CS324/CS394

LANGUAGE PROCESSORS

RTL – Register Transfer Language

gcc intermediate representation

Project Report

under the guidance of

Prof. Uday Khedker

Date : 29th April, 06

Group Members:

Sanchit Kr. Garg (03d05005)
Prekshu Ajmera (03d05006)
Vipul G. Shingde (03d05013)

GCC Compilation process – where RTL stands?

Introduction

We don't usually program computers using strings of bits. Instead, we write all the instructions in a language we can understand and then have it translated into the appropriate sequence of 1s and 0s that the computer can understand. Everything you tell the computer to do needs to go through a translation process. Sometimes this translation is done on-the-fly (for instance, Perl scripts and spreadsheets) in a process called interpretation. Other times the translation is done once before the program starts (for instance, C or C++ applications) in a process called compilation. There are also hybrid approaches that are becoming popular which mix interpretation and compilation in a process called just-in-time compilation (JIT).

The Compilation Process

A compiler takes a program written in one language (source code) and transforms it into a semantically equivalent program in another language (object code). The best way of thinking about a compiler is that of a pipeline that gradually converts the program into different intermediate shapes until it reaches the final object code.

There are three major components to this pipeline. *The Front End*, which reads and validates the syntax of the input program. *The Middle End*, which analyzes and transforms the program into a more efficient form. Finally, *The Back End* is responsible for doing the final mapping into object code.

Front End

Computers are not very good at dealing with free form text, so the front end takes away all the spacing, comments and other formatting and converts the original program into a more concise form called intermediate representation (IR). These representations are nothing more than internal data structures that are much easier to manipulate than the original stream of text.

In general, compilers manipulate more than a single representation for the program. Initially, the Front End parses the input program and builds a data structure called Abstract Syntax Trees (AST), which represent each statement in a hierarchical structure. For eg, given the statement *x = y - z * 3*, its corresponding AST is shown in Fig 1.

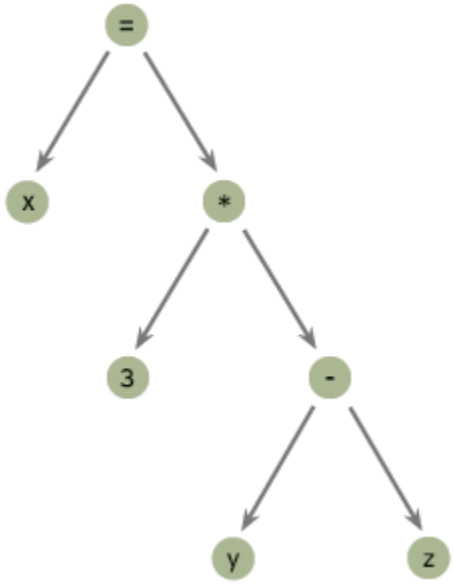


Fig 1. *Abstract Syntax Tree for x = y - z * 3*

The Front End is responsible for validating the syntactical structure of the input program, emitting most diagnostics about language conformance, creating internal data structures for data types and variables declared in the program, debugging information like file names and line numbers, and building the initial AST representation.

Middle End

With the intermediate representation built by the front end, the middle end proceeds to analyze and transform the program. All the transformations done here and in the back end usually have two goals: (a) make the object code run as fast as possible (performance optimizations), or (b) make the object code take as little space as possible (space optimizations). These optimizations are typically machine and target independent.

The Front End knows very little about what the program actually does. Optimization is possible when the compiler understands the flow of control in the program (control-flow analysis) and how the data is transformed as the program executes (data-flow analysis). Analysis of the control and data flow of the program allows the compiler to improve the runtime performance of the code. Many different optimizations are possible once the compiler understands the control and data flow of the program.

