**A PRELIMINARY REPORT ON**

# GENERATING COMPILER PHASES FROM SPECIFICATIONS OF INTERMEDIATE REPRESENTATIONS

SUBMITTED TO THE SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE
IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF THE DEGREE
OF

**BACHELOR OF COMPUTER ENGINEERING**

**SUBMITTED BY**


| | |
|---|---|
| Sai Ghule | C22018221334 |
| Bhagyashree Rane | C22018221311 |
| Shravasti Deore | C22018881911 |



**DEPARTMENT OF COMPUTER ENGINEERING**

**2021 -2022**


# CERTIFICATE

This is to certify that the project report entitled

**"GENERATING COMPILER PHASES FROM SPRCIFICATIONS of INTERMEDIATE REPRESENTATIONS (IRs)"**

Submitted by


| | |
|---|---|
| Sai Ghule | C22018221334 |
| Bhagyashree Rane | C22018221311 |
| Shravasti Deore | C22018881911 |


is a bonafide student of this institute and the work has been carried out by her under the supervision of **Prof. Dr. Chhaya Gosavi** and it is approved for the partial fulfillment of the requirement of Savitribai Phule Pune University, for the award of the degree of **Bachelor of Computer Engineering.**

**Prof. Dr. Chhaya Gosavi**
Guide
Department of Computer Engineering

**Dr. Supriya Kelkar**
Head,
Department of Computer Engineering

**Dr. M. B. Khambete**
Principal,
Cummins College of Engineering for Women Pune – 52

Place : Pune

Date : 25/1/2022

# ACKNOWLEDGEMENT

We would like to express our sincere gratitude towards our project guide Prof.Dr.Chhaya Gosavi for her constant encouragement and valuable guidance during the completion of this project work.

We would like to thank our external mentor Prof.Dr.Uday Khedkar sir for his constant guidance and support.

We would also like to thank Dr. Supriya Kelkar (H.O.D.,Comp) for her continuous valuable guidance, support, valuable suggestions and her precious time in every possible way inspite of her busy schedule throughout our project activity. We take this opportunity to express our sincere thanks to all the staff members of Comp. Department for their constant help whenever required. Finally, we express our sincere thanks to all those who helped us directly or indirectly in many ways in completion of this project work.

**Sai Ghule**

**Bhagyashree Rane**

**Shravasti Deore**

# ABSTRACT

The motivation behind the implementation of SCLP has been to create a small well-crafted code base for a UG compiler construction lessons such that it can be enhanced systematically by the students. The focus is on incremental construction with some increments coming from language features and some coming from the phases in compilation.
The project, as a whole, aims to construct a simple compiler used to teach compilers at IITB, however, it does not include production compiler complexities such as code optimization, etc. Generating Compiler Phases from Specifications of IRs is just one incremental construction toward SCLP.

A compiler generator is a program which from its input produces a compiler which translates from one programming language to another. Compiler generation is the field of Computer Science concerned with using computers as a tool for constructing compilers.
Programmers who have written compilers by hand have realized that the process of writing compilers is completely systematic to be written by hand and there should be a way to be done by a machine.

The Lexical Analysis and Parsing phases are well understood that it has been possible to produce parser generators that produce a parser for the specific language. The parser generator are produced from the description of the syntax of the programming language. For example, LEX & YACC.

To generate a code for a language, it is necessary to have a clear understanding of the semantics of the language. In order to be able to make a compiler generator, one needs a theoretical framework for expressing the semantics of the languages involved.

The formal specifications of a language provides insights and understanding of the language requirements and the language statements design.

This project is to create formal language specifications for AST(Abstract Syntax Tree), TAC(Three Address Code), RTL(Register Transfer Logic) of the SCLP compiler to generate Translation Rules from the language specifications.
The Translation Rules thus generated will be the input to the Translator Generator which in turn would generate an auto generated IRGen to translate the output of the previous phase to the next phase.

# TABLE OF CONTENTS

## LIST OF ABBREVATIONS

| ABBREVIATION | ILLUSTRATION |
|---|---|
| AST | Abstract Syntax Tree |
| TAC | Three Address Code |
| RTL | Register Transfer Language |
| ASM | Assembly Language |

# LIST OF TABLES

# 1. INTRODUCTION

## 1.1 MOTIVATION

The SCLP compiler was initiated to create a small well-crafted code base which has been used for UG compiler construction lessons. It is designed in such a way that it can be enhanced systematically by the students. The prominent focus is on increments from language features and some from phases of compilation.

Generating Compiler Phases from Specifications of IRs is one such incremental construction.

The secondary goal of SCLP is to understand the relationship between compilation and interpretation and to support interpretation.

## 1.2 PROBLEM DEFINITION

A compiler generator is a program which produces a compiler from its input, which translates from one programming language to another. In the field of Computer Science, compiler generation is concerned with using computers as a tool for constructing compilers.
Programmers who have hand-written compilers have realized that the process of writing compilers is completely systematic to be written by hand and there should be a way to be done by a machine. Lexical Analysis and Parsing have been understood well enough that it has been possible to produce parser generators that produce a parser for the specific language.
However, the interfaces between the subsequent intermediate representations of compilation (namely, TAC, RTL, ASM) have not yet achieved code generation on the scale scanners and parsers have.

# 2. LITERATURE SURVEY

## 2.1 BACKGROUND OF DOMAIN

- SCLP which is a language processor, takes input programs written in languages that are a subset of C. The read program based on command line parameters is either interpreted or compiled which generates an equivalent .spim file (assembly language program).
- Compilation of the input program through SCLP involves translating one representation into another representation (each representation is an Intermediate Representation) in a series of steps.
- The series of steps of translating an input to asm are:

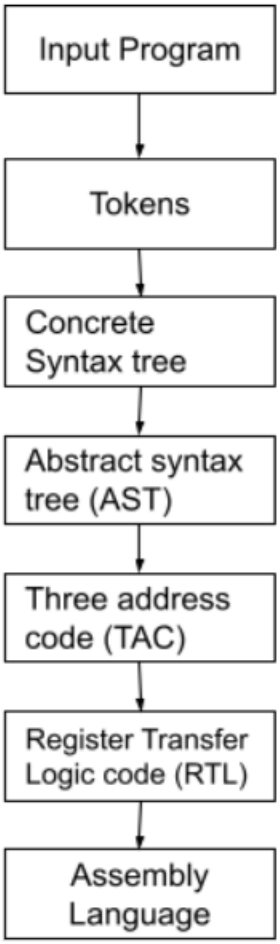



Figure 2.1- Steps for translating an input to ASM

- Other than the concrete syntax tree, which is constructed implicitly during parsing all these representations are

available as text files. These are produced by appropriate command line switches.

- Till date there are six levels of the input language accepted by the SCLP compiler. Each level includes all the features of the previous level. Studying these levels was to be given preference.
- Some errors in the levels needed to be pointed out and studied in order to find solutions for them.
- Studying the specifications written for ASM(Assembly language) and generating the specifications for AST, TAC and RTL.
- Specification document has statements expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.
  For example, generating specifications for ASM
  ASM:
  addi rt, rs, imm

  Specification:

$$[Add\ ints]\frac{S(addi, o1, o2, o3) : \tau^S_{cmp} \qquad o1 : \tau^S_{gpr} \qquad o2 : \tau^S_{gpr} \qquad o3 : \tau^S_{int}}{H; D; C; R \vdash S(addi, o1, o2, o3) \rightarrow H; D; C; R[o1 \mapsto R[o2] + o3] \qquad R : \tau^S_{gpr}}$$

- The formal expression clearly tells the language designer that the instruction **addi** is of the phase **S**(i.e. ASM) and of type **cmp**(i.e. computational), it takes in 3 operands: **o1, o2, o3** of which **o1** and **o2** are of type **gpr**, and o3 is of type **int**.
- **H;D;C;R** is the environment that undergoes a change after the expression **S(addi, o1, o2, o3)** is executed.
- **R**(the Register File) updates **R[o1]** to **R[o2]+o3** according to the instruction.

## 2.2 COMPARISIONS, RESEARCH PAPERS STUDIED / PRODUCTS COMPARED

- V. Aho, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, J. D. Ullman, 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

  Standard textbook about compiler construction for programming languages. Has in depth coverage of topics like lexical analysis, parsing, generation of intermediate code, etc. Also, describes development of a mini compiler.

  Has level-wise production of a mini compiler.
  Later the book introduces to the principal ideas in syntax-directed definitions and syntax-directed translations.

  The theory of the syntax-directed definitions and syntax-directed translations is used in the later chapters to generate intermediate code for one programming language.

  The book also covers code optimization and garbage collection which this project is sidelining so as to reduce the complexity and not go off the main aim of the project i.e. compiler generators from IR specifications.

- Tofte. M. 2012 compiler generators: what they can do, what they might do and what they will never do. Springer Science & Business Media.

  Reports on the Compiler Generator CERES. The concept of Compiler Generator is described along with the comparison of CERES with other systems. Although, it does not say anything about intermediate representations.

  Has insights about language specifications and how to use them to create compiler generators:

  *"A structural language definition consists of a finite number of rewrite rules that describe how source programs can be rewritten into terms of another language, the semantic language. There is one rule for each language construction of the source language. The construction in question appears on the left-hand side of the rule and the right-hand side is a term of the semantic language, the subterms of which may be expressed indirectly as the translation of the subphrases of the source language construction"*

- Lee, P.. Realistic Compiler Generation. MIT Press.

  Formal Specifications, their need and high-level descriptions, and generation of realistic compilers is primarily treated. Used denotational semantics.

  Presents a new method for expressing the formal semantics of programming languages that allows realistic compilers to be generated automatically. The compilers thus generated are as efficient as the hand written compilers. The introduction of formal semantics make it easier to read and comprehend the semantics.

  Demonstrates a working compiler generator called MESS which generates a compiler for pascal like language.

# 3. REQUIREMENTS

## 3.1. SYSTEM REQUIREMENTS SPECIFICATION

### 3.1.1 In Scope:

- Generation of ASM generator for SPIM
- Specifications for SPIM
- Specifications for RTL
- Machine readable form of the specs and translation rules
- Implementation of Translator Generator

- Generation of ASM generator for RIPES
- Specifications for RIPES
- Specifications for RTL
- Machine readable form of the specs and translation rules

- Generation of TAC generator
- Specifications for AST
- Specifications for TAC
- Translation rules from ASM to TAC
- Machine readable form of the specs and translation rules
- Implementation of Translator Generator

### 3.1.2 Not In Scope:

Inclusion of production compiler complexities like Code Optimization.

