Claims

Name: P. V. Sriram

Roll No.: 1801CS37

Assembler

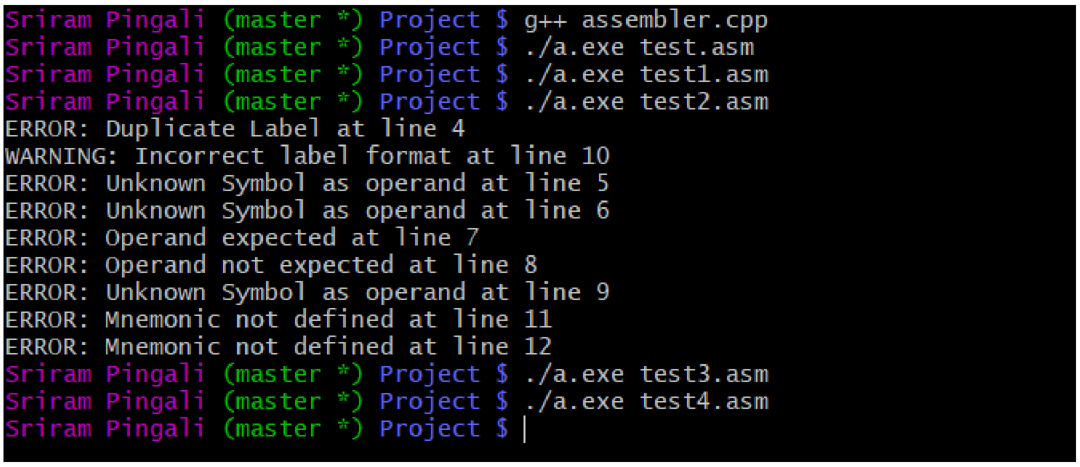
The project contains the assembler code. This is a 2-pass assembler which follows the SIMPLE instruction format.

To Compile: g++ assembler.cpp -o asm

To Run: ./asm.exe test1.o

Compilation

We can see below that assembler.cpp compiles without any error or warning. We also execute the test assembler files and there are errors in the code of test3.asm.

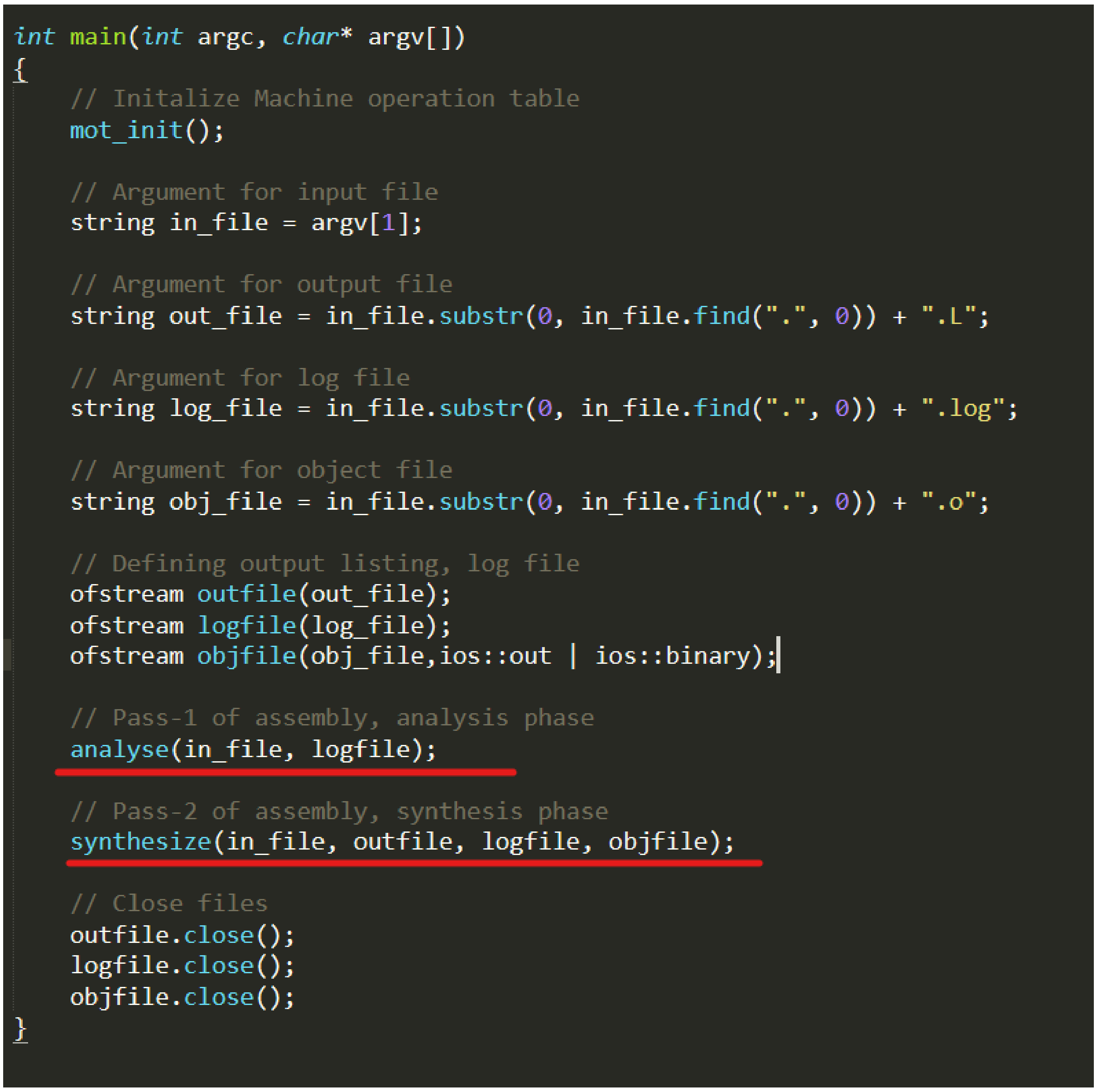


```
Sriram Pingali (master *) Project $ g++ assembler.cpp
Sriram Pingali (master *) Project $ ./a.exe test.asm
Sriram Pingali (master *) Project $ ./a.exe test1.asm
Sriram Pingali (master *) Project $ ./a.exe test2.asm
ERROR: Duplicate Label at line 4
WARNING: Incorrect label format at line 10
ERROR: Unknown Symbol as operand at line 5
ERROR: Unknown Symbol as operand at line 6
ERROR: Operand expected at line 7
ERROR: Operand not expected at line 8
ERROR: Unknown Symbol as operand at line 9
ERROR: Mnemonic not defined at line 11
ERROR: Mnemonic not defined at line 12
Sriram Pingali (master *) Project $ ./a.exe test3.asm
Sriram Pingali (master *) Project $ ./a.exe test4.asm
Sriram Pingali (master *) Project $ |
```