Back End

A final translation phase produces machine code for the target architecture. At this stage, the compiler needs to have very detailed knowledge about the hardware where the program will run. This means that the intermediate representation of the program is usually converted into a form resembling machine language. In this form, it is possible to apply transformations that take advantage of the target architecture. For instance:

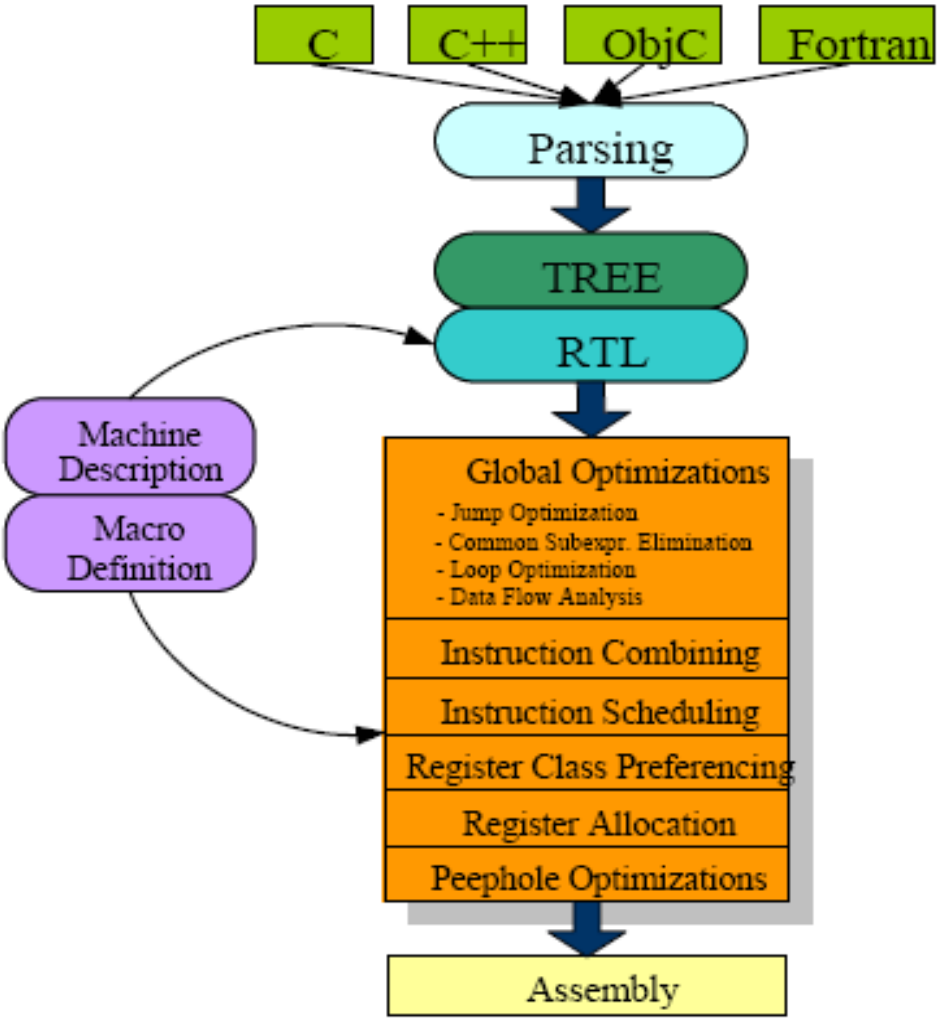
Register allocation

Tries to maximize the amount of program variables that are assigned to hardware registers instead of memory (registers are orders of magnitude faster than main memory).

Code scheduling

Takes advantage of the super-scalar features of modern processors to rearrange instructions so that multiple instructions are in different stages of execution simultaneously.

Even after final code generation, the resulting code is typically not ready to run as-is. GCC generate assembly code which is then fed into the assembler for object code generation. After object code has been generated, the linker is responsible for collecting all the different object files and libraries needed to build the final executable.



The Structure of GCC Compiler

RTL as Intermediate Representation

GCC is designed around two different IRs. One called tree, which is essentially the abstract syntax trees. The other IR is called **RTL (Register Transfer Language)**, is used by GCC to do optimizations and code generation. The following diagrams shows an overview of the compilation process in GCC.