Implementation of generators that will construct interpreters.

### 3.1.3 Functional Requirements

- The IR generator should be generated using the Translator Generator using the machine readable form of the specifications and translation rules.
- The compiler should be able to read an input .c program and generate the expected outputs.
- SCLP should be able to generate AST, TAC, and RTL files as the outputs of the .c program.
- SCLP should be able to generate subsequent IR files for IR inputs.

### 3.1.4 Non-Functional Requirements

Software Quality Attributes:

- Availability

The generators should perform the tasks it is assigned to perform.

- Reliability

All levels of .c programs should be accurately executed.

- Flexibility

The system should be flexible for future development.

- Usability

The system should be easily usable for the UG courses.

### 3.1.5 Software Requirements

Linux - Ubuntu(Terminal)
Python 3
gcc(GNU Compiler Collection)
RIPES - RISCV simulator
SPIM - MIPS simulator
LEX & YACC

Hardware Requirements

Operating System - Linux distribution, Ubuntu

# 3. SYSTEM DESIGN

## 4.1 SYSTEM ARCHITECTURE

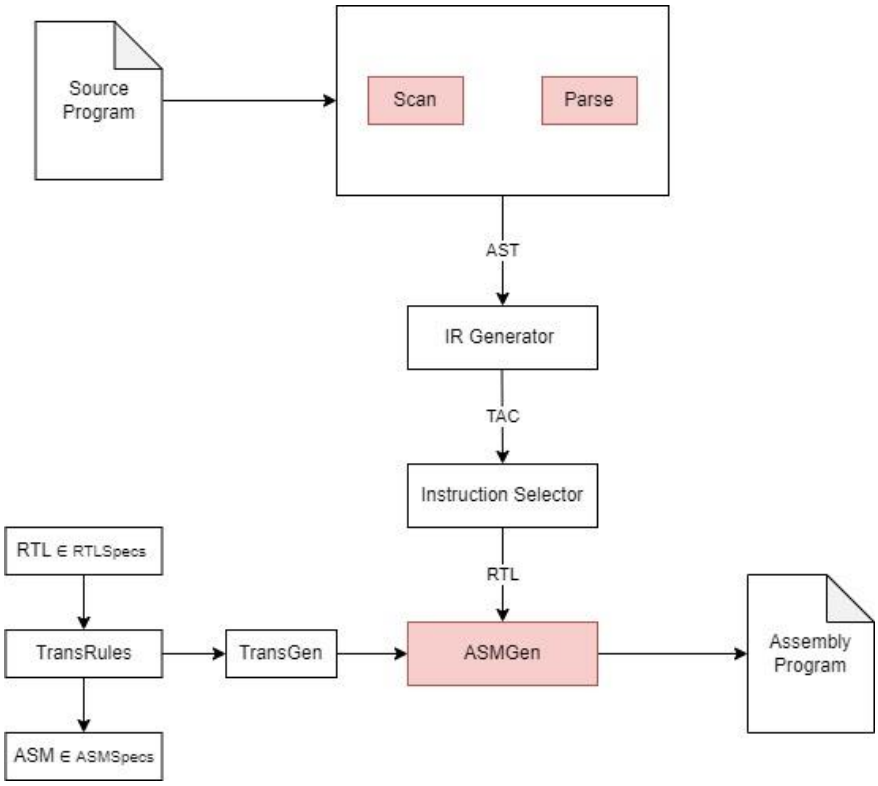



Fig 4.1- System Architecture of SCLP

Description-

- The above diagram is the previous implementation of SCLP.
- When a source program was passed to the sclp compiler, it was first scanned and parsed and the AST i.e the Abstract Syntax Tree was generated.
- The AST was then passes to the IR generator which converted the AST to a TAC i.e Three Address Code.
- The TAC was then passed to the Instruction Selector which produced the RTL i.e Register Transfer Level output.
- The RTL output was passed on to the ASM Gen which produced the Assemble Program.
- The previous group tried to automate the process of generating Assembly program by writing RTL specifications and then generating Translation Rules from them.
- Then they wrote the TransGen i.e the Translator Generator and the ASM gen i.e the ASM generator which generated the Assembly program.
- Their work made it possible to automatically generate the assembly program from RTL specifications.

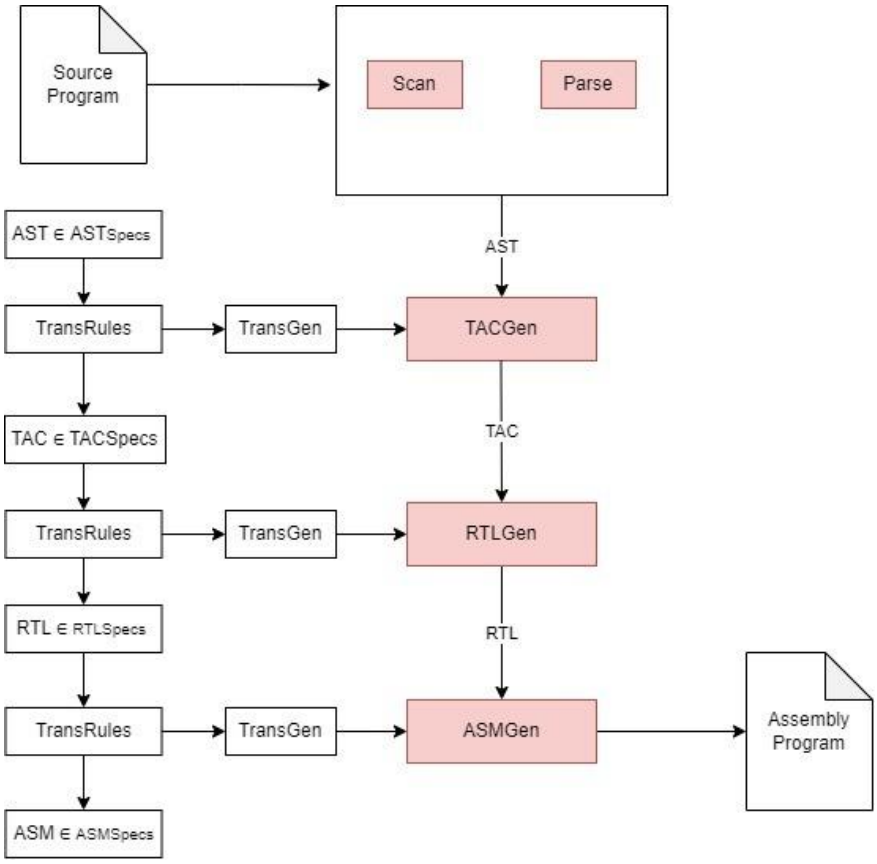



Fig 4.2- High level model

Description-

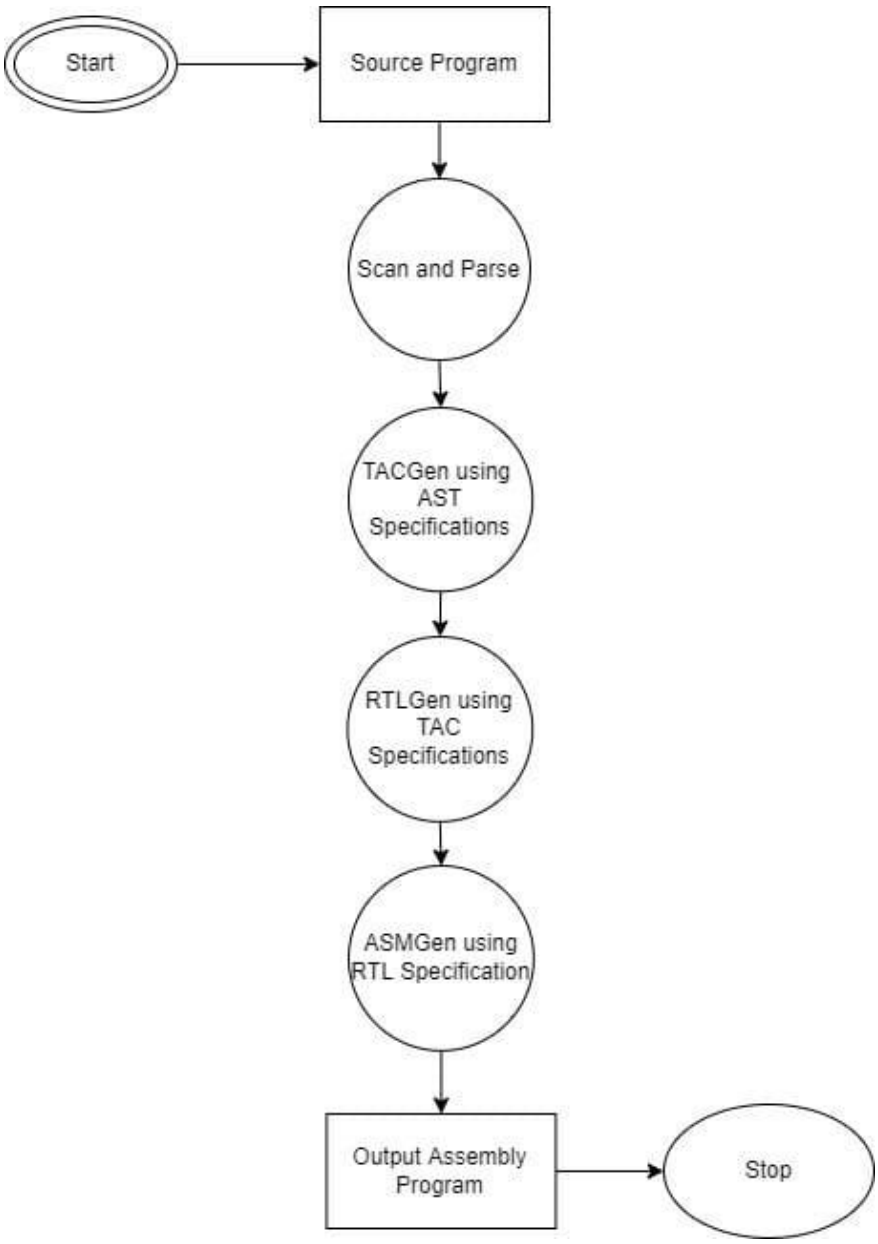
- The previous group tried to automate the process of generating Assembly program by writing RTL specifications and then generating Translation Rules from them, we are trying the same for the rest of the IR phases.

