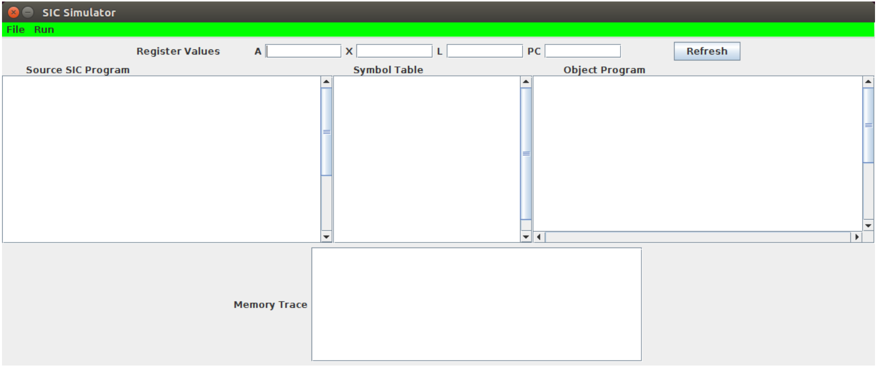



Figure: Final GUI Version of SIC Simulator

# Simulator in Action

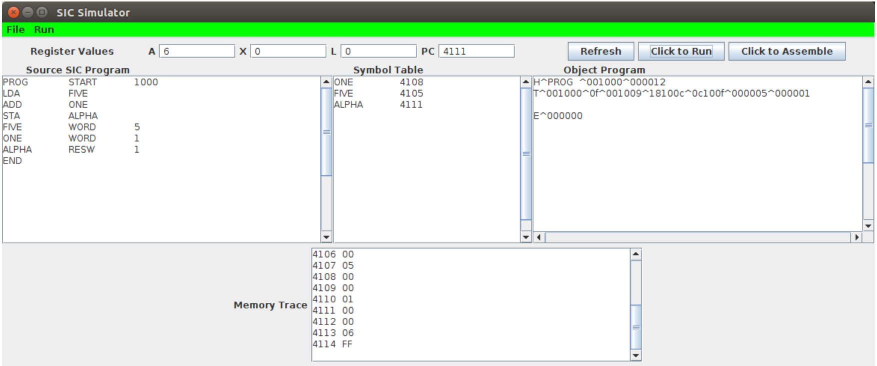



Figure: Final GUI Version of SIC Simulator simulating addition of TWO numbers

# Prerequisite

You need to have JAVA installed in your machine. This is the easiest version. First update the package index.

```
sudo apt-get update
```

Then check if java is intalled or not.

```
java -version
```

If it returns “The program java can be found in the following packages”, Java hasn’t been installed yet, so execute the following command:

```
sudo apt-get install default-jre
sudo apt-get install default-jdk
```

That is everything you need to install Java.

# Run the Simulator

First get into the folder SIC-Assembler-Simulator :

`cd SIC-Assembler-Simulator/`

Now compile the program :

`javac UI.java`

Run the compiled program :

`java UI`

You will notice a dialog appear like shown above.

Enter your SIC code in the `Source SIC Program` box.

Then click on `Click to Assemble` button to generate the object program as shown below :

# Run the Simulator

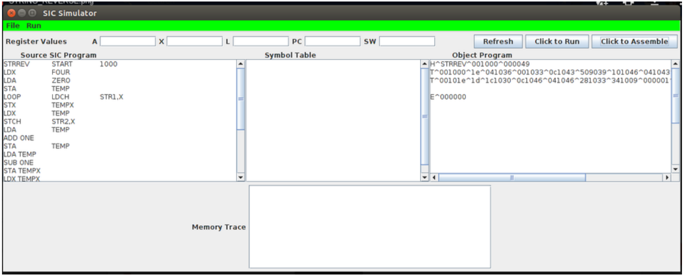
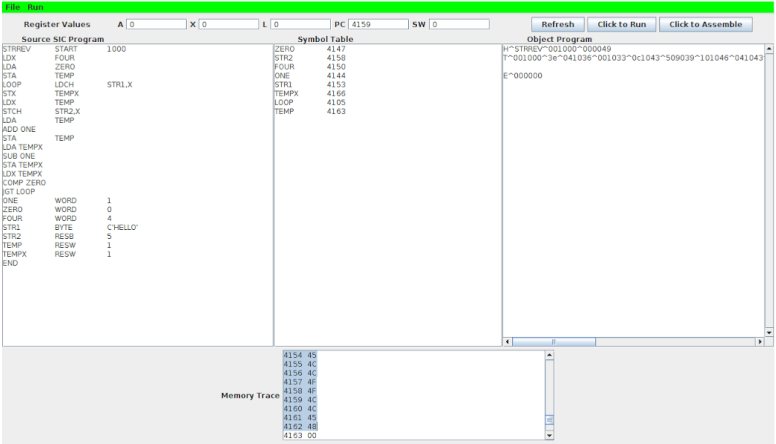



Figure: Generating the object program

# Run the Simulator

Now click on `Click to Run` button. You will get something like below as output if your SIC program is correct :



# Additional Information

- `Register Values` gives the values stored in different registers.
- `Symbol Table` gives us addresses assigned to the labels.
- `Object Program` gives us the object program for the SIC code.
- `Memory Trace` shows the value occupied in different memory locations.

# File Architecture

The file architecture for the project is shown below :

```
.
├── ExecEngine.class
├── ExecEngine.java
├── logo.jpg
├── MyMenuBar.class
├── panel$1.class
├── panel$2.class
├── panel$3.class
├── panel$4.class
├── panel$5.class
├── panel$actlist.class
├── panel.class
├── panel$savelist.class
├── Read_problem.pdf
├── sicassem
│   ├── Assembler.class
│   ├── Assembler.java
│   ├── EndRecord.class
│   ├── EndRecord.java
│   ├── HeaderRecord.class
│   ├── HeaderRecord.java
│   ├── ObjectProg.class
│   ├── ObjectProg.java
│   ├── Optable.class
│   ├── Optable.java
│   ├── SymTab.class
│   ├── SymTab.java
│   ├── TextRecord.class
│   └── TextRecord.java
├── Test.class
├── Test.java
├── UI.class
└── UI.java
```

We can see that the `sicassem` folder contains the codes for the assembler while the root folder has codes for the graphic user interface.