
COMPILER DESIGN PROJECT REPORT

TEAM - 11

CS18B004 Akshitha
CS18B009 Deepanvi Penmetcha
CS18B013 G Nikhitha Vedi
CS18B031 S Bhavana



INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

TABLE OF CONTENTS

INTRODUCTION.....	3
TOOLS AND IMPLEMENTATION CHOICES.....	3
COMPILER's DESIGN AND MAJOR COMPONENTS.....	3
Lexer.....	4
Parser.....	4
Grammar.....	4
Abstract Syntax Tree (AST).....	5
Intermediate Code Generation (ICG).....	6
Code Generation (CG).....	6
Symbol Table.....	6
DESIGN DIAGRAM.....	7
SOURCE CODE ORGANIZATION.....	7
Folder and files.....	7
Build and generate a compiler executable.....	8
TESTING AND EVALUATION OF THE COMPILER.....	9
Test 1.....	9
Test 2.....	10
Test 3.....	12
Test 4.....	13
Test 5.....	15
LIMITATIONS.....	16
CONCLUSION.....	16
CONTRIBUTIONS.....	17

INTRODUCTION

The aim of this project is to **design a compiler for a custom programming language**¹. The compiler takes a source code file as an input and generates an assembly code for the same. The compiler has the general compiler phases and modules like the lexical analysis, parsing (syntax and semantic analysis), symbol table, intermediate code generation and code generation. This report focuses on the design, implementation and major design decisions taken, along with source code details, test cases and limitations of the compiler.

TOOLS AND IMPLEMENTATION CHOICES

For writing the compiler the following tools and languages are used

1. **Flex** for lexical analysis
2. **Yacc** to write a grammar that facilitates a syntax driven abstract syntax tree generation during parsing
3. **CPP** and GNU cpp compiler for writing the remaining modules
4. **MIPs assembly** is the target assembly language for the compiler
5. The final mips assembly file is run using **spim** from terminal or QtSpim can also be used

Most of the implementations are **object oriented** with usage of OOP concepts like use of classes, inheritance etc., through cpp.

COMPILER'S DESIGN AND MAJOR COMPONENTS

The compiler has the following components

1. Lexer
2. Parser
3. Abstract Syntax Tree (AST)
4. Intermediate Code Generation (ICG)
5. Code Generation (CG)
6. Symbol table

¹ Refer the language reference manual for the custom programming language - MinCode

A brief description of each of the components' design is as follows

Lexer

The lexer handles the lexical analysis phase. The parser calls the lexical analyzer for tokens as it reads through the source code. Flex is chosen to write the lexer because it allows identification of tokens through **regular expression matching mechanism**. And writing a bunch of regular expressions for a small programming language is a pretty easy task.

Parser

In this compiler, parser is a place for many functionalities and integration of various modules. The parser implementation involves writing a **context free grammar** with each grammar rule attached with appropriate actions. The actions involve **installation and retrieval of symbols** from the symbol table, **construction of the abstract syntax tree** with parallel semantic analysis happening and **creation of temporaries and labels** (for ICG and CG). Yacc is chosen because it offers a prebuilt **LALR parser** that takes a grammar for syntax analysis and also allows to write the actions for each grammar rule. The matching of grammar rules happens in a **bottom-up manner** which is really helpful in the construction of an abstract syntax tree.

Grammar

```

start → prog
prog → init { stmts }
stmts → stmt stmts | ε
stmt → int id ;
      | loc = expr ;
      | if ( condition ) { stmts }
      | if ( condition ) { stmts } else {stmts}
      | loop ( condition ) { stmts }
      | print ( loc ) ;
      | print ( strconst ) ;
      | get ( loc ) ;
condition → bool

```

```

bool → bool || bool
      | bool && bool
      | ! bool
      | ( bool )
      | expr == expr
      | expr != expr
      | expr > expr
      | expr >= expr
      | expr < expr
      | expr <= expr
      | true
      | false
expr → expr + expr
      | expr - expr
      | expr * expr
      | expr / expr
      | ( expr )
      | - expr
      | loc
      | intconst
loc  → id

```

Associativity and precedence of the operators is the same as in standard C language.

Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) is a byproduct of the parser. AST is a hierarchical structure of nodes partially connected to each other. Each non terminal in the above grammar has a class associated with it. The structure of nodes in the AST are written as **cpp classes of which objects** are created during the rule matching process in the parser within each grammar rule's action.

**For cpp classes description please refer to the code/ comments in the related files or the video.*

Intermediate Code Generation (ICG)

The classes corresponding to the nodes in ast have the intermediate code generation functions where the intermediate code generation rules are written and invoked **upon creation of a node object**. The bottom-up parser helps in the maintaining order of information collection and generation (like the identifier, the value of a variable, symbol table entry, generation of a temporary, the next statement or labels for statements in case of the control flow structures etc.,). Hence, it is easy to make sure that a node gets all the information it needs for its construction and thus it is possible to generate the intermediate code on node creation itself during the parsing phase.

Code Generation (CG)