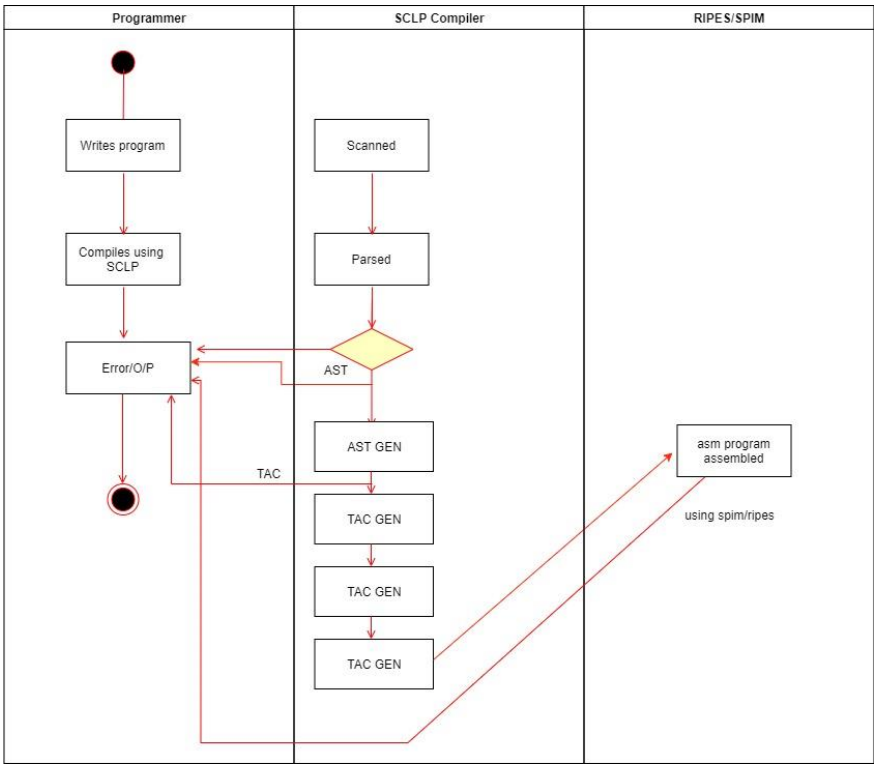
- We are going to write the AST specifications first and then generate the translation rules for AST. Then write the TransGen and TACGen and generate TAC from that.

- We are going to write the TAC specifications and then generate the translation rules for TAC. Then write the TransGen and TACGen and generate RTL from that.