2-Pass System

This Assembler is a 2-Pass system wherein the first pass is an analyzer, to detect the symbols and literals of the code. And the second pass is a synthesizer generates the machine code for the assembler files.

We do it in two steps here where the first pass happens through the function analyse(), which takes arguments as file pointer for input file as “in_file” and log file pointer as “log_file”. The second pass happens through the function synthesize (), which takes arguments as file pointer of input file as “in_file”, listing file as “outfile”, log file as “logfile” and object file as “objfile”.



```
int main(int argc, char* argv[])
{
    // Initalize Machine operation table
    mot_init();

    // Argument for input file
    string in_file = argv[1];

    // Argument for output file
    string out_file = in_file.substr(0, in_file.find(".", 0)) + ".L";

    // Argument for log file
    string log_file = in_file.substr(0, in_file.find(".", 0)) + ".log";

    // Argument for object file
    string obj_file = in_file.substr(0, in_file.find(".", 0)) + ".o";

    // Defining output listing, log file
    ofstream outfile(out_file);
    ofstream logfile(log_file);
    ofstream objfile(obj_file,ios::out | ios::binary);

    // Pass-1 of assembly, analysis phase
    analyse(in_file, logfile);

    // Pass-2 of assembly, synthesis phase
    synthesize(in_file, outfile, logfile, objfile);

    // Close files
    outfile.close();
    logfile.close();
    objfile.close();
}
```

Error Detection

Below is the log file of Test2.asm and all the errors are identified.

```
ERROR: Duplicate Label at line 4
WARNING: Incorrect label format at line 10
ERROR: Unknown Symbol as operand at line 5
ERROR: Unknown Symbol as operand at line 6
ERROR: Operand expected at line 7
ERROR: Operand not expected at line 8
ERROR: Unknown Symbol as operand at line 9
ERROR: Mnemonic not defined at line 11
ERROR: Mnemonic not defined at line 12
```

Listing File Generation

Listing file shows the program counter, machine code and instruction. Here are the lists generated from the test programs.

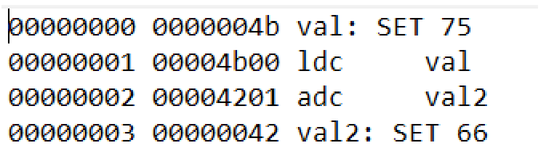
Test1.L

```
00000000          label:
00000000 00000000 ldc 0
00000001 fffffb00 ldc -5
00000002 00000500 ldc +5
00000003 ffffff11 loop: br loop
00000004 00000011 br next
00000005          next:
00000005 00000300 ldc loop
00000006 00000700 ldc var1
00000007 00000000 var1: data 0
```