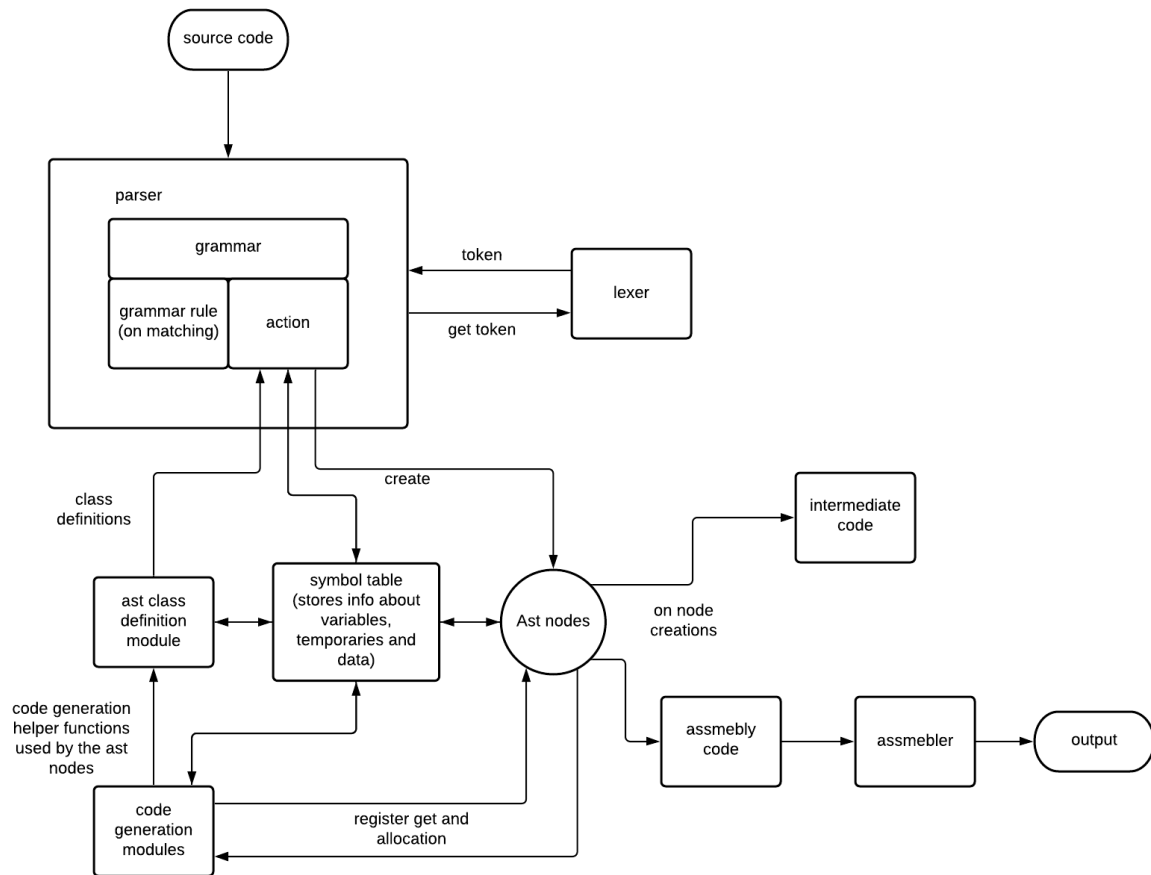
The code generation is similar to the intermediate code generation process. Expect that it requires extra modules taking care of the register allocation for various variables. For this a **register descriptor** is maintained that carries the information about which register is allocated to which variables. There is a get register function that allocates a register to a variable or a temporary. The main challenge in code generation is the limited number of registers but countless number of variables. So whenever all the registers are full, stores and loads are generated to free a few registers and allocate them according to the current need. A FIFO policy has been followed to choose which register to free in such cases.

Symbol Table

The symbol table has been designed to **store variables and temporaries** generated during the intermediate code generation phase. Each table entry carries the information as (identifier, {value, offset, data-type, size, register holding the value of the variable}). So, literally all the information regarding a variable is stored in its symbol table entry and hence all the other compiler phases interact with it. Including the register information in the symbol table entry is an implementation of the **address descriptor**². The symbol table gives put and get endpoints. It also has a function to store the data segment, which will then be written to output mips file.

² Register descriptor and address descriptor are inspired from the dragon textbook

DESIGN DIAGRAM



SOURCE CODE ORGANIZATION

Folder and files

The submitted team11_source_code.tar.gz on extraction (in ubuntu) gives team11_source_code folder which contains compiler folder

- In Folder **compiler**,
source files: lexer.l, parser.y
header files: ast_nodes.hpp, code_generation.hpp, symbol_table.hpp
- In Folder **compiler**, Makefile

- In the Folder **compiler**, folder **test** contains prog1, prog2, prog3, prog4, prog5 test programs and output1, output2, output3, output4, output5 files for the respective test programs and the **executable named compiler** upon build.

Build and generating compiler executable

Download and unzip and move the compiler folder into a folder of your choice and in that directory

```
cd compiler/  
make
```

This will generate an executable named compiler and puts it in the test folder

```
make clean
```

To remove the extra files generated during the build

Usage of the compiler executable

Now to run the test cases, In compiler directory

```
cd test/  
./compiler <infile> <outfile>
```

Output file <outfile> has the assembly code for the corresponding input file <infile>

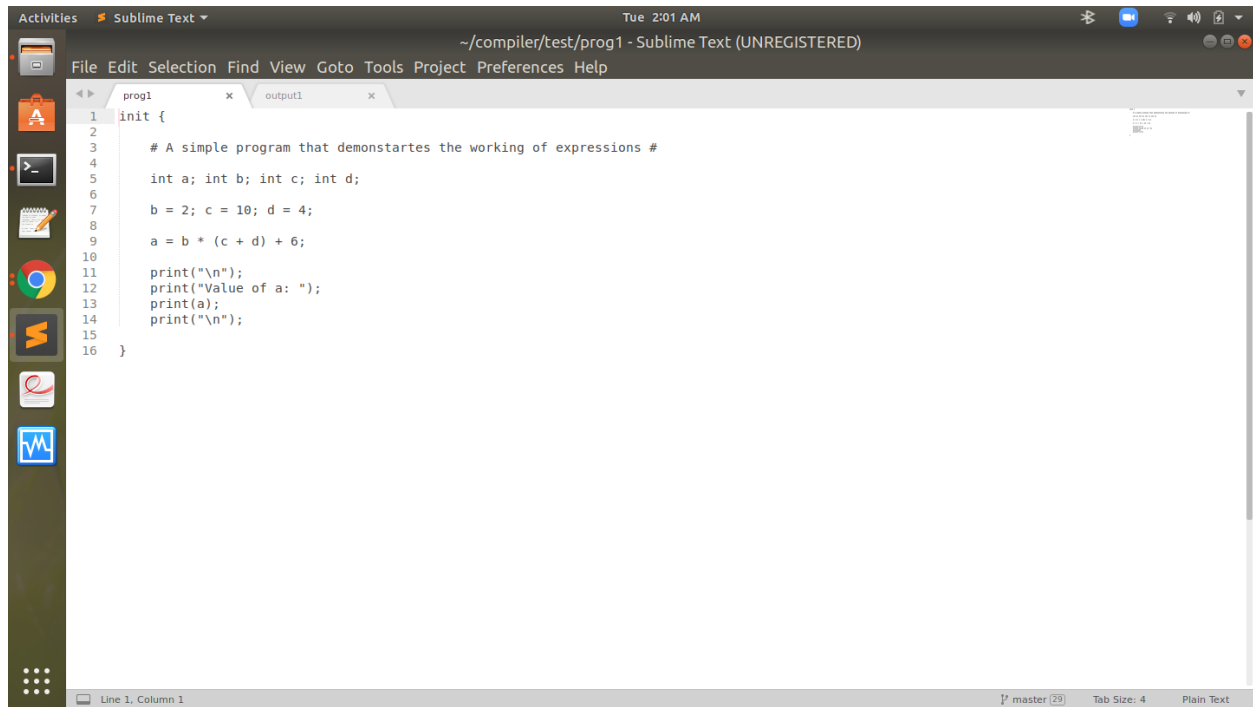
To run this output file execute the following command (spim should be installed in prior)

```
spim -file <outfile> run
```