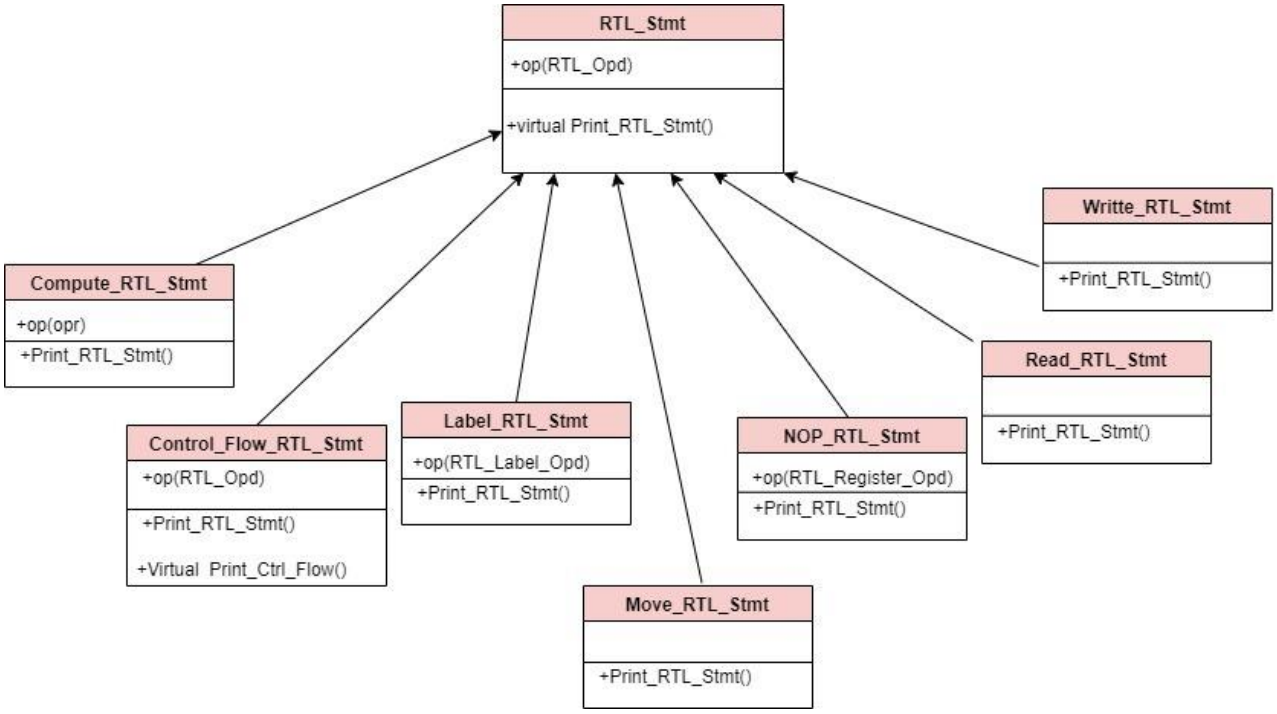
4.2 **UML DIAGRAMS** –

4.2.1 Data Flow Diagram

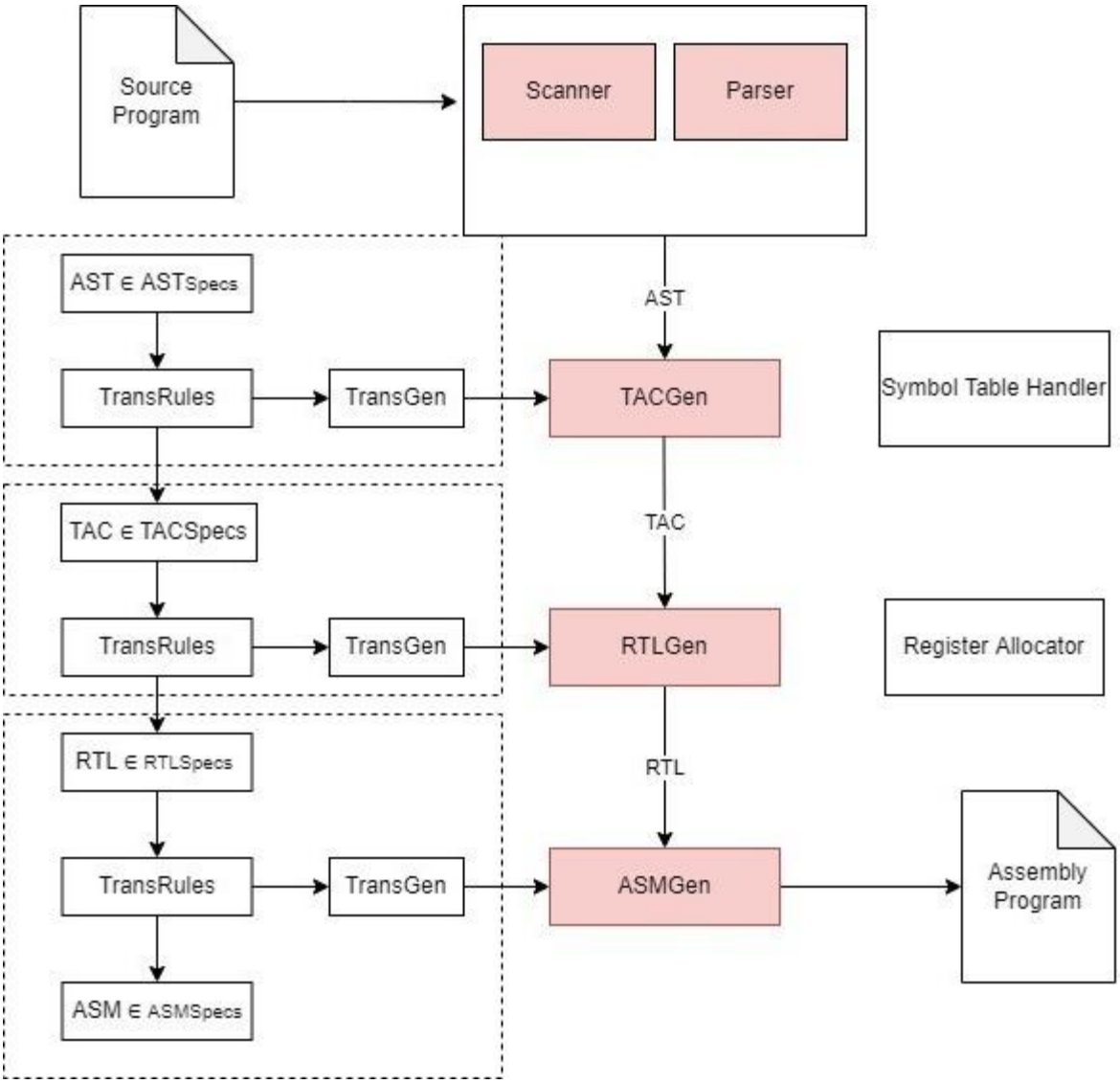




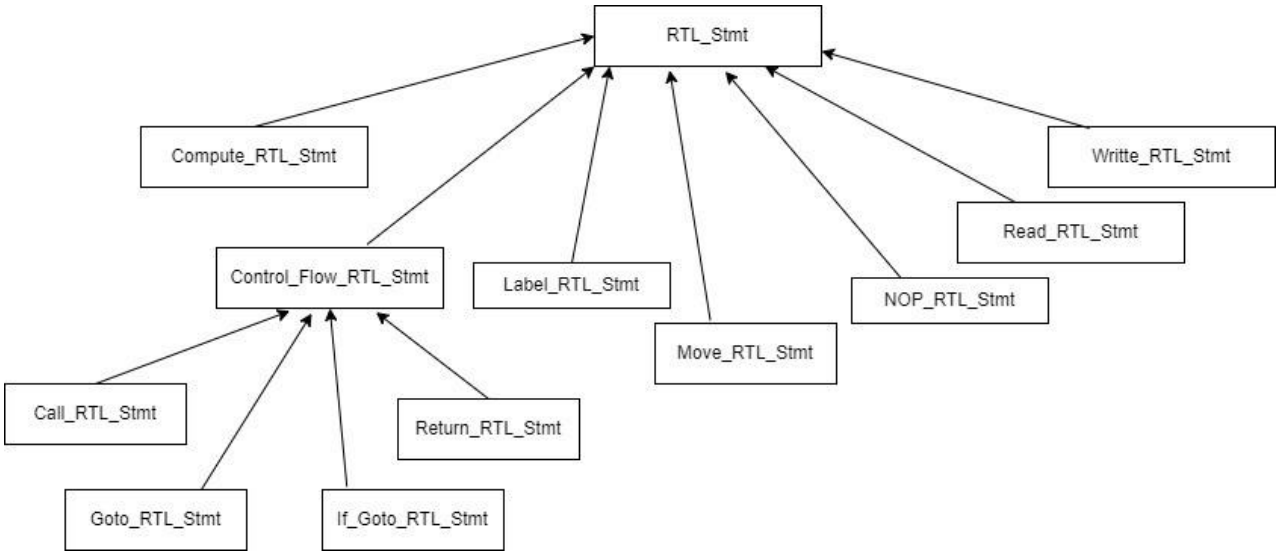

4.2.2 Activity Diagram



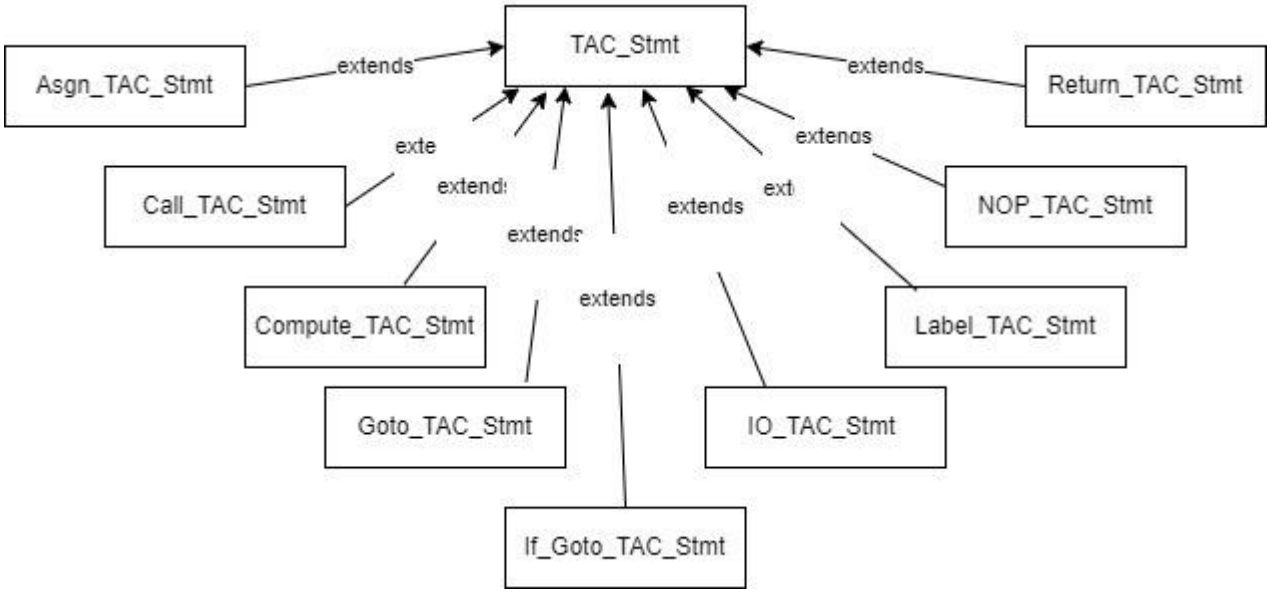

4.2.3 Class Diagram



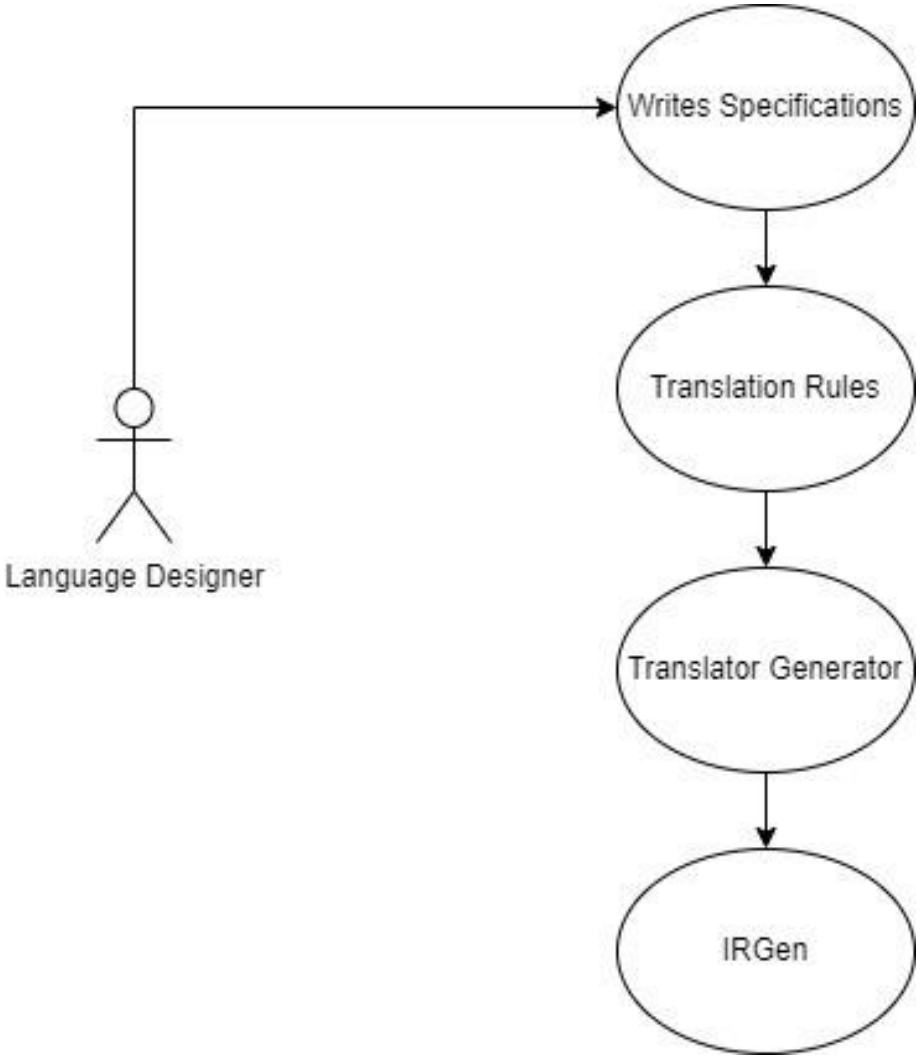

4.2.4 Component Diagram



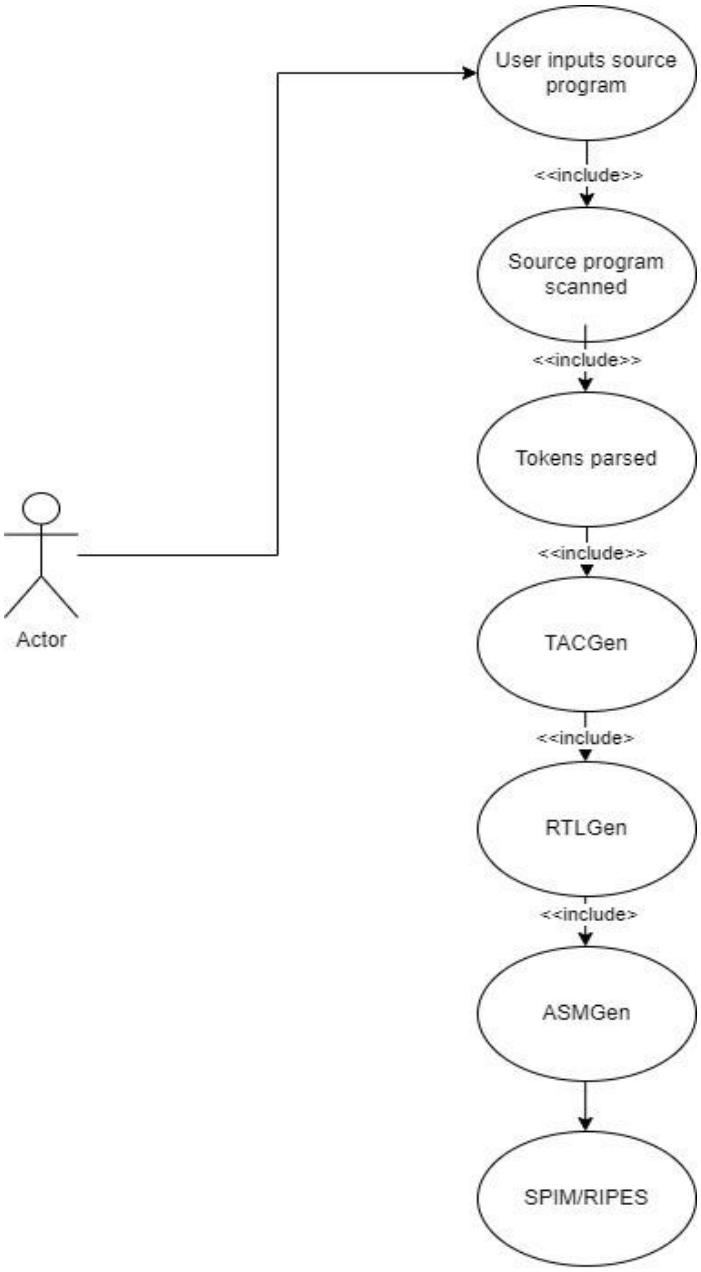

4.2.5 Class Diagram for RTL




4.2.6 Class Diagram for TAC




4.2.7 IR Generator Use Case Diagram




<<include>>

4.2.8 SCLP Use Case Diagram

# 5. IMPLEMENTATION ASPECTS

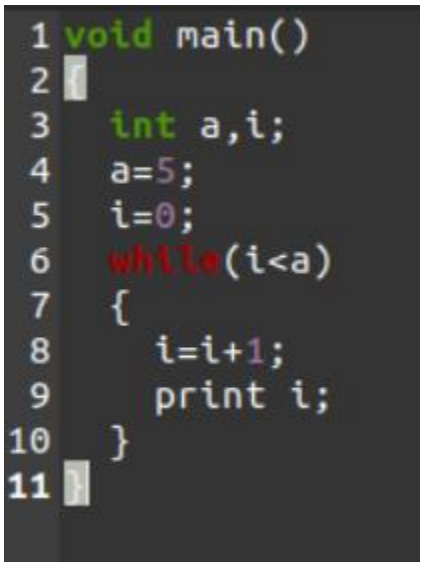
## 5.1 STUDY SCLP -

SCLP has 6 levels built atop each other.