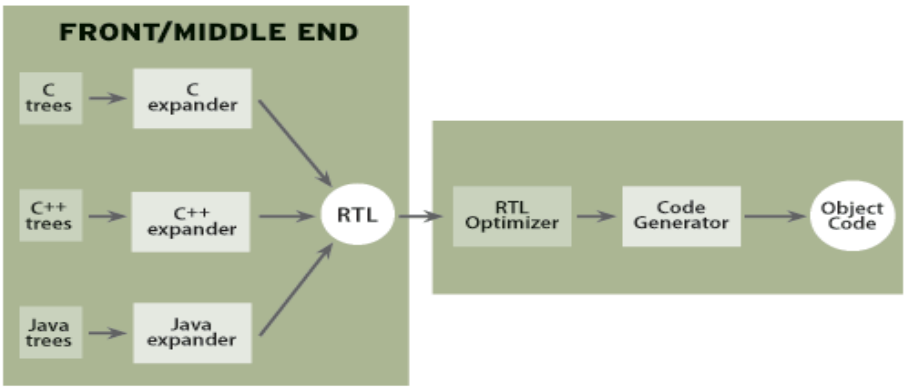
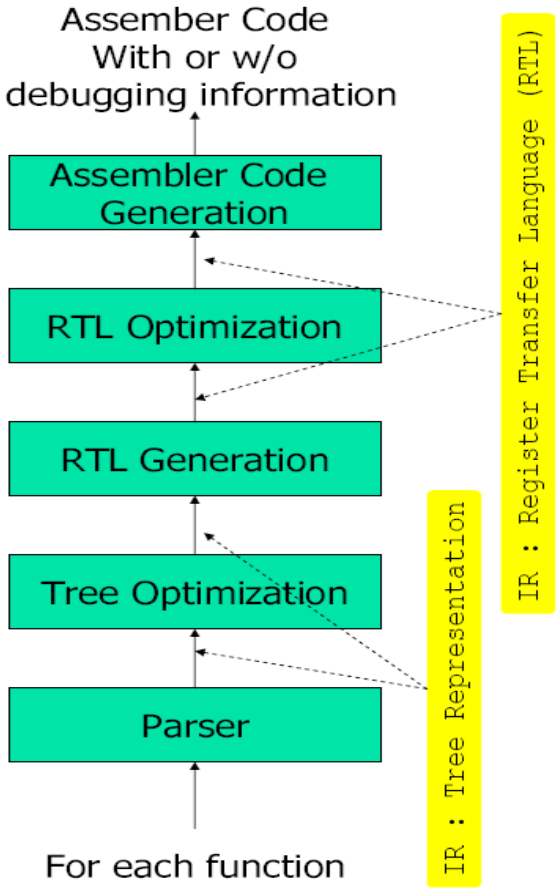


Fig 2. *Existing Compilation Process in GCC*

The following figure will prove out to be more descriptive:



Notice how the Front and Middle ends are sort of fused together. This is because, GCC does not operate on trees all that much. Following are the main conversions in the compiler -

- The front end reads the source code and builds a ***parse tree***.
- The ***parse tree*** is used to generate an ***RTL insn list*** based on ***named instruction patterns***.
- The ***insn list*** is matched against the ***RTL templates*** to produces ***assembler code***.

Need of RTL

Trees are mostly used as a stepping stone in the generation of RTL. Until recently, the only operation done on trees was function inlining. Most everything else was relegated to the RTL optimizer. In a way, this makes sense because from the Fig 2, one can notice that each language Front End generates its own version of trees. The C parser generates C trees, the C++ parser generates C++ trees, etc. Each version is different in its own way, so analyzes and optimizations on trees would require N different implementations, one for each front end. That is hardly scalable, so all the optimization work is done in RTL. The following are a few of the most popular optimization techniques used in standard optimizing compilers at middle end:

Algebraic simplifications

Expressions are simplified using algebraic properties of their operators and operands. For instance, i + 1 - i is converted to 1. Other properties like associativity, commutativity, and distributivity are also used to simplify expressions.

Constant folding

Expressions for which all operands are constant can be evaluated at compile time and replaced with their values. For instance, the expression a = 4 + 3 - 8 can be replaced with a = -1. This optimization yields best results when combined with constant propagation (see below).

Redundancy elimination

There are several techniques that deal with the elimination of redundant computations. Some of the more common ones include:

Loop-invariant code motion

Computations inside loops that produce the same result for every iteration are moved outside the loop.

Common sub-expression elimination

If an expression is computed more than once on a specific execution path and its operands are never modified, the repeated computations are replaced with the result computed in the first one.

Partial redundancy elimination

A computation is partially redundant if some execution path computes the expression more than once. This optimization adds and removes computations from execution paths to minimize the number of redundant computations in the program. It encompasses the effects of loop-invariant code motion and common sub-expression elimination.