To write and run a new program create a file in the test folder and use the commands specified above to run the code

TESTING AND EVALUATION OF THE COMPILER

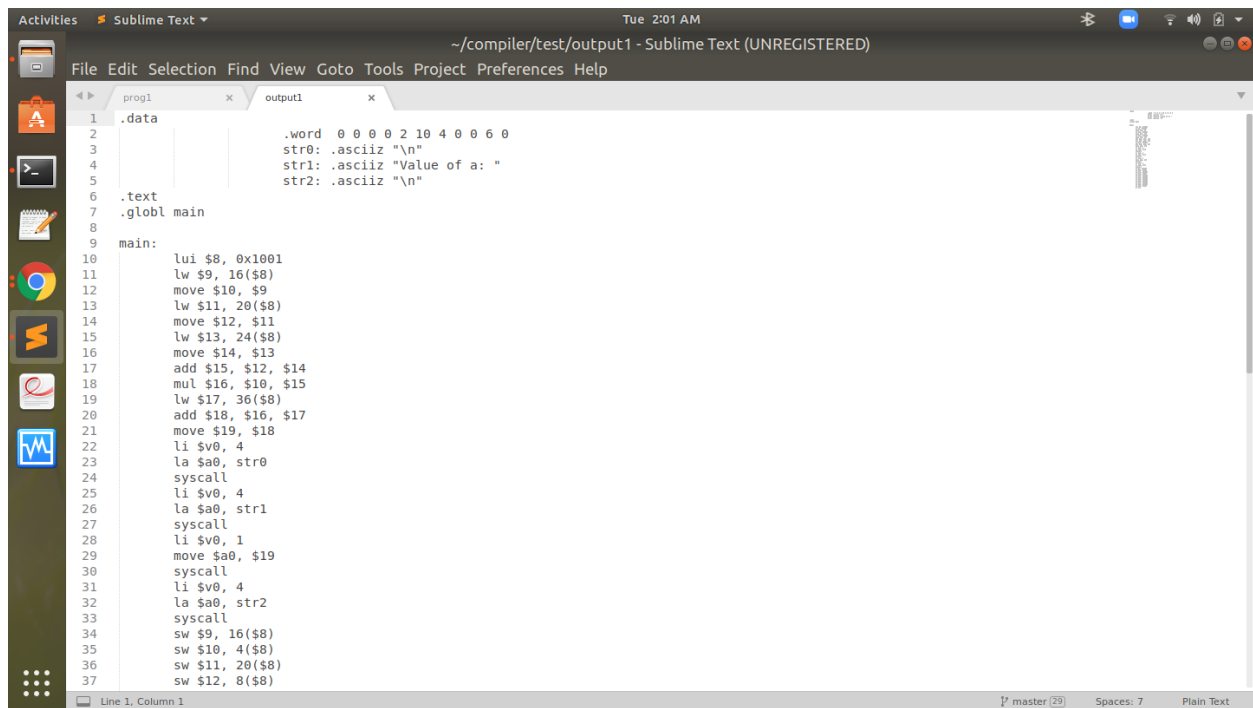
Test 1



```

1 init {
2
3     # A simple program that demonstrates the working of expressions #
4
5     int a; int b; int c; int d;
6
7     b = 2; c = 10; d = 4;
8
9     a = b * (c + d) + 6;
10
11     print("\n");
12     print("Value of a: ");
13     print(a);
14     print("\n");
15
16 }
  
```

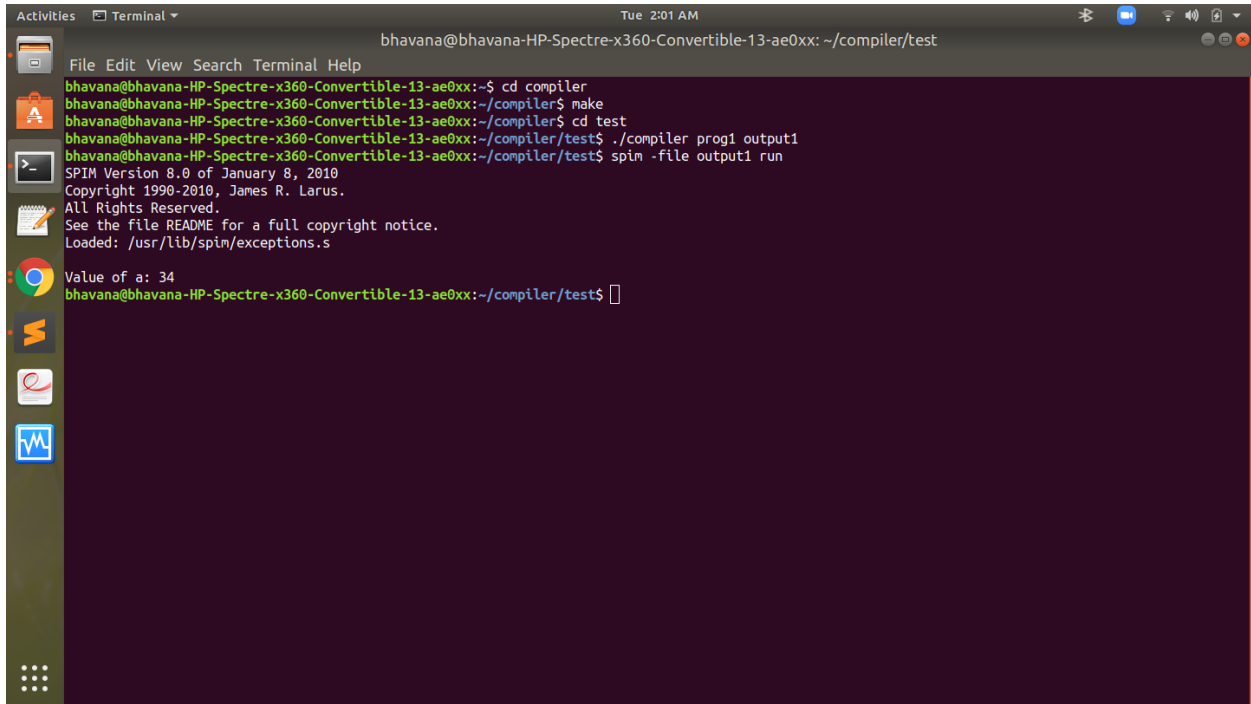
Program-1



```

1 .data
2     .word 0 0 0 0 2 10 4 0 0 6 0
3     str0: .asciiz "\n"
4     str1: .asciiz "Value of a: "
5     str2: .asciiz "\n"
6
7 .text
8 .globl main
9 main:
10     lui $8, 0x1001
11     lw $9, 16($8)
12     move $10, $9
13     lw $11, 20($8)
14     move $12, $11
15     lw $13, 24($8)
16     move $14, $13
17     add $15, $12, $14
18     mul $16, $10, $15
19     lw $17, 36($8)
20     add $18, $16, $17
21     move $19, $18
22     li $v0, 4
23     la $a0, str0
24     syscall
25     li $v0, 4
26     la $a0, str1
27     syscall
28     li $v0, 1
29     move $a0, $19
30     syscall
31     li $v0, 4
32     la $a0, str2
33     syscall
34     sw $9, 16($8)
35     sw $10, 4($8)
36     sw $11, 20($8)
37     sw $12, 8($8)
  