Each level allows more and more functionalities incrementally.

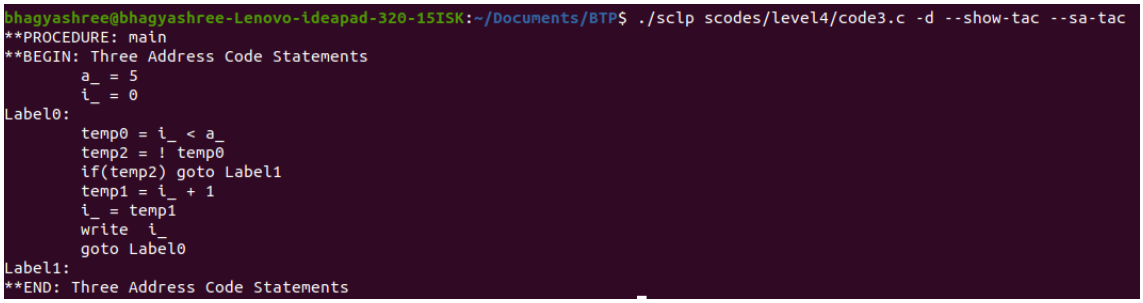
Compiled each level .c programs to produce AST, TAC, RTL, ASM, Symbol Table of each code.

Eg. Level 4 code



```
1  void main()
2  {
3    int a,i;
4    a=5;
5    i=0;
6    while(i<a)
7    {
8      i=i+1;
9      print i;
10   }
11 }
```

TAC for the above code



```
bhagyashree@bhagyashree-Lenovo-ideapad-320-15ISK:~/Documents/BTP$ ./sclp scodes/level4/code3.c -d --show-tac --sa-tac
**PROCEDURE: main
**BEGIN: Three Address Code Statements
        a_ = 5
        i_ = 0
Label0:
        temp0 = i_ < a_
        temp2 = ! temp0
        if(temp2) goto Label1
        temp1 = i_ + 1
        i_ = temp1
        write  i_
        goto Label0
Label1:
**END: Three Address Code Statements
```

Executed codes of each level on each SCLP level and verified the results. Generated a report of the verification.

### 5.2 BUG REPORTS

The current implementation of SCLP had a handful of bugs which needed to be reported.

So far, 14 bugs have been found and reported.

Eg.

```
void main()
{
   int a[5];
   int b, c;

   a[0] = 5;
   a[1] = 4;
   a[2] = 3;
   a[3] = 2;
   a[4] = 1;

   c = 0;

   while(c < 5)
   {
         print "\n c: ";
         print c;
         print "\n";
         b = a[c];
         print b;
         print "\n";
         c = c+1;
   }
}
```

```
(spim) load "scodes/level6/code7.c.spim"
(spim) run

 c: 0
5

 c: 1
5

 c: 2
2

 c: 3
2

 c: 4
1
```

The code on the left does not produce expected output. The bug was speculated and the source of the bug was found through the symbol table output and the stack of SPIM simulator.

### 5.3 SCLP WEB PAGE CHANGES

The SCLP web page needed updates. It wasn't up-to-date with the newer version of SCLP. It had a couple of errors which needed to be rectified.

Point - [Features of the language supported by Sclp]

Line - [These sequences are limited to a length of 80]

Character sequences not limited to the length of 80. Accepts character sequences greater than the length of 80.

The changes and updates were reported, and the website was updated.

### 5.4 THOROUGH STUDY OF MIPS INSTRUCTIONS FOR SPIM

Studied MIPS instructions and registers.

Compiled numerous codes to investigate:

- how the registers are allocated.
- how data segment, register file, and the code segment used.

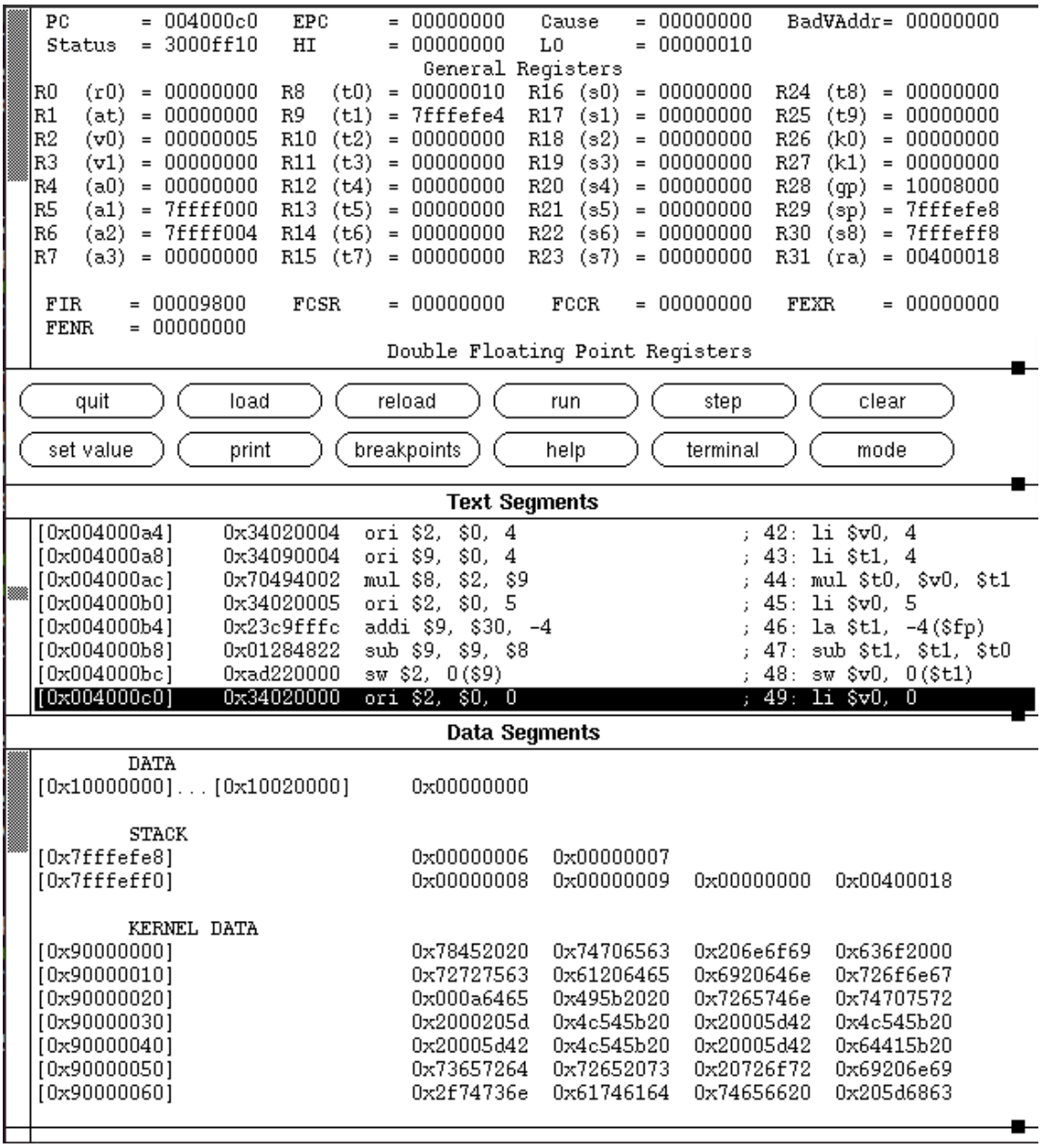



Fig 5.1- MIPS Instructions Working

### 5.5 FORMALIZED ASM INSTRUCTIONS FOR SPIM

The predecessors of the project had created specifications document for ASM which needed cleaning and deciding one proper syntax for all the specifications.

One universal syntax was decided to be used in all the specifications documents and ASM was formalized.