Test2.L

```
00000000          label:
00000000          label:
00000000 br nonesuch
00000001 ldc 08ge
00000002 ldc
00000003 00000006 add 5
00000004 ldc 5, 6
00000005          0def:
00000005 fibble
00000006 0def
```

Test3.L



```
00000000 0000004b val: SET 75
00000001 00004b00 ldc     val
00000002 00004201 adc     val2
00000003 00000042 val2: SET 66
```

Test4.L

```
00000000 00100000 ldc 0x1000

00000001 0000000b a2sp

00000002 ffffff0a adj -1

00000003 00004b00 ldc result

00000004 00000003 stl 0

00000005 00004a00 ldc count

00000006 00000004 ldnl 0

00000007 0000020d call main

00000008 0000010a adj 1

00000009 00000012 HALT

0000000a fffffd0a main:   adj -3

0000000b 00000103 stl 1

0000000c 00000203 stl 2

0000000d 00000000 ldc 0

0000000e 00000003 stl 0

0000000f ffffff0a loop:   adj -1

00000010 00000302 ldl 3

00000011 00000003 stl 0

00000012 00000102 ldl 1

00000013 0000100d call triangle

00000014 0000010a adj 1

00000015 00000302 ldl 3

00000016 00000005 stnl 0

00000017 00000302 ldl 3
```

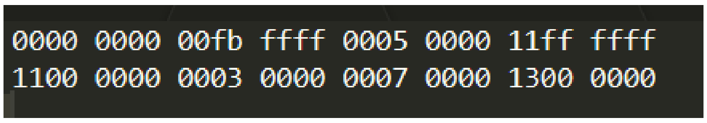
```
00000018 00000101 adc 1

00000019 00000303 stl 3

0000001a 00000002 ldl 0

0000001b 00000101 adc 1

0000001c 00000003 stl 0

0000001d 00000002 ldl 0

0000001e 00000202 ldl 2

0000001f 00000007 sub

00000020 ffffee10 brlz loop

00000021 00000102 ldl 1

00000022 0000030a adj 3

00000023 0000000e return

00000024 fffffd0a triangle:adj -3

00000025 00000103 stl 1

00000026 00000203 stl 2

00000027 00000100 ldc 1

00000028 00000008 shl

00000029 00000302 ldl 3

0000002a 00000007 sub

0000002b 00000410 brlz skip

0000002c 00000302 ldl 3

0000002d 00000202 ldl 2

0000002e 00000007 sub

0000002f 00000203 stl 2

00000030 00000202 skip:   ldl 2

00000031 0000140f brz one

00000032 00000302 ldl 3

00000033 ffffff01 adc -1

00000034 00000003 stl 0
```

```
00000035 ffffff0a adj -1

00000036 00000102 ldl 1

00000037 00000003 stl 0

00000038 00000302 ldl 3

00000039 ffffff01 adc -1

0000003a ffffe90d call triangle

0000003b 00000102 ldl 1

0000003c 00000003 stl 0

0000003d 00000103 stl 1

0000003e 00000302 ldl 3

0000003f ffffe40d call triangle

00000040 0000010a adj 1

00000041 00000002 ldl 0

00000042 00000006 add

00000043 00000102 ldl 1

00000044 0000030a adj 3

00000045 0000000e return

00000046 00000100 one:    ldc 1

00000047 00000102 ldl 1

00000048 0000030a adj 3

00000049 0000000e return

0000004a 0000000a count:  data 10

0000004b 00000000 result: data 0
```

Object File Generation

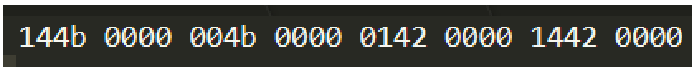
Object file compiles the machine code of each program into a single encoded format. We use the fstream binary format to encode the integer representation of each instructions machine code into unsigned 32-bit format. We open the object file in the hex format and below are the contents.

Test1.o



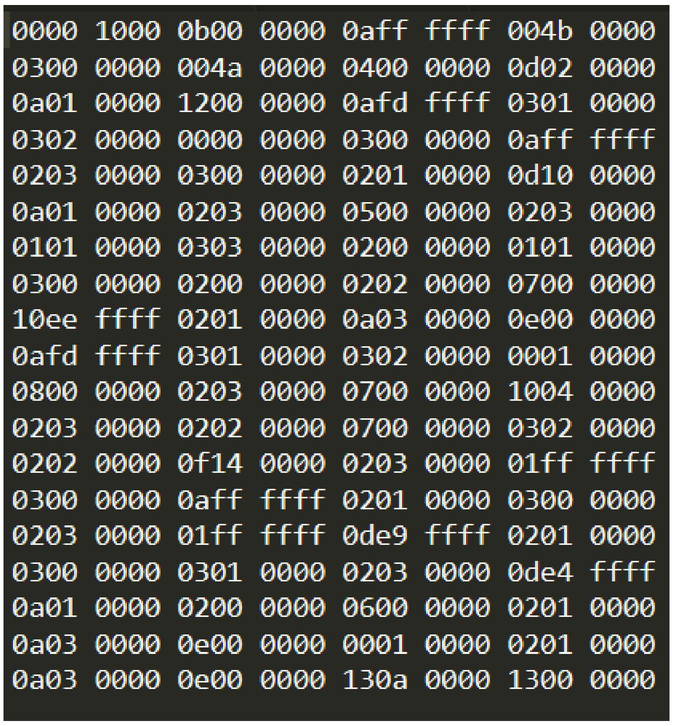
```
0000 0000 00fb ffff 0005 0000 11ff ffff
1100 0000 0003 0000 0007 0000 1300 0000
```