```

Output-1



```

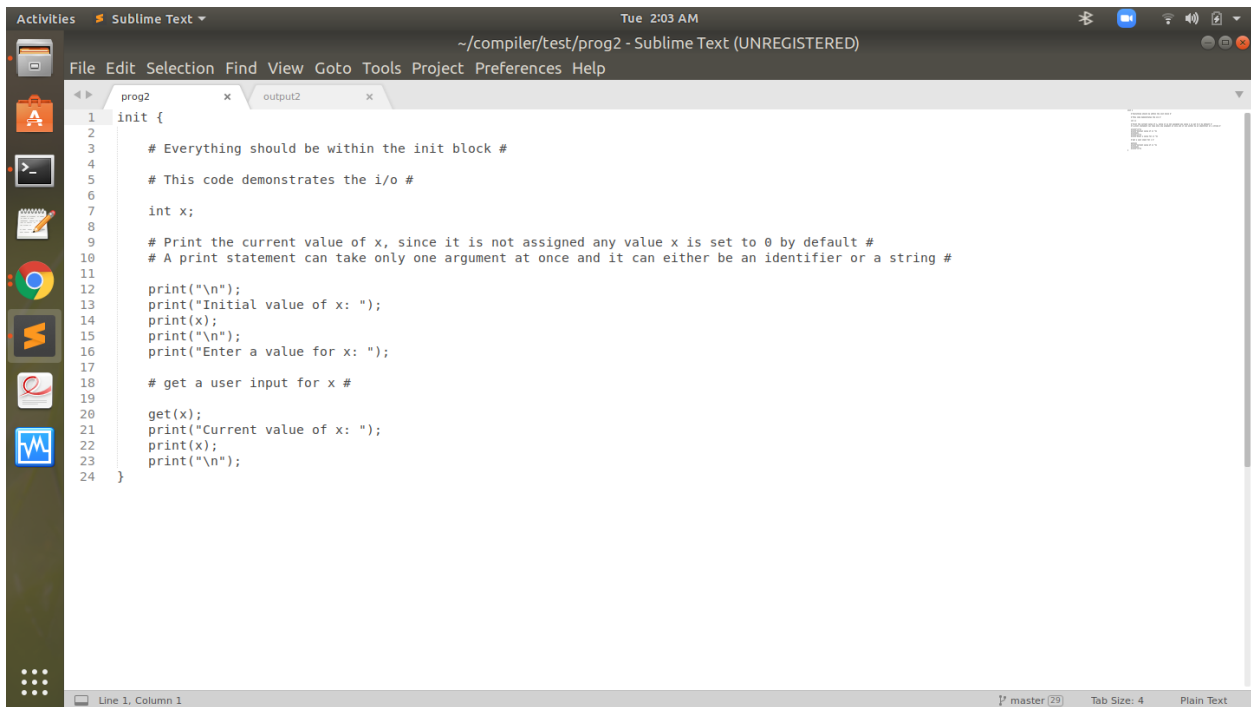
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx: ~/compiler/test
File Edit View Search Terminal Help
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~$ cd compiler
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler$ make
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler$ cd test
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ ./compiler prog1 output1
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ spin -file output1 run
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s

Value of a: 34
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$

```

Result-1

Test 2

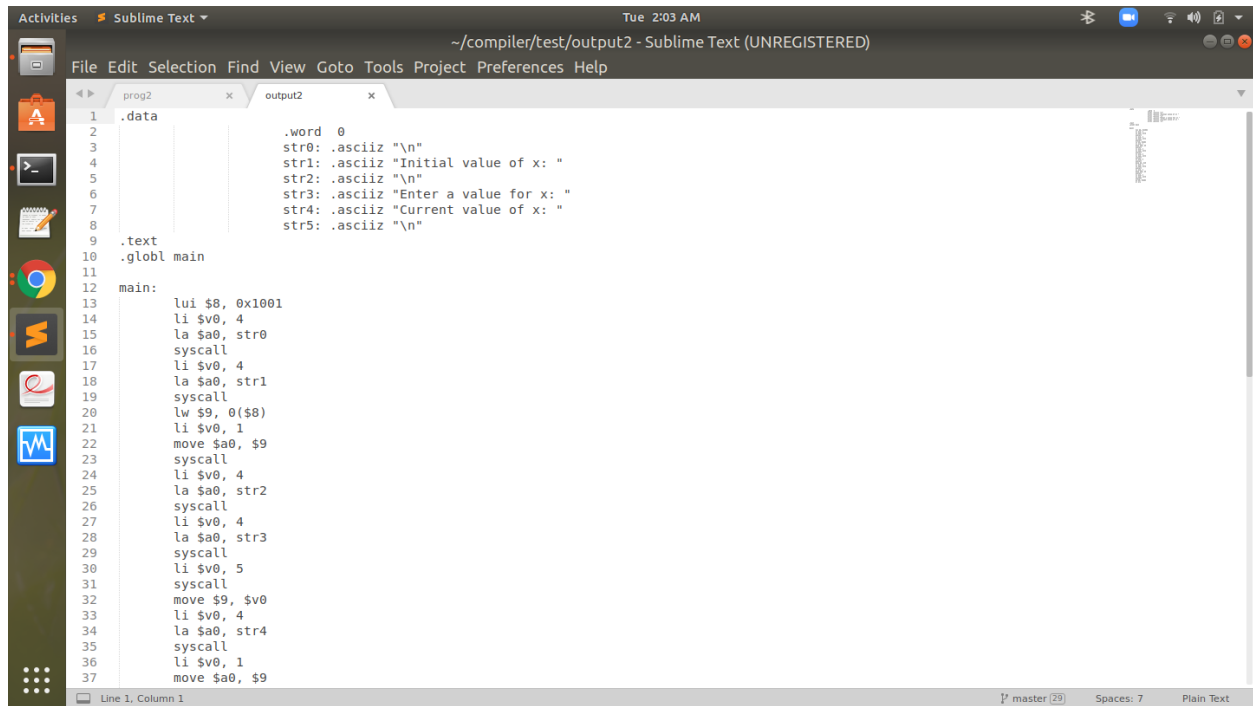


```

~/compiler/test/prog2 - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
1 init {
2
3     # Everything should be within the init block #
4
5     # This code demonstrates the i/o #
6
7     int x;
8
9     # Print the current value of x, since it is not assigned any value x is set to 0 by default #
10    # A print statement can take only one argument at once and it can either be an identifier or a string #
11
12    print("\n");
13    print("Initial value of x: ");
14    print(x);
15    print("\n");
16    print("Enter a value for x: ");
17
18    # get a user input for x #
19
20    get(x);
21    print("Current value of x: ");
22    print(x);
23    print("\n");
24 }