A few Load Instructions from the specifications document:

$$I \in \{mv, cmp, cf\} \qquad \{mv, cmp, cf\} : \tau_C^I$$
$$O \in \{int, dub, wrd, addr, lab, gpr, fpr, off\} \qquad \{lab\} : \tau_C^O \qquad \{gpr\} : \tau_R^O \qquad \{fpr\} : \tau_{FPR}^O$$

## 2 Load Instructions

$$[\textit{Load Integer Immediate}]\frac{S(li, o1, o2) : \tau_{mv}^S \qquad o1 : \tau_{gpr}^S \qquad o2 : \tau_{int}^S}{H; D; C; R \vdash S(li, o1, o2) \rightarrow H; D; C; R[o1 \mapsto o2] \qquad R : \tau_{gpr}^S} \quad (1)$$

$$[\textit{Load Float Immediate}]\frac{S(li.d, o1, o2) : \tau_{mv}^S \qquad o1 : \tau_{fpr}^S \qquad o2 : \tau_{dub}^S}{H; D; C; R \vdash S(li.d, o1, o2) \rightarrow H; D; C; R[o1 \mapsto o2] \qquad R : \tau_{fpr}^S} \quad (2)$$

$$[\textit{Load Word}]\frac{S(lw, o1, o2) : \tau_{mv}^S \qquad o1 : \tau_{gpr}^S \qquad o2 : \tau_{addr}^S}{H; D; C; R \vdash S(lw, o1, o2) \rightarrow H; D; C; R[o1 \mapsto D[o2 : o2 + 4] \qquad R : \tau_{gpr}^S} \quad (3)$$

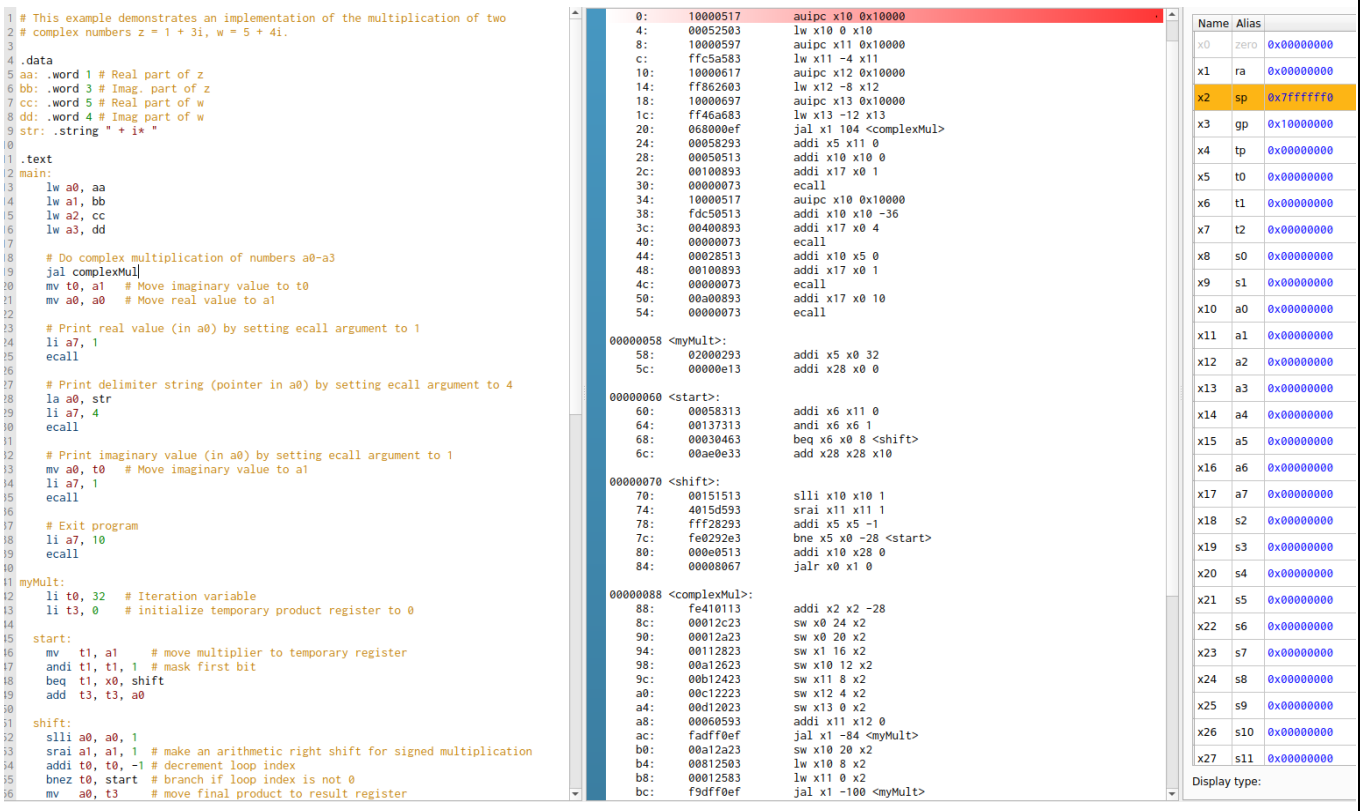
$$[\textit{Load Double}]\frac{S(l.d, o1, o2) : \tau_{mv}^S \qquad o1 : \tau_{fpr}^S \qquad o2 : \tau_{off}^S \qquad o3 : \tau_{gpr}^S}{H; D; C; R \vdash S(l.d, o1, o2) \rightarrow H; D; C; R[o1 \mapsto D[R[o3] + o2 : R[o3] + o2 + 8]] \qquad R : \tau_{fpr}^S} \quad (4)$$

$$[\textit{Load Address}]\frac{S(la, o1, o2) : \tau_{mv}^S \qquad o1 : \tau_{gpr}^S \qquad o2 : \tau_{addr}^S}{H; D; C; R \vdash S(la, o1, o2) \rightarrow H; D; C; R[o1 \mapsto o2] \qquad R : \tau_{gpr}^S} \quad (5)$$

### 5.6 THOROUGH STUDY OF RISC-V INSTRUCTIONS FOR RIPES

Studied the instructions and compared them with the MIPS instructions. Discussed the mapping of MIPS register and RISCV registers.

RIPES simulator showing the assembly editor, the assembled code.



```
# This example demonstrates an implementation of the multiplication of two
# complex numbers z = 1 + 3i, w = 5 + 4i.

.data
aa: .word 1 # Real part of z
bb: .word 3 # Imag. part of z
cc: .word 5 # Real part of w
dd: .word 4 # Imag part of w
str: .string " + i* "

.text
main:
    lw a0, aa
    lw a1, bb
    lw a2, cc
    lw a3, dd

    # Do complex multiplication of numbers a0-a3
    jal complexMul
    mv t0, a1   # Move imaginary value to t0
    mv a0, a0   # Move real value to a1

    # Print real value (in a0) by setting ecall argument to 1
    li a7, 1
    ecall

    # Print delimiter string (pointer in a0) by setting ecall argument to 4
    la a0, str
    li a7, 4
    ecall

    # Print imaginary value (in a0) by setting ecall argument to 1
    mv a0, t0   # Move imaginary value to a1
    li a7, 1
    ecall

    # Exit program
    li a7, 10
    ecall

myMult:
    li t0, 32   # Iteration variable
    li t3, 0    # initialize temporary product register to 0

  start:
    mv   t1, a1     # move multiplier to temporary register
    andi t1, t1, 1  # mask first bit
    beq  t1, x0, shift
    add  t3, t3, a0

  shift:
    slli a0, a0, 1
    srai a1, a1, 1  # make an arithmetic right shift for signed multiplication
    addi t0, t0, -1 # decrement loop index
    bnez t0, start  # branch if loop index is not 0
    mv   a0, t3     # move final product to result register
```

```
    0:      10000517        auipc x10 0x10000
    4:      00052503        lw x10 0 x10
    8:      10000597        auipc x11 0x10000
    c:      ffc5a583        lw x11 -4 x11
   10:      10000617        auipc x12 0x10000
   14:      ff862603        lw x12 -8 x12
   18:      10000697        auipc x13 0x10000
   1c:      ff46a683        lw x13 -12 x13
   20:      068000ef        jal x1 104 <complexMul>
   24:      00058293        addi x5 x11 0
   28:      00050513        addi x10 x10 0
   2c:      00100893        addi x17 x0 1
   30:      00000073        ecall
   34:      10000517        auipc x10 0x10000
   38:      fdc50513        addi x10 x10 -36
   3c:      00400893        addi x17 x0 4
   40:      00000073        ecall
   44:      00028513        addi x10 x5 0
   48:      00100893        addi x17 x0 1
   4c:      00000073        ecall
   50:      00a00893        addi x17 x0 10
   54:      00000073        ecall

00000058 <myMult>:
   58:      02000293        addi x5 x0 32
   5c:      00000e13        addi x28 x0 0

00000060 <start>:
   60:      00058313        addi x6 x11 0
   64:      00137313        andi x6 x6 1
   68:      00030463        beq x6 x0 8 <shift>
   6c:      00ae0e33        add x28 x28 x10

00000070 <shift>:
   70:      00151513        slli x10 x10 1
   74:      4015d593        srai x11 x11 1
   78:      fff28293        addi x5 x5 -1
   7c:      fe0292e3        bne x5 x0 -28 <start>
   80:      000e0513        addi x10 x28 0
   84:      00008067        jalr x0 x1 0

00000088 <complexMul>:
   88:      fe410113        addi x2 x2 -28
   8c:      00012c23        sw x0 24 x2
   90:      00012a23        sw x0 20 x2
   94:      00112823        sw x1 16 x2
   98:      00a12623        sw x10 12 x2
   9c:      00b12423        sw x11 8 x2
   a0:      00c12223        sw x12 4 x2
   a4:      00d12023        sw x13 0 x2
   a8:      00060593        addi x11 x12 0
   ac:      fadff0ef        jal x1 -84 <myMult>
   b0:      00a12a23        sw x10 20 x2
   b4:      00812503        lw x10 8 x2
   b8:      00012583        lw x11 0 x2
   bc:      f9dff0ef        jal x1 -100 <myMult>
```

| Name | Alias | |
|---|---|---|
| x0 | zero | 0x00000000 |
| x1 | ra | 0x00000000 |
| x2 | sp | 0x7ffffff0 |
| x3 | gp | 0x10000000 |
| x4 | tp | 0x00000000 |
| x5 | t0 | 0x00000000 |
| x6 | t1 | 0x00000000 |
| x7 | t2 | 0x00000000 |
| x8 | s0 | 0x00000000 |
| x9 | s1 | 0x00000000 |
| x10 | a0 | 0x00000000 |
| x11 | a1 | 0x00000000 |
| x12 | a2 | 0x00000000 |
| x13 | a3 | 0x00000000 |
| x14 | a4 | 0x00000000 |
| x15 | a5 | 0x00000000 |
| x16 | a6 | 0x00000000 |
| x17 | a7 | 0x00000000 |
| x18 | s2 | 0x00000000 |
| x19 | s3 | 0x00000000 |
| x20 | s4 | 0x00000000 |
| x21 | s5 | 0x00000000 |
| x22 | s6 | 0x00000000 |
| x23 | s7 | 0x00000000 |
| x24 | s8 | 0x00000000 |
| x25 | s9 | 0x00000000 |
| x26 | s10 | 0x00000000 |
| x27 | s11 | 0x00000000 |

Display type:

Fig 5.2, 5.3- Ripes Simulator showing the memory and the registers used.