Test3.o



```
144b 0000 004b 0000 0142 0000 1442 0000
```

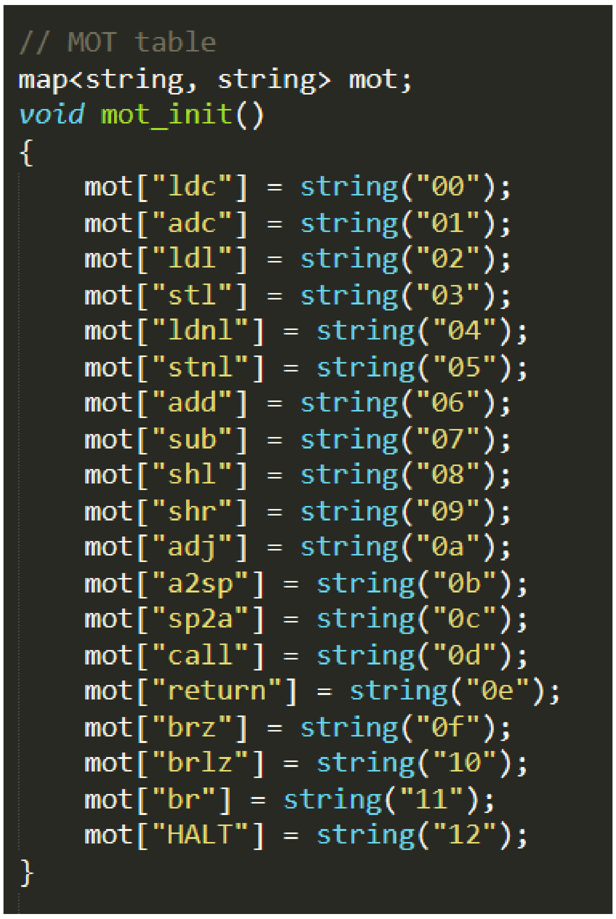
Test4.o



```
0000 1000 0b00 0000 0aff ffff 004b 0000
0300 0000 004a 0000 0400 0000 0d02 0000
0a01 0000 1200 0000 0afd ffff 0301 0000
0302 0000 0000 0000 0300 0000 0aff ffff
0203 0000 0300 0000 0201 0000 0d10 0000
0a01 0000 0203 0000 0500 0000 0203 0000
0101 0000 0303 0000 0200 0000 0101 0000
0300 0000 0200 0000 0202 0000 0700 0000
10ee ffff 0201 0000 0a03 0000 0e00 0000
0afd ffff 0301 0000 0302 0000 0001 0000
0800 0000 0203 0000 0700 0000 1004 0000
0203 0000 0202 0000 0700 0000 0302 0000
0202 0000 0f14 0000 0203 0000 01ff ffff
0300 0000 0aff ffff 0201 0000 0300 0000
0203 0000 01ff ffff 0de9 ffff 0201 0000
0300 0000 0301 0000 0203 0000 0de4 ffff
0a01 0000 0200 0000 0600 0000 0201 0000
0a03 0000 0e00 0000 0001 0000 0201 0000
0a03 0000 0e00 0000 130a 0000 1300 0000
```

Machine Opcode Table

We use a hash map to map the mnemonics to their opcodes. We initialize the table in the following manner.



```
// MOT table
map<string, string> mot;
void mot_init()
{
    mot["ldc"] = string("00");
    mot["adc"] = string("01");
    mot["ldl"] = string("02");
    mot["stl"] = string("03");
    mot["ldnl"] = string("04");
    mot["stnl"] = string("05");
    mot["add"] = string("06");
    mot["sub"] = string("07");
    mot["shl"] = string("08");
    mot["shr"] = string("09");
    mot["adj"] = string("0a");
    mot["a2sp"] = string("0b");
    mot["sp2a"] = string("0c");
    mot["call"] = string("0d");
    mot["return"] = string("0e");
    mot["brz"] = string("0f");
    mot["brlz"] = string("10");
    mot["br"] = string("11");
    mot["HALT"] = string("12");
}
```

The map is based on the following instructions given in the Question pdf.