RTL can be thought of as an assembly language for a machine with an infinite number of registers. The initial representation contains code that uses as many as soft registers as needed, only as it reaches its final representations like 35th mach (test.c.35.mach) only hard registers are actually used. Being a low-level representation, RTL works well for optimizations that are close to the target (for example, register allocation, delay slot optimizations, peepholes, etc). Furthermore, GCC offers a fairly flexible way for developers to describe their target machine when porting GCC to a new architecture. In fact, when it was initially implemented, GCC would generate RTL almost immediately, so it's only natural that most of the compiler is implemented around it.

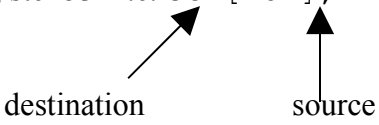
RTL - Semantics and statement constructs

Overview

- Used to describe insns (instructions)
- Written in LISP - like form

```
(set (mem:SI (reg: SI 54))
     (reg:SI 53))
```

Above RTL statement means “set memory pointed by register 54 with value in register 53” i.e. `st [r54], r53`



- `(plus:SI (reg:SI 55) (const_int -1))`

- Adds two 4-byte integer (SImode) operands.
- First operand is register
  - Register is also `4-byte integer`.
  - Register number is `55`.
- Second operand is a constant integer.
  - Value is `"-1"`
  - Mode is `VOIDmode` (not given)

Semantics of a RTL file

A good collection of basic RTL grammer is available at http://sinhagad.cse.iitb.ac.in/~gcc_docs/onlinedocs/GCC/PL.Group.Notes/gcc.3.3.3.structure/. Various operators and other RTL statement constructs are listed here.

Detailed information about particular RTL constructs can be seen at http://sinhagad.cse.iitb.ac.in/~gcc_docs/onlinedocs/GCC/internals/4.0.0/RTL.html#RTL.

However, in this report we have tried not to give information already provided by the above sources and this report consists mainly of our implementation of RTL interpreter and how program constructs like loops and other basic operations are written in RTL. We start with basic structure of a RTL file.

An RTL file is written in a linear fashion with statements depicting executable expressions in sequence. The RTL's are linear because compiler use them for optimization of code, as structure has already been identified in AST trees. The RTL representation of the code for a function is a doubly-linked chain of objects called *insns*. All *insns* have 3 mandatory field:

INSN_UID : Represents the unique id of *insn i*.

***PREV_INSN* (i)**: Accesses the chain pointer to the *insn* preceding *i*. If i is the first *insn*, this is a null pointer.

***NEXT_INSN* (i)**: Accesses the chain pointer to the *insn* following *i*. If *i* is the last *insn*, this is a null pointer.

Other then these fields, there are various others which depend on type of *insns*.
The main statement constructs in RTL or type of *insns* are :

- *note*
- *insn*
- *jump*
- *barrier*

note

note insns are used to represent additional debugging and declarative information. They are useful to mark beginning and end of functions, blocks like:

NOTE_INSN_BLOCK_BEG
NOTE_INSN_BLOCK_END

These types of notes indicate the position of the beginning and end of a level of scoping of variable names. As such, these statements do not execute anything .But they are printed symbolically when they appear in debugging dumps.
Example: *(note 3 11 18 NOTE_INSN_FUNCTION_BEG)*

insn

The expression code insn is used for instructions where we have to perform actual execution. Like *a = b + c* will be mapped to an insn expression. RTL file is mainly composed of *insn* instructions. They have additional fields like expression itself which is written in low level machine instructions with registers and memory locations, with a few basic operators like *plus, if_then_else, pre_dec*, etc.

Memory locations are accessed by *(mem:m address alias)* where *m* is the mode. Alias are given to the memory location. Similarly *(reg:m n)* stands for nth register, since registers are hard registers in this RTL file, its number is enough to tell its location. (Eg. 6 stands for base pointer).

Other then these, the last three fields provide information about the instruction and various other logs. RTL provides such information for better debugging, though it does not provide with any help while actual execuation, thats the reason they are eliminated at next level, where actual machine level instructions are generated.

The last three fields are:
INSN_CODE **:** Facilitates pattern matching in machine description.
LOG_LINKS **:** Gives information about dependencies between instructions within a basic block.
REG_NOTES **:** Gives miscellaneous information about the insn.