```

Program-2

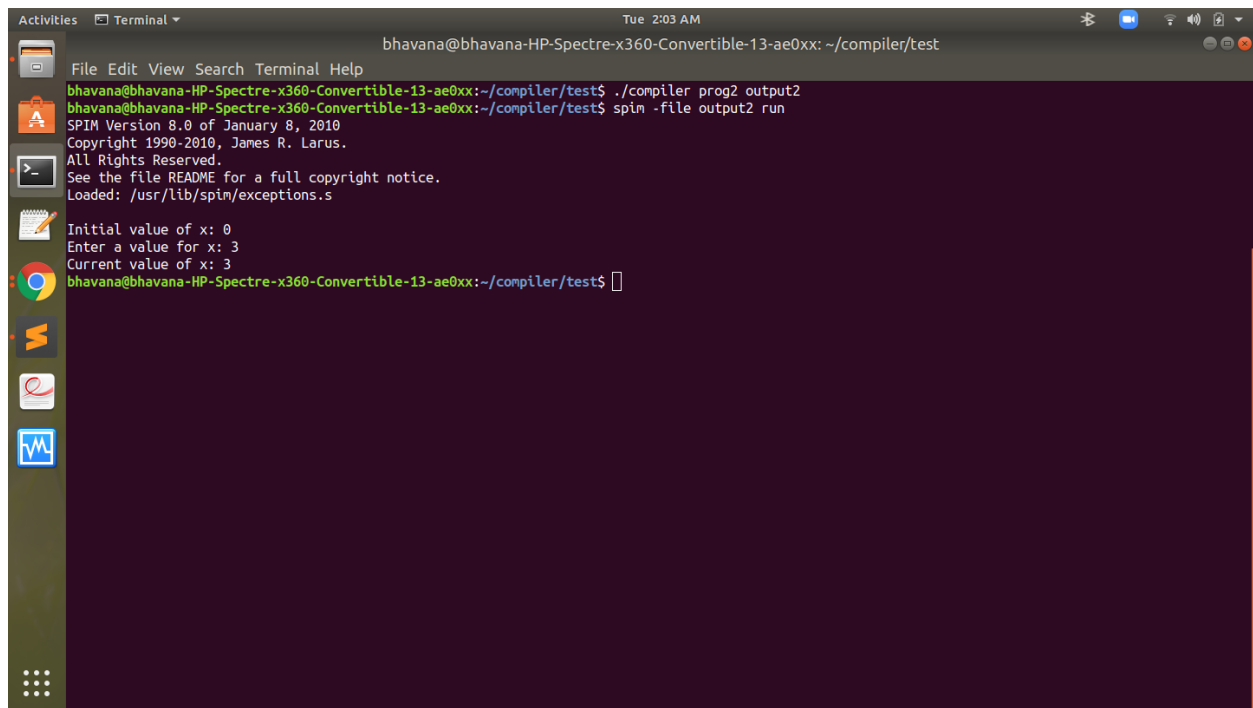


```

1  .data
2
3      .word 0
4      str0: .asciiz "\n"
5      str1: .asciiz "Initial value of x: "
6      str2: .asciiz "\n"
7      str3: .asciiz "Enter a value for x: "
8      str4: .asciiz "Current value of x: "
9      str5: .asciiz "\n"
10
11  .text
12  .globl main
13
14  main:
15      lui $8, 0x1001
16      li $v0, 4
17      la $a0, str0
18      syscall
19      li $v0, 4
20      la $a0, str1
21      syscall
22      lw $9, 0($8)
23      li $v0, 1
24      move $a0, $9
25      syscall
26      li $v0, 4
27      la $a0, str2
28      syscall
29      li $v0, 4
30      la $a0, str3
31      syscall
32      li $v0, 5
33      move $9, $v0
34      li $v0, 4
35      la $a0, str4
36      syscall
37      li $v0, 1
38      move $a0, $9

```

Output-2



```

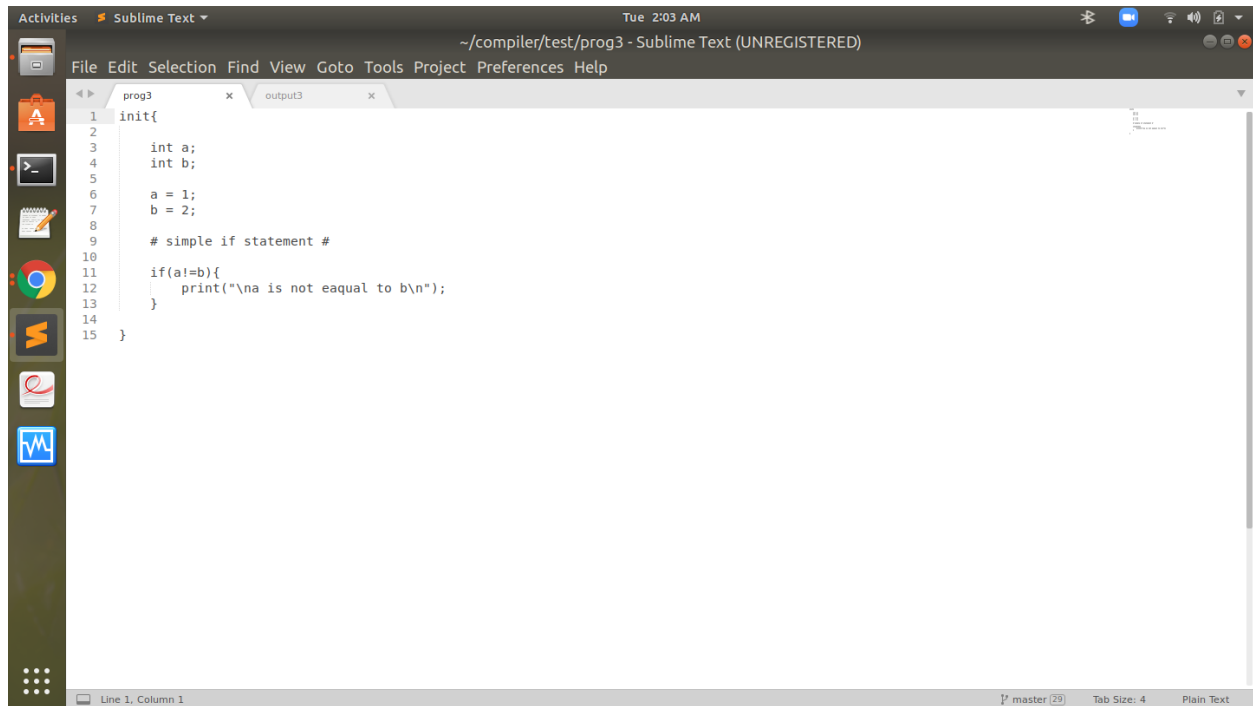
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx: ~/compiler/test
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ ./compiler prog2 output2
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ spin -file output2 run
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s

Initial value of x: 0
Enter a value for x: 3
Current value of x: 3
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$