The Instructions

The instruction semantics do not show the incrementing of the PC to the next instruction. This is implicitly performed by each instruction *before* the actions of the instruction are done.

Mnemonic	Opcode	Operand	Formal Specification	Description
data		value		Reserve a memory location, initialized to the value specified
ldc	0	value	B := A; A := value;	Load accumulator with the value specified
adc	1	value	A := A + value;	Add the value specified to the accumulator

ldl	2	offset	B := A;
			A := memory[SP + offset];
				Load local

stl	3	offset	memory[SP + offset] := A;
			A := B;
				Store local

ldnl	4	offset	A := memory[A + offset];	Load non-local
stnl	5	offset	memory[A + offset] := B;	Store non-local
add	6		A := B + A;	Addition
sub	7		A := B - A;	Subtraction
shl	8		A := B << A;	Shift left
shr	9		A := B >> A;	Shift right
adj	10	value	SP := SP + value;	Adjust SP
a2sp	11		SP := A; A := B	Transfer A to SP;
sp2a	12		B := A; A := SP;	Transfer SP to A

call	13	offset	B := A; A := PC;
			PC := PC + offset;
				Call procedure

return	14		PC := A; A := B;	Return from procedure

brz	15	offset	if A == 0 then
			PC := PC + offset;
				If accumulator is zero, branch to specified offset

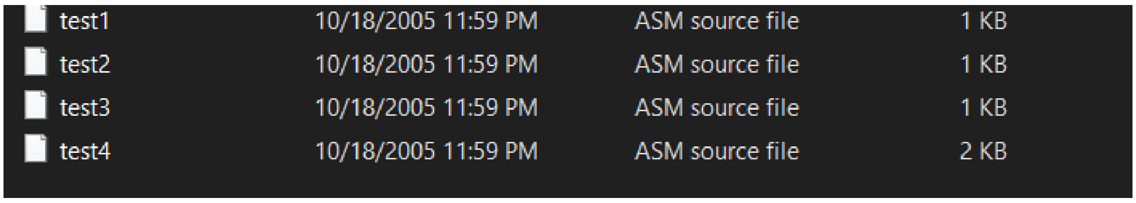
brlz	16	offset	if A < 0 then
			PC := PC + offset;
				If accumulator is less than zero, branch to specified offset

br	17	offset	PC := PC + offset;	Branch to specified offset
HALT	18			Stop the emulator. This is not a `real' instruction, but needed to tell your emulator when to finish.
SET		value		Set the label on this line to the specified value (rather than the PC). This is an optional extension, for which additional marks are available.

Test Programs

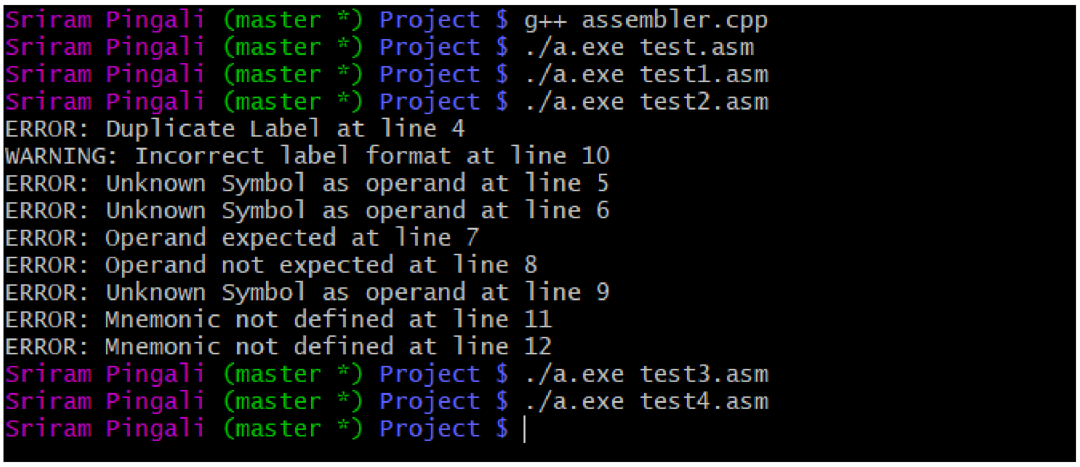
We run the assembler on the given test files. i.e. Test1.asm, Test2.asm, Test3.asm, Test4.asm.