Example:

(insn 22 18 27 (clobber (reg/i:SI 0 ax)) -1 (nil)
(nil))

jump

The expression code jump_insn is used for instructions that may jump (or, more generally, may contain *label_ref* expressions). This construct is useful when we have to jump to some other insn of the RTL file which is not in sequence. In following cases they are used:

- If there is an instruction to return from the current function.

- In non-sequential program constructs like loops, if then else statements
- To apply label to some statement.

Example:

(jump_insn 38 37 39 (return) -1 (nil)
(nil))

Return insns are also counted as jumps, but since they do not refer to any labels, their ***JUMP_LABEL*** is ***NULL_RTX***.

Barrier

Barriers are placed in the instruction stream when control cannot flow past them. They are placed after unconditional jump instructions. They contain no information beyond the three standard fields.

Example:

(jump_insn 38 37 39 (return) -1 (nil)
(nil))
(barrier 39 38 31)

From RTL to Assembly

// C code //test.c

```
void funct()
{
      int a = 1;
      int b = a + 7;
}
```

// corresponding RTL code snippet // test.c.35.mach (generated by **gcc -da test.c**)

```
(insn 8 26 22 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])
        (const_int 1 [0x1])) 35 {*movsi_1} (nil)
    (nil))

// set the value at memory location pointed to by (base pointer - 8) to
value of 'a' i.e. 1

(insn 22 8 10 (set (reg:SI 0 ax [58])
        (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])) 35
{*movsi_1} (nil)
    (nil))
```

```
// set the value in ax register to the value at memory location pointed
to by (base pointer - 8) which is the value of 'a'

(insn 10 22 11 (parallel [
            (set (reg:SI 0 ax [58])
                (plus:SI (reg:SI 0 ax [58])
                    (const_int 7 [0x7])))
            (clobber (reg:CC 17 flags))
        ]) 139 {*addsi_1} (nil)
    (nil))

// set the value in the auxiliary register to sum of its current value
i.e. 1 and 7

(insn 11 10 12 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -4 [0xfffffffc])) [0 b+0 S4 A32])
        (reg:SI 0 ax [58])) 35 {*movsi_1} (nil)
    (expr_list:REG_EQUAL (plus:SI (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                    (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])
            (const_int 7 [0x7]))
        (nil)))

// set the value at memory location pointed to by (base pointer - 4)
i.e. value of 'b' to the value stored in auxiliary register.
```

// Machine code snippet // test.s (generated by **gcc -S test.c**)

```
      movl  $1, -8(%ebp)

      // move 1 to the memory location pointed to by (base pointer - 8)

      movl  -8(%ebp), %eax

      // move the value at memory location pointed to by
            (base pointer - 8) to the auxiliary register

      addl  $7, %eax

      // add 7 to the auxiliary register

      movl  %eax, -4(%ebp)

      // move value in auxiliary register to the memory location
           pointed to by (base pointer - 4) as the size of 'int'
           type is 4
```

Above example shows that RTL files instructions map to machine level code. In RTL files we often talk about memory and registers, how are these initialize!! What is meant by bp, sp and ax. Though this is not RTL specific, instead how gcc compiler implements and run its code, we would give a brief idea of stacks initialization to give an understanding of RTL files as a whole.

//diag of RTL stack
<diagram>
local parameters....
:
:
:
bp
parameters passed
:
:
:

bp is ***base pointer*** and it points to base of memory stack of the function. Each time a function is called the first thing it does is store the **bp** of its caller function, so that after it gets executed it could set **bp** to its previous position. Following is the corresponding RTL statement..

```
(insn/f 27 6 28 (set (mem:SI (pre_dec:SI (reg/f:SI 7 sp)) [0 S4 A8])
        (reg/f:SI 6 bp)) -1 (nil)
    (nil))
```

sp is ***stack pointer*** is global for all functions and points to the top of memory stack. So above RTL statement stores the original **bp** (which happens to be **bp** of caller function), on the top of stack (given by **sp**). If there are any parameters passed to this function, then those had already been pushed by caller function on this stack.
Going ahead, we would like to set the **bp** of present funciton now, which is done by next statement.

```
(insn/f 28 27 29 (set (reg/f:SI 6 bp)
        (reg/f:SI 7 sp)) -1 (nil)
    (nil))
```

Set **bp** to **sp**, i.e top of stack. Thus if any variable is declared now, we can access its memory by bp (which gives the address of memory where function starts) and offset of variable.

```
(insn 19 3 20 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -4 [0xfffffff8])) [0 var1+0 S4 A32])
        (const_int 0 [0x0])) 35 {*movsi_1} (nil)
    (nil))
```

Thus, if we declare var1 as 0 now, corresponding RTL construct will look like this. Here var1 is allocated memory space **bp** – 4 (size of **int**).