```

Result-2

Test 3



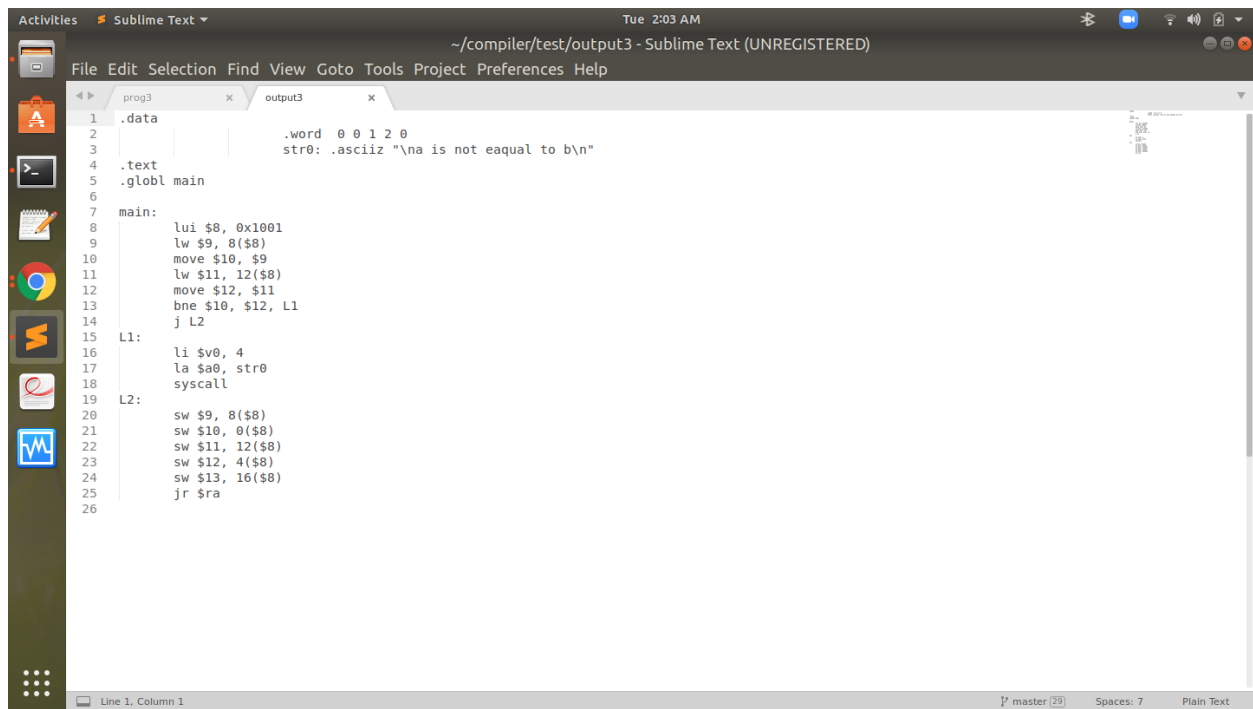
The screenshot shows the Sublime Text editor with a file named `prog3` open. The code is as follows:

```

1  init{
2
3      int a;
4      int b;
5
6      a = 1;
7      b = 2;
8
9      # simple if statement #
10
11     if(a!=b){
12         print("\na is not eaqual to b\n");
13     }
14
15 }
```

The status bar at the bottom indicates "Line 1, Column 1", "master (29)", "Tab Size: 4", and "Plain Text".

Program-3



The screenshot shows the Sublime Text editor with a file named `output3` open. The code is as follows:

```

1  .data
2      .word 0 0 1 2 0
3      str0: .ascii "\na is not eaqual to b\n"
4
5  .text
6  .globl main
7
8  main:
9      lui $8, 0x1001
10     lw $9, 0($8)
11     move $10, $9
12     lw $11, 12($8)
13     move $12, $11
14     bne $10, $12, L1
15     j L2
16
17 L1:
18     li $v0, 4
19     la $a0, str0
20     syscall
21
22 L2:
23     sw $9, 0($8)
24     sw $10, 0($8)
25     sw $11, 12($8)
26     sw $12, 4($8)
27     sw $13, 16($8)
28     jr $ra
29
```

The status bar at the bottom indicates "Line 1, Column 1", "master (29)", "Spaces: 7", and "Plain Text".