Below is the circumstance before running the asm executable file on the test programs



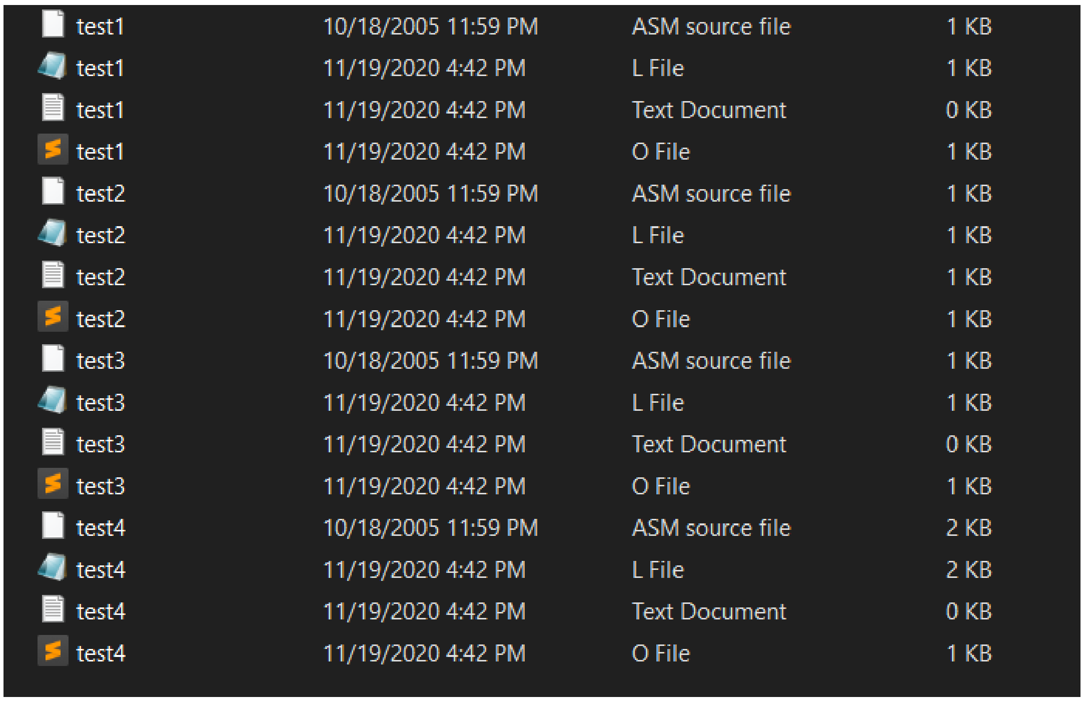
test1	10/18/2005 11:59 PM	ASM source file	1 KB
test2	10/18/2005 11:59 PM	ASM source file	1 KB
test3	10/18/2005 11:59 PM	ASM source file	1 KB
test4	10/18/2005 11:59 PM	ASM source file	2 KB

Below is the screen shot of the terminal which shows that the assembling is done successfully. We can also see that test2.asm contains errors which are shown in the terminal as well as the corresponding log file.



```
Sriram Pingali (master *) Project $ g++ assembler.cpp
Sriram Pingali (master *) Project $ ./a.exe test.asm
Sriram Pingali (master *) Project $ ./a.exe test1.asm
Sriram Pingali (master *) Project $ ./a.exe test2.asm
ERROR: Duplicate Label at line 4
WARNING: Incorrect label format at line 10
ERROR: Unknown Symbol as operand at line 5
ERROR: Unknown Symbol as operand at line 6
ERROR: Operand expected at line 7
ERROR: Operand not expected at line 8
ERROR: Unknown Symbol as operand at line 9
ERROR: Mnemonic not defined at line 11
ERROR: Mnemonic not defined at line 12
Sriram Pingali (master *) Project $ ./a.exe test3.asm
Sriram Pingali (master *) Project $ ./a.exe test4.asm
Sriram Pingali (master *) Project $ |
```

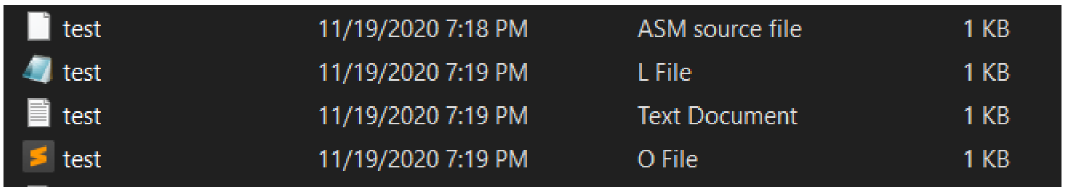
After the assembling of all the files, we can see that the corresponding list(.L), log(.log, text file) and object(.o) files are generated.



test1	10/18/2005 11:59 PM	ASM source file	1 KB
test1	11/19/2020 4:42 PM	L File	1 KB
test1	11/19/2020 4:42 PM	Text Document	0 KB
test1	11/19/2020 4:42 PM	O File	1 KB
test2	10/18/2005 11:59 PM	ASM source file	1 KB
test2	11/19/2020 4:42 PM	L File	1 KB
test2	11/19/2020 4:42 PM	Text Document	1 KB
test2	11/19/2020 4:42 PM	O File	1 KB
test3	10/18/2005 11:59 PM	ASM source file	1 KB
test3	11/19/2020 4:42 PM	L File	1 KB
test3	11/19/2020 4:42 PM	Text Document	0 KB
test3	11/19/2020 4:42 PM	O File	1 KB
test4	10/18/2005 11:59 PM	ASM source file	2 KB
test4	11/19/2020 4:42 PM	L File	2 KB
test4	11/19/2020 4:42 PM	Text Document	0 KB
test4	11/19/2020 4:42 PM	O File	1 KB

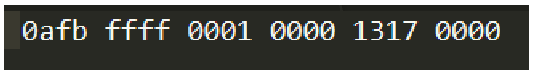
I've also written a failing script called test.asm and executed the assembler on it, the following are the results.