Note:
main() function is treated differently by GCC, and hence there are many more instructions before actual code. Thus to see stack initialization, RTL constructs for some other function should be generated.

After a basic overview, we now move on to how basic programming constructs like loops, if-else-then, switch are implemented in RTL. They will be covered while explaining 5 different levels of complexity, for which we have implemented scanner, parser and interpreter.

Implementation of ***Scanner***

Basic tokens were written to describe the keywords and identifiers. Separate tokens were used for all those keywords which were later used in parser and interpreter. All the other identifiers were grouped together so as to minimize the number of tokens, though we had to keep different tokens for operators which take different number of arguments. List of all such tokens and lex script is in *scan.l* file. All the RTL constructs which were identified in scanner can be looked at above mentioned references.

Implementation of ***Parser***

Since we used yacc tool, our parser is LR (bottom up) parser. Meaning to say the derivation is right sequential in reverse order, reading input from left to right. Various checks were possible in this parser like -
a) operators should take the same number and type of operands as they require
b) distinction between integer, float, char type of values
c) error at incomplete program constructs like loops.

But still the semantic analysis like order of instructions could not be checked and may lead to error during interpretation. The CFG is in *parse.y* file.

A small snippet of grammar implemented:

```
Start :-        InsnObject1             //Captures the sequential insn
                                        //statements  & thus the entire RTL
                                        //code
InsnObject1:-   InsnObject1 InsnObject
          |     InsnObject

InsnObject :-   '(' Insn ')'            //Captures a single insn statement
          |     '(' Barrier ')'
          |     '(' Note ')'
          |     '(' Jump ')'

                                        //Captures the entire note statement
Note :-         NOTE INT INT INT NOTECONST
          |     NOTE INT INT INT Vector NOTECONST

Barrier :-      BARRIER INT INT INT     //Captures the barrier statement

                                        //Captures the jump instruction
Jump :-         JUMP INT INT INT SideExp INT RestInsn
```

```
                                        //Captures the main insn instruction
Insn :-         INSN INT INT INT SideExp INT RestInsn
          |     INSN FLAG INT INT INT SideExp INT RestInsn
```

Objective: To create an ***interpreter*** for RTL code

An interpreter of RTL file takes a RTL file as an input and execute it!!! Elaborating, it scans and parse the file and then run it using actual registers and memory, as its original `.c` file would had run. However, its just a simulation, as it use C compiler beneath it. Interpreter would be useful if we want to know the results of specific changes made in RTL files.

Different ways:

1. Implement the interpreter in the parser itself
   Problems: a. Large grammar
   b. Jump statements, therefore not possible to implement.

2. Get the output from the parser in a txt file and then use a C/JAVA compiler/interpreter to interpret it.
   Problems: a. Using another compiler to implement a compiler
   b. Difficulties of file handling & its interpretation

3. Maintain a data structure in the parser which will be later used in the interpreter. (we are doing this)
   Advantages:
   a. During parsing only data structure is filled and expressions are evaluated after parsing has been completed. For evaluation of jump statements we don't need to invoke parser again and evaluation will be performed for previous statements in C Code itself.
   b. Large size of parsing grammar can be handled easily.

Our Implementation

Data structures used:

1. ***int reg_array[100];*** This array simulates registers .

2. ***struct mem_element mem_array[100]*;**
   To simulate memory stack. A function maps offsets of memory with respect to base pointer to index value of this array. Structure *mem_element* contains the value that is stored in corresponding memory location, along with the type of value (int/char).

3. ***struct statement fil[200];***

   In some sense this contains whole of the RTL file that has to be interpreted. Each element of 'fil' array maps to in instruction of the RTL file. Structure 'statement' contains various fields like id of the instruction, previous id, next id, type of instruction, and most importantly the expressions to be evaluated in this statement.