Output-3

```

bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx: ~/compiler/test
File Edit View Search Terminal Help
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ ./compiler prog3 output3
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ spin -file output3 run
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s
a is not equal to b
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$

```

Result-3

Test 4

```

~/compiler/test/prog4 - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
prog4 x output4 x
1 init {
2
3     # this code demonstrates the functioning of a simple if-else and if statements #
4
5     int a;
6     a = 2;
7
8     print("\n");
9
10    if(!((a==1)||(a==1))){
11        print("abs(a) is not 1\n");
12    }
13    else{
14        print("abs(a) is 1\n");
15    }
16
17    if(a!=0){
18        print("a is non zero\n");
19    }
20
21 }

```

Program-4

The screenshot shows the Sublime Text editor with two files open: 'prog4' and 'output4'. The 'prog4' file contains MIPS assembly code for a program that checks if the absolute value of a number is 1 or non-zero. The 'output4' file is empty.

```

1 .data
2     .word 0 2 1 0 0 1 0 0 0
3     str0: .asciiz "\n"
4     str1: .asciiz "abs(a) is not 1\n"
5     str2: .asciiz "abs(a) is 1\n"
6     str3: .asciiz "a is non zero\n"
7 .text
8 .globl main
9
10 main:
11     lui $8, 0x1001
12     lw $9, 4($8)
13     move $10, $9
14     li $v0, 4
15     la $a0, str0
16     syscall
17     lw $11, 8($8)
18     sub $12, $0, $11
19     lw $13, 20($8)
20     beq $10, $12, L2
21     j L3
22 L3:
23     beq $10, $13, L2
24     j L1
25 L1:
26     li $v0, 4
27     la $a0, str1
28     syscall
29     j L4
30 L2:
31     li $v0, 4
32     la $a0, str2
33     syscall
34 L4:
35     lw $16, 20($8)
36     bne $10, $16, L5
37     j L6

```

Output-4

The screenshot shows a terminal window with the following commands and output:

```

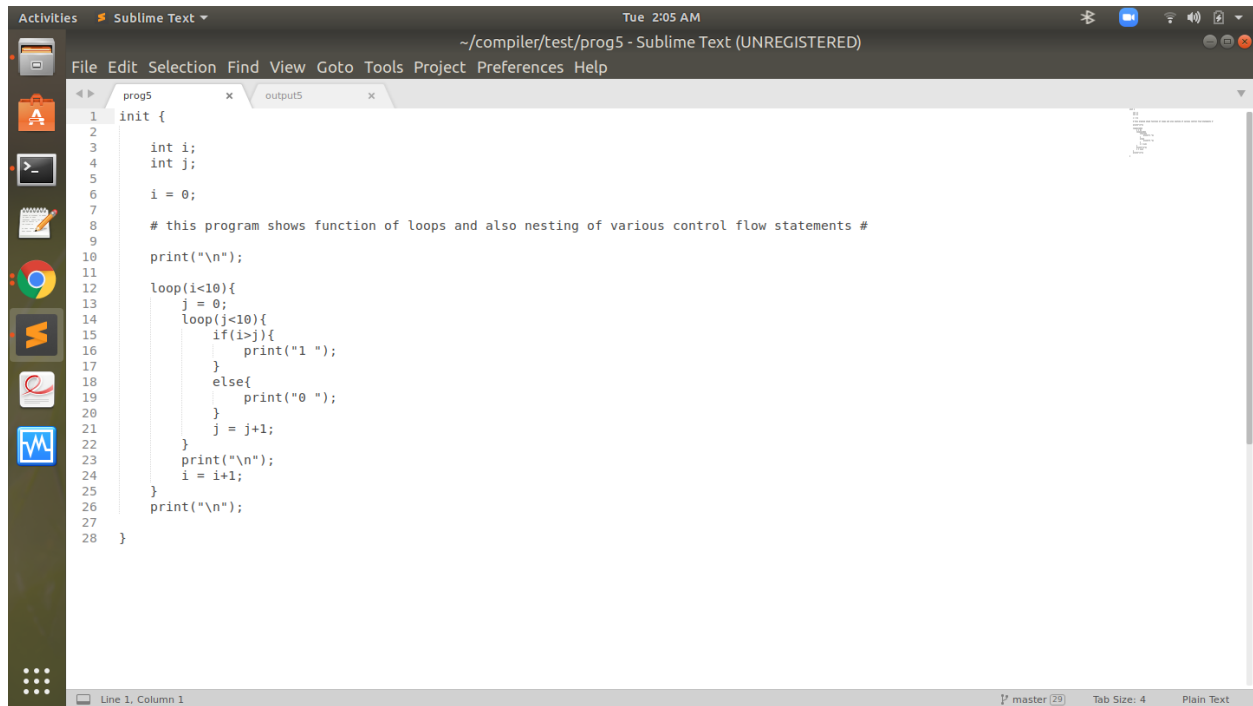
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx: ~/compiler/test
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ ./compiler prog4 output4
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ spin -file output4 run
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s

abs(a) is not 1
a is non zero
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$

```

Result-4

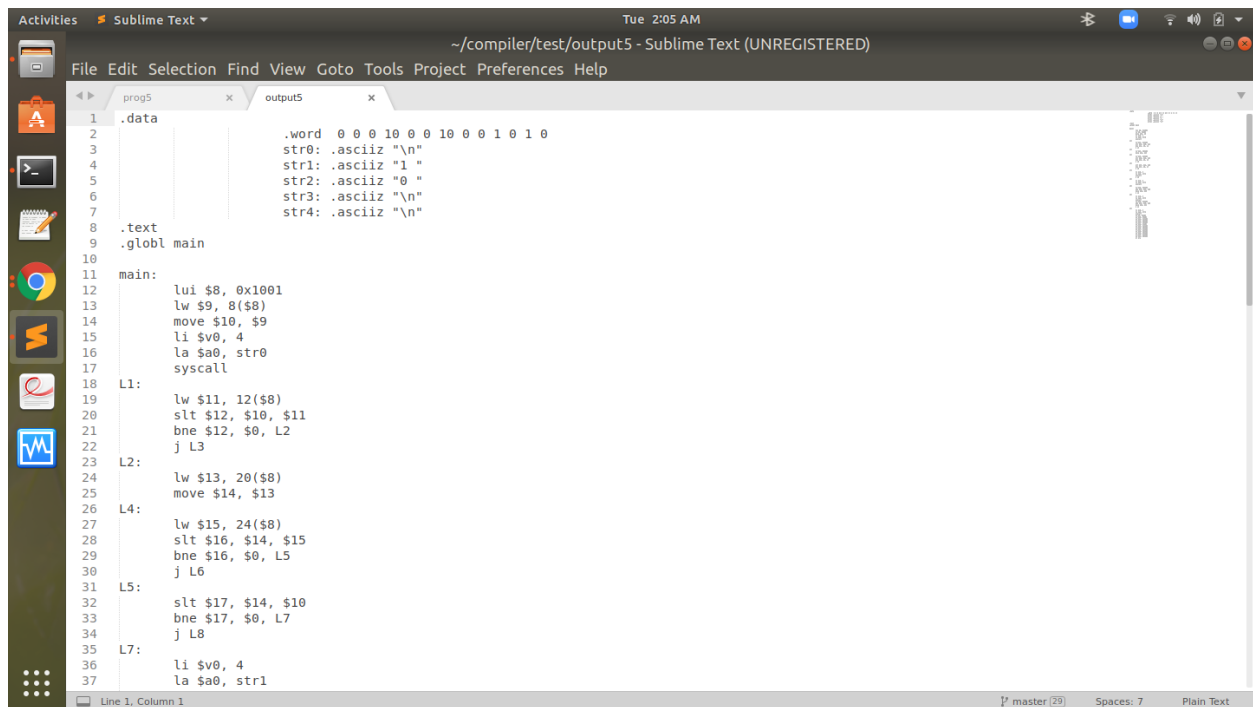
Test 5



```

1 init {
2
3     int i;
4     int j;
5
6     i = 0;
7
8     # this program shows function of loops and also nesting of various control flow statements #
9
10    print("\n");
11
12    loop(i<10){
13        j = 0;
14        loop(j<10){
15            if(i>j){
16                print("1 ");
17            }
18            else{
19                print("0 ");
20            }
21            j = j+1;
22        }
23        print("\n");
24        i = i+1;
25    }
26    print("\n");
27
28 }
  