Folder



test	11/19/2020 7:18 PM	ASM source file	1 KB
test	11/19/2020 7:19 PM	L File	1 KB
test	11/19/2020 7:19 PM	Text Document	1 KB
test	11/19/2020 7:19 PM	O File	1 KB

Object File



```
0afb ffff 0001 0000 1317 0000
```

Error file

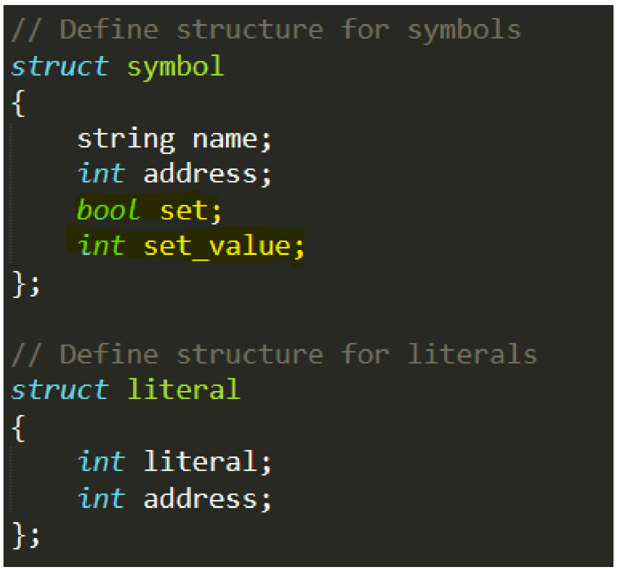
```
ERROR: Unknown Symbol as operand at line 2
```

Listing file

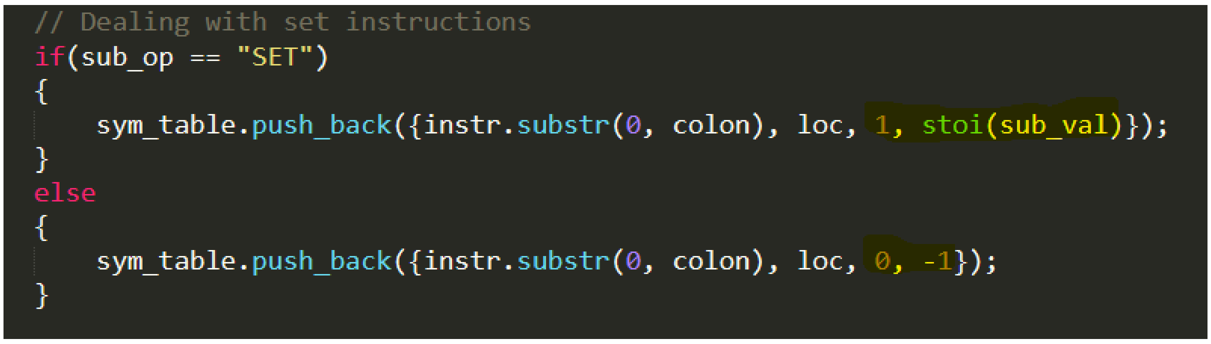
```
00000000          bogus:
00000000 ldc bogus_1
00000001 fffffb0a var1:adj -5
00000002 00000100 ldc var1
00000003          var2:
00000003 00001713 result: data 23
```

SET Assembler directive

I have incorporated the SET instructions in the following manner. In the symbol structure, we incorporate the set (bool) which indicates if the symbol was defined as a SET. And also its value. This process happens in the first half.