Memory viewer

| Address | Word | Byte 0 | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|---|---|
| 0x00000070 | 0x00151513 | 0x13 | 0x15 | 0x15 | 0x00 |
| 0x0000006c | 0x00ae0e33 | 0x33 | 0x0e | 0xae | 0x00 |
| 0x00000068 | 0x00030463 | 0x63 | 0x04 | 0x03 | 0x00 |
| 0x00000064 | 0x00137313 | 0x13 | 0x73 | 0x13 | 0x00 |
| 0x00000060 | 0x00058313 | 0x13 | 0x83 | 0x05 | 0x00 |
| 0x0000005c | 0x00000e13 | 0x13 | 0x0e | 0x00 | 0x00 |
| 0x00000058 | 0x02000293 | 0x93 | 0x02 | 0x00 | 0x02 |
| 0x00000054 | 0x00000073 | 0x73 | 0x00 | 0x00 | 0x00 |
| 0x00000050 | 0x00a00893 | 0x93 | 0x08 | 0xa0 | 0x00 |
| 0x0000004c | 0x00000073 | 0x73 | 0x00 | 0x00 | 0x00 |
| 0x00000048 | 0x00100893 | 0x93 | 0x08 | 0x10 | 0x00 |
| 0x00000044 | 0x00028513 | 0x13 | 0x85 | 0x02 | 0x00 |
| 0x00000040 | 0x00000073 | 0x73 | 0x00 | 0x00 | 0x00 |
| 0x0000003c | 0x00400893 | 0x93 | 0x08 | 0x40 | 0x00 |
| 0x00000038 | 0xfdc50513 | 0x13 | 0x05 | 0xc5 | 0xfd |

Memory map

| Name | Size | Range |
|---|---|---|
| .text | 260 | 0x00000000 – 0x00000104 |
| .data | 23 | 0x10000000 – 0x10000017 |
| .bss | 0 | 0x11000000 – 0x11000000 |

## 5.7 FORMALIZED ASM SPECIFICATIONS FOR RIPES

Created specifications document for RISC-V instructions for RIPES

Formalixed specifications few compute instructions

## Compute Instructions

$$[Add\ ints]\frac{S(add,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{gpr}}{H;D;C;R\vdash S(add,o1,o2)\rightarrow H;D;C;R[o1\mapsto R[o2]+R[o3]] \quad R:\tau^S_{gpr}} \tag{6}$$

$$[Add\ ints]\frac{S(addi,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{int}}{H;D;C;R\vdash S(addi,o1,o2)\rightarrow H;D;C;R[o1\mapsto R[o2]+o3] \quad R:\tau^S_{gpr}} \tag{7}$$

$$[Sub\ ints]\frac{S(sub,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{gpr}}{H;D;C;R\vdash S(sub,o1,o2,o3)\rightarrow H;D;C;R[o1\mapsto R[o2]-R[o3]] \quad R:\tau^S_{gpr}} \tag{8}$$

$$[Sub\ ints]\frac{S(sub,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{int}}{H;D;C;R\vdash S(sub,o1,o2,o3)\rightarrow H;D;C;R[o1\mapsto R[o2]-o3] \quad R:\tau^S_{gpr}} \tag{9}$$

$$[Multiply\ ints]\frac{S(mul,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{gpr}}{H;D;C;R\vdash S(mul,o1,o2,o3)\rightarrow H;D;C;R[o1\mapsto R[o2]\times R[o3]] \quad R:\tau^S_{gpr}} \tag{10}$$

$$[Divide\ ints]\frac{S(div,o1,o2,o3):\tau^S_{cmp} \quad o1:\tau^S_{gpr} \quad o2:\tau^S_{gpr} \quad o3:\tau^S_{gpr}}{H;D;C;R\vdash S(div,o1,o2,o3)\rightarrow H;D;C;R[o1\mapsto R[o2]\div R[o3]] \quad R:\tau^S_{gpr}} \tag{11}$$

### 5.8 SPECIFICATIONS DOCUMENTS FOR RTL

Currently working on the specifications document for RTL for SPIM and for RIPES.

Eg. RTL specification of Li (Load Immediate) Instruction

$$[Load\ Integer\ Immediate]\frac{R(iLoad,o1,o2):\tau^R_{mv} \quad o1:\tau^R_{reg} \quad o2:\tau^R_{int}}{R\vdash S(iLoad,o1,o2)\rightarrow R[o1\mapsto o2] \quad R:\tau^S_{gpr}} \tag{12}$$

### 5.9 TRANSLATION RULES

#### 5.9.1. Original Translation Rules Syntax

- The original translation rules were written in SML(Standard Meta Language).
- In SML, each function deals with the translation of a group of similar RTL instructions to ASM instructions.
- These groupings have been per-defined in the SCLP code base and include Move, Control Flow, Compute, Label, Read, Write and NOP instructions.

```
fun gen_asm (rstmt: Compute_RTL_Stmt ) astmts: ASM_Stmt list =
       astmts : ASM_Stmt list = []
       case rstmt.op of
              | rtl_and | rtl_or | rtl_add | rtl_imm_add | rtl_sub | rtl_mult
              | rtl_div | rtl_add_d | rtl_sub_d | rtl_mult_d |  rtl_div_d
              | rtl_slt_d | rtl_sle_d | rtl_sgt_d | rtl_sge_d | rtl_sne_d | rtl_seq_d  =>
                     astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, (opd)rstmt.opd2, (opd)rstmt.res))

              | rtl_slt | rtl_sle | rtl_sgt | rtl_sge | rtl_sne | rtl_seq =>
                     astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, (opd)rstmt.opd2, NULL))

              | rtl_not | rtl_uminus | rtl_uminus_d =>
                     astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, NULL, (opd)rstmt.res))
```

Fig – Original Translation Rules Syntax

5.9.2. Problems with Original Syntax

- A newer, flexible syntax was expected as the above syntax was hard to read.
- The translation rules were supposed to be closer in syntax to the paper specifications written.
- The types of operands and other details to be mentioned in the translation rules itself

5.9.3. Changes Suggested in the Original Syntax

```
move:
  rule {rtl_move}

  antecedent
  {
    RTL(iLoad, o1, o2, o3):Move_RTL_Stmt
    o1 : reg
    o2 : null
    o3 : reg
  }

  consequent
  {
    ASM(li, o1, o2, o3):Move_ASM_Stmt
    o1 : reg
    o2 : null
    o3 : reg
  }
```

- Suggested to divide the instruction into two parts i.e antecedents and consequent.
- The antecedents part would include the name of the RTL instruction and the type of

operands(i.e above the line part of paper specifications).

- The consequents part would include the respective ASM instruction to the RTL instruction with the operands type specified as above.(i.e below part of the paper specifications).

```
gen_asm compute <astmts: ASM_Stmt list>:
  rule {rtl_and | rtl_or | rtl_add | rtl_imm_add | rtl_sub | rtl_mult
      | rtl_div | rtl_add_d | rtl_sub_d | rtl_mult_d |  rtl_div_d
      | rtl_slt_d | rtl_sle_d | rtl_sgt_d | rtl_sge_d | rtl_sne_d | rtl_seq_d}
    antecedant {rstmt: Compute_RTL_Stmt}
    consequent {astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, (opd)rstmt.opd2, (opd)rstmt.res))}

  rule {rtl_slt | rtl_sle | rtl_sgt | rtl_sge | rtl_sne | rtl_seq}
    antecedant {rstmt: Compute_RTL_Stmt}
    consequent {astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, (opd)rstmt.opd2, NULL))}

  rule {rtl_not | rtl_uminus | rtl_uminus_d}
    antecedant {rstmt: Compute_RTL_Stmt}
    consequent {astmts.append(ASM(get_asm_op(rstmt.op), (opd)rstmt.opd1, NULL, (opd)rstmt.res))}
```

- The above syntax was almost same to the previous one but only divided into antecedents and consequents part.
- The problem with this was also that it was hard to read and didn't specify the operands.

```
rule {rtl_and | rtl_or | rtl_add | rtl_imm_add | rtl_sub | rtl_mult
            | rtl_div | rtl_add_d | rtl_sub_d | rtl_mult_d |  rtl_div_d
            | rtl_slt_d | rtl_sle_d | rtl_sgt_d | rtl_sge_d | rtl_sne_d | rtl_seq_d}

    antecedent
    {
        RTL(iLoad, o1, o2, o3) = rstmt : Compute_RTL_Stmt
        o1 = rstmt.opd1 : RTL_Reg
        o2 = rstmt.opd2 : RTL_Reg
        o3 = rstmt.opd3 : RTL_Reg
    }

    consequent
    {
        ASM(li, o1, o2, o3) = astmt : ASM_Stmt
        o1 = rstmt.opd1 : ASM_Reg
        o2 = rstmt.opd2 : ASM_Reg
        o3 = rstmt.opd3 : ASM_Reg
    }
```

- The above syntax is also divided into antecedents and consequents.

```
{
    "move":{
        "rtl_move":{
            "antecedent":{
                "RTL(move, o1, o2)":"Move_RTL_Stmt",
                "o1":"RTL_Reg",
                "o2":"RTL_Reg"
            },
            "consequent":{
                "conse":{
                    "arrow":{
                        "RTL(move, o1, o2)":"Move_RTL_Stmt",
                        "ASM(move, o1, o2)":"ASM_Stmt"
                    }
                },
                "quent":[
                    {
                        "Reg_File[o2]":"Reg_File[o1]"
                    }
                ]
            }
        }
    }

}
```

- After the previously given syntax didn't solve the problem we thought of using JSON syntax as it was closer to paper specifications.
- Also, Parsing JSON becomes easier as it is a well-known format for carrying data.
- The above syntax thus is JSON with firstly the type of instruction (here "move"), then the RTL instruction for the same.
- Then the antecedent and consequent with the consequent further divided into 2 i.e. conse and quent.
- The conse part included the particular instruction in RTL and ASM.
- The quent part included the changes in the HDCR i.e. Heap, Data, Code, Register.

Problems with the above syntax-

- Has overhead, for e.g. the double quotes
- Translation and interpretation is not separated

Improvement in the above syntax

```
{

    "move":{
        "rtl_move":{
            "antecedent":{
                "RTL(move, o1, o2)":"Move_RTL_Stmt",
                "o1":"RTL_Reg",
                "o2":"RTL_Reg"
            },
            "consequent":{
                    "arrow":{
                        "RTL(move, o1, o2)":"Move_RTL_Stmt",
                        "ASM(move, o1, o2)":"ASM_Stmt"
                    }
            }
        }
    }

}
```

- The conse and quent part in the above syntax was replaced by arrow denoting the mapping of and RTL instruction to ASM.