```

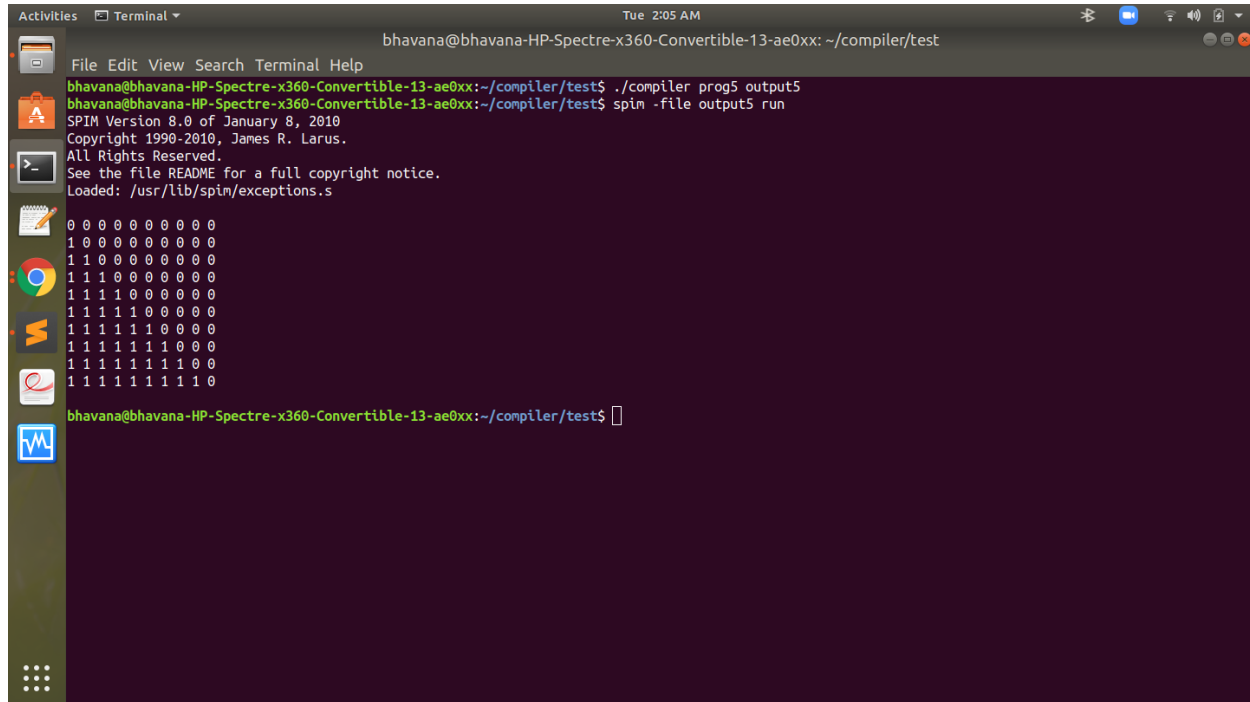
Program-5



```

1 .data
2
3     .word 0 0 0 10 0 0 10 0 0 1 0 1 0
4     str0: .ascii "\n"
5     str1: .ascii "1 "
6     str2: .ascii "0 "
7     str3: .ascii "\n"
8     str4: .ascii "\n"
9
10 .text
11 .globl main
12
13 main:
14     lui $8, 0x1001
15     lw $9, 0($8)
16     move $10, $9
17     li $v0, 4
18     la $a0, str0
19     syscall
20
21 L1:
22     lw $11, 12($8)
23     slt $12, $10, $11
24     bne $12, $0, L2
25     j L3
26
27 L2:
28     lw $13, 20($8)
29     move $14, $13
30
31 L4:
32     lw $15, 24($8)
33     slt $16, $14, $15
34     bne $16, $0, L5
35     j L6
36
37 L5:
38     slt $17, $14, $10
39     bne $17, $0, L7
40     j L8
41
42 L7:
43     li $v0, 4
44     la $a0, str1
  
```

Output-5



```

bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx: ~/compiler/test
File Edit View Search Terminal Help
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ ./compiler prog5 output5
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$ spin -file output5 run
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s
000000000000
100000000000
110000000000
111000000000
111100000000
111110000000
111111000000
111111100000
111111110000
111111111000
111111111100
111111111110
bhavana@bhavana-HP-Spectre-x360-Convertible-13-ae0xx:~/compiler/test$

```

Result-5

LIMITATIONS

The symbol table can support scope handling but this functionality has not been integrated to the current version of the compiler. The compiler does not do extensive semantic analysis. And the error messages are not specific. The compiler does not do linking and loading. It simply generates the MIPS assembly code which should then be executed using spim from the terminal or Qtspim by the user manually (as this process is not automated)

CONCLUSION

By the end of this project we have

- Designed a custom programming language
- Written a compiler for it with all the phases till the code generation
- Learnt designing a complicated system like a compiler by dividing it into various phases and modules

- Got a good grip over OOP concepts through cpp
- Have obtained a better understanding of compiler design principles

CONTRIBUTIONS

CS18B004 Akshitha

- Initial stages of coding in the symbol table (later developed in cpp)
- Helped in language manual

CS18B009 Deepanvi Penmetcha

- Initial stages of coding Lexer, Parser in C (later developed in cpp)
- Language Manual
- Helped in report writing

CS18B013 G Nikhitha Vedi

- Initial stages of coding Lexer, Parser in C (later developed in cpp)
- Language Manual
- Helped in report writing

CS18B031 S Bhavana

- Designed, developed and coded all the compiler modules : lexer, parser, abstract syntax tree, symbol table, intermediate code generation and code generation
- Planned test cases and tested the compiler along with spim
- Project report, Video demonstration and helped in language manual