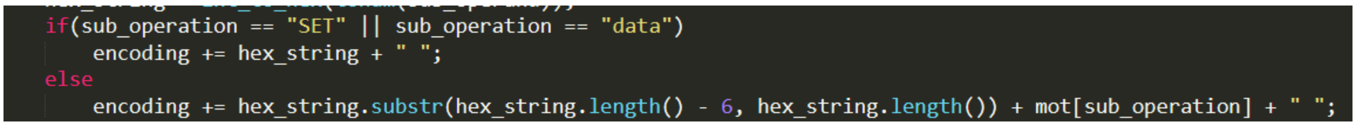


```
// Define structure for symbols
struct symbol
{
    string name;
    int address;
    bool set;
    int set_value;
};

// Define structure for literals
struct literal
{
    int literal;
    int address;
};
```



```
 // Dealing with set instructions
 if(sub_op == "SET")
 {
     sym_table.push_back({instr.substr(0, colon), loc, 1, stoi(sub_val)});
 }
 else
 {
     sym_table.push_back({instr.substr(0, colon), loc, 0, -1});
 }
```

In 2nd pass, if we face a set instruction, we just take the operand and encode it in 8 bits.



```
    if(sub_operation == "SET" || sub_operation == "data")
        encoding += hex_string + " ";
    else
        encoding += hex_string.substr(hex_string.length() - 6, hex_string.length()) + mot[sub_operation] + " ";
```

Bubble Sort

Following are the results from assembling the bubble sort program

Listing

```
00000000 00100000 ldc 0x1000

00000001 0000000b a2sp

00000002 ffffff0a adj -1

00000003 00000a00 ldc 10

00000004 00000003 stl 0

00000005 00004100 ldc array

00000006 0000010d call sort

00000007 00000012 HALT

00000008 ffffff0a sort: adj -1

00000009 00000003 stl 0

0000000a 00000203 stl 2

0000000b fffffd0a adj -3

0000000c 00000000 ldc 0

0000000d 00000003 stl 0

0000000e 00000100 ldc 1

0000000f 00000103 stl 1

00000010 00000000 ldc 0

00000011 00000203 stl 2

00000012 00000402 loop_out: ldl 4

00000013 00000202 ldl 2

00000014 00000007 sub

00000015 00000100 ldc 1

00000016 00000007 sub

00000017 0000260f brz done

00000018 00000100 ldc 1
```

```
00000019 00000103 stl 1
0000001a 00000402 loop_in: ldl 4
0000001b 00000202 ldl 2
0000001c 00000007 sub
0000001d 00000102 ldl 1
0000001e 00000007 sub
0000001f 0000160f brz addx
00000020 00000502 ldl 5
00000021 00000004 ldnl 0
00000022 00000000 ldc 0
00000023 00000104 ldnl 1
00000024 00000007 sub
00000025 00000110 brlz swap
00000026 00000a11 br addy
00000027 00000502 swap: ldl 5
00000028 00000004 ldnl 0
00000029 00000003 stl 0
0000002a 00000502 ldl 5
0000002b 00000104 ldnl 1
0000002c 00000502 ldl 5
0000002d 00000005 stnl 0
0000002e 00000002 ldl 0
0000002f 00000502 ldl 5
00000030 00000105 stnl 1
00000031 00000100 addy: ldc 1
00000032 00000102 ldl 1
00000033 00000006 add
00000034 00000103 stl 1
00000035 ffffe411 br loop_in
```

```
00000036 00000100 addx: ldc 1

00000037 00000202 ldl 2

00000038 00000006 add

00000039 00000203 stl 2

0000003a 00000100 ldc 1

0000003b 00000502 ldl 5

0000003c 00000006 add

0000003d ffffd411 br loop_out

0000003e 00000302 done: ldl 3

0000003f 0000050a adj 5

00000040 0000000e return

00000041 deadbeef array: data -559038737

00000042 5eedbed5 data 1592639189

00000043 c0edbabe data -1058161986

00000044 5eaf00d5 data 1588527317

00000045 ab5c155a data -1420028582

00000046 ca5caded data -899895827

00000047 feedface data -17958194

00000048 c0c0f00d data -1061097459

00000049 decea5ed data -556882451

0000004a 50fabed5 data 1358610133
```

Object File

```
0000 1000 0b00 0000 0aff ffff 000a 0000

0300 0000 0041 0000 0d01 0000 1200 0000

0aff ffff 0300 0000 0302 0000 0afd ffff

0000 0000 0300 0000 0001 0000 0301 0000

0000 0000 0302 0000 0204 0000 0202 0000

0700 0000 0001 0000 0700 0000 0f26 0000
```

```
0001 0000 0301 0000 0204 0000 0202 0000

0700 0000 0201 0000 0700 0000 0f16 0000

0205 0000 0400 0000 0000 0000 0401 0000

0700 0000 1001 0000 110a 0000 0205 0000

0400 0000 0300 0000 0205 0000 0401 0000

0205 0000 0500 0000 0200 0000 0205 0000

0501 0000 0001 0000 0201 0000 0600 0000

0301 0000 11e4 ffff 0001 0000 0202 0000

0600 0000 0302 0000 0001 0000 0205 0000

0600 0000 11d4 ffff 0203 0000 0a05 0000

0e00 0000 13ef bead 13d5 beed 13be baed

13d5 00af 135a 155c 13ed ad5c 13ce faed

130d f0c0 13ed a5ce 13d5 befa
```