5.9.4. Final Syntax for Translation Rules

```
1 {
2         compute{
3
4                 rtl_and | rtl_or | rtl_add {
5                         {
6                                 RTL(op; o1; o2; o3):Compute_RTL_Stmt,
7                                 o1:RTL_Reg,
8                                 o2:RTL_Reg,
9                                 o3:RTL_Reg
10                        },
11
12                        {
13                                RTL(op; o1; o2; o3):Compute_RTL_Stmt,
14                                ASM(opt; o1; o2; o3):ASM_Stmt
15                        }
16
17                }
18
19        }
20 }
```

- Differentiates between compiler state and program state
- Translation Rules, thus, corresponds to compilation
- IR Rules correspond to execution(or interpretation)

## 6.0 LEX SCRIPT TO PARSE TRANSLATION RULES

- The LEX script written parses the translation rules.

```
RTL_Reg {
                printf("%s\n", yytext);
                yylval.oclass = RTL_Reg;
                return OPDCLASS;
};

RTL_Int {
                printf("%s\n", yytext);
                yylval.oclass = RTL_Int;
                return OPDCLASS;
};

Compute  {
                printf("%s\n", yytext);
                yylval.itype = Compute;
                return INSTYPE;
};

OpCodes {
                printf("%s\n", yytext);
                yylval.itype = OpCodes;
                return INSTYPE;
}

rtl_and  {
                printf("%s\n", yytext);
                yylval.otype = rtl_and;
                return INSTNAME;
};
```

## 6.1 YACC SCRIPT TO PARSE TRANSLATION RULES

- The YACC script below contains grammar rules written in order to parse the paper specifications.

```
Start : '{' Instructions '}'
        ;

Instructions : Instructions ',' INSTYPE '{' InstList '}'       { $$ = process_Instructions($1, $3, $5); }
             | INSTYPE '{' InstList '}'                        { $$ = process_Instructions($1, $3); }
        ;

InstList : InstList ',' Item                                   { $$ = process_InstList($1, $3); }
             | Item                                            { $$ = process_InstList($1); }
             | TgtOpList                                       { $$ = process_InstList($1); }
        ;

Item : ItemBreak                                               { $$ = process_Item($1); }
        ;

ItemBreak : InstNameList '{' '{' Antecedent '}' ',' '{' Consequent '}' '}'            { $$ = process_ItemBreak($1, $8); }
        ;

TgtOpList : TgtOpList ',' TgtOps                               { $$ = process_TgtOpsList($1, $3); }
             | TgtOps                                          { $$ = process_TgtOpsList($1); }
        ;

TgtOps : TgtOpsBreak                                           { $$ = process_TgtOps($1); }
        ;

TgtOpsBreak : InstName ':' Values                                      { $$ = process_TgtOpsBreak($1, $3); }
        ;

Antecedent : SrcFormat ',' OperandList
        ;
```

## 6.2 TRANSLATOR GENERATOR

- Reads the input i.e. the RTL to ASM Translation Rules
- Gives ASMGen (Assembly Generator) as the output.

```
class Statement
{
        inter_rep ir;
        string * type_class;
        list<int> opds;

   public:
        Statement() { }
        Statement(inter_rep ir, list<int> opds, string * type_class) {
                this->ir=ir; this->opds=opds; this->type_class=type_class;
        }
        ~Statement();

        void print_stmt();
};
```

```
class Consequent {

        Statement * src;
        Statement * tgt;

        public:
                Consequent() {}
                Consequent(Statement * src, Statement * tgt) { this->src=src; this->tgt=tgt; }
                ~Consequent() {}
};

class Instruction {

        Code * inst_type_list;
        Consequent * cons;
        Expr_Attribute * key;
        Expr_Attribute * val;

        public:
                Instruction() {}
                Instruction(Code * inst_type_list, Consequent * cons) { this->inst_type_list=inst_type_list; this->cons=cons; }
                Instruction(Expr_Attribute * key, Expr_Attribute * val) { this->key=key; this->val=val; }
                ~Instruction() {}
};
```

Fig- Code Snippet of the Translator Generator.

## 6.3 **ASM GENERATOR**

```
#include<iostream>
#include<fstream>
#include<typeinfo>

using namespace std;

#include "common-headers.hh"

ASM_For_RTL & Compute_RTL_Stmt :: gen_asm() {

        list<ASM_Stmt*> & astmt = *new list <ASM_Stmt*>;
        RTL_Op rstmt_op = this->rtl_desc.get_op();

        switch (rstmt_op) {
                case rtl_add:
                case rtl_and:
                case rtl_or:
                {
                        CHECK_INVARIANT((this->rstmt_!= NULL),"operand cannot be null");
                        RTL_Opd* rstmt_= this->;
                        ASM_Opd* asm_= rstmt_->gen_asm_opd();

                        CHECK_INVARIANT((this->rstmt_opd1!= NULL),"operand cannot be null");
                        RTL_Opd* rstmt_opd1= this->opd1;
                        ASM_Opd* asm_opd1= rstmt_opd1->gen_asm_opd();

                        CHECK_INVARIANT((this->rstmt_opd2!= NULL),"operand cannot be null");
                        RTL_Opd* rstmt_opd2= this->opd2;
                        ASM_Opd* asm_opd2= rstmt_opd2->gen_asm_opd();

                        break;
                }

        ASM_For_RTL* for_rtl_astmts = new ASM_For_RTL(astmts, machine_desc_object.get_no_reg());
        return for_rtl_astmts;
```

# 6. TECHNOLOGY

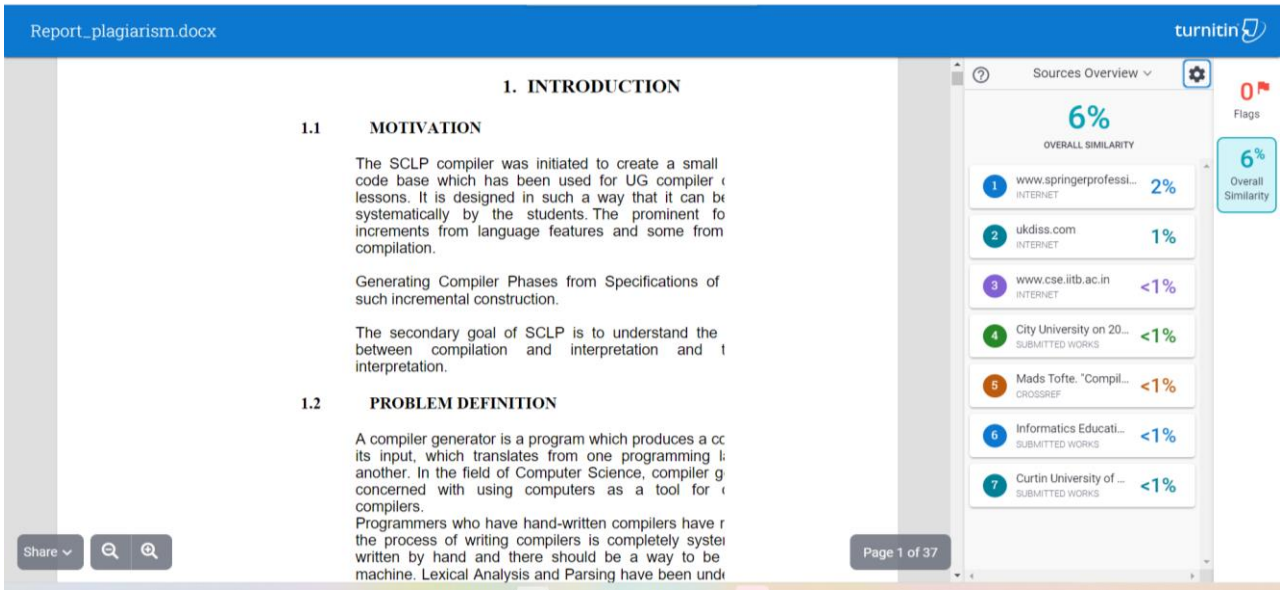
- To write the translator generators we are using the languages C++ and python.
- We are using lex and yaac as lexical analyzers and parser generators.
- Yacc is a computer program for the Unix operating system and Lex is a computer program that generates lexical analyzers.
- The current processor being used is MIPS processor and the simulator is SPIM.
- We are going to change the processor to RISCV and the simulator to RIPES.
- The software system LaTeX is used for document preparation.

# 7. CONCLUSIONS & FUTURE WORK

Thus, we conclude that our project is the development of SCLP compiler by automating the generation of IR phases like AST, TAC and RTL. The future work would be implementing generators that will construct interpreters and thus providing virtual machines for the IR phases i.e AST, TAC and RTL.

# 8. APPENDIX

## A: PLAGIARISM REPORT



## 9. REFERENCES


[1] A. V. Aho, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, J. D. Ullman, 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

[2] Tofte. M. 2012 compiler generators : what they can do , what they might do and what they will never do. Springer Science & Business Media.

[3] William Waite, "A Complete Specification of a Simple Compiler" Available- http://eli-project.sourceforge.net/pascal_html/pascal-.html#s2.1

[4] Dr.Uday Khedkar, "Sclp: A Language Processor for a Small C-like Language" Available- https://www.cse.iitb.ac.in/~uday/sclp-web/

[5] Central Connecticut State University, "Programmed Introduction to MIPS Assembly Language" Available- https://chortle.ccsu.edu/AssemblyTutorial/

[6] "SPIM Quick Reference", Available- http://www.cse.iitm.ac.in/~krishna/cs3300/spim_ref.html#traps

[7] Charles Prince, "MIPS IV Instruction Set", Available- https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

[8] James.R.Larus, "SPIM Documentation", Available- https://www.dsi.unive.it/~architet/LAB/spim.htm

[9] Portland State University, "Instruction Set Architecture", Available- https://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf

[10] Andrew Waterman, Krste Asanovi´, "The RISC-V Instruction Set Manual", Available- https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[11] "GitHub - mortbopet/Ripes: A graphical processor simulator and assembly editor for the RISC-V ISA", Available- https://github.com/mortbopet/Ripes