Requirements of data structure

This is the most important part in storing data structure, as it defines the complexity of C functions that will be later employed to evaluate these. The main issues that need to be dealt with are:

1. There should not be any ambiguity while evaluating data structure in C functions later. This means that later when we traverse the data structure there should not be any shift-reduce type of error.
2. Since yacc parser is bottom up parser therefore, care should be taken while entering the values in data structure, as the order in which they are actually entered may be different.
3. Data structure representation should be able to handle the necessary complexity that might be encountered during the interpretation of the program.

Our representation for expressions

We have represented the statement in the form of a tree using arrays .Each of the array element has following structure:

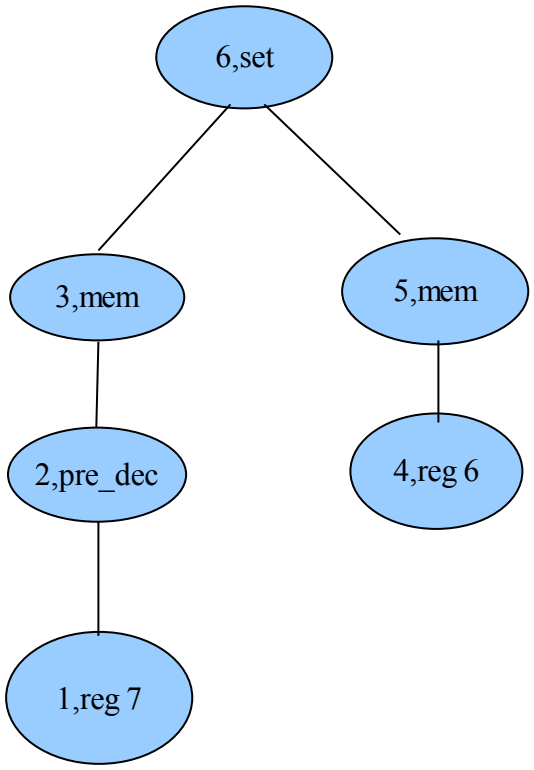
Attribute	***Description***
Code	Type of entry, like const_int or mem_addr
Value	Value associated with this entry, like register offset. In case of constants the actual value.
Mode	Represents mode(eg: SI) of corresponding operation
Operator	Operator if any.
Op1,Op2	These point to indices of its operands from which value of current entry will be evaluated.

Some of the above entries might be empty or partially filled depending upon the type of expression being stored.

Examples:
Following expression will be stored as shown:

```
(insn/f 32 16 33 (set (mem:SI (pre_dec:SI (reg/f:SI 7 sp)) [0 S4 A8])
        (reg/f:SI 6 bp)) -1 (nil)
    (nil))
```

Index No.	Type	mode	Operator	Value	Op1	Op2
1	Reg	SI		7		
2	Operator	SI	pre_dec		1	
3	Mem	SI			2	
4	Reg	SI		6		
5	Mem	SI			4	
6	Set				3	5



Tree depicting the expression storage for the given expression

Programming constructs in RTL

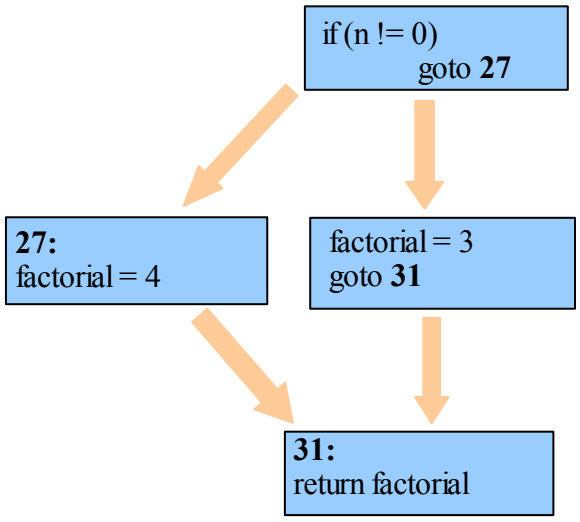
The interpreter for RTL files was completed in five stages. Each subsequent stage introduces some high level programming constructs, we will take same order to explain how they are implemented in RTL. For details about the levels please refer here: http://www.cse.iitb.ac.in/~uday/courses/cs324-05/gccProjects/node2.html

Level 0 is expression oriented, it uses only integer addition, the assignment operation and variables. In this level we learned allocating the memory for variables using stack and base pointers, also how addition and assignment operators are implemented. Much about these have already been explained in above examples. test cases 0 to 4 corresponds to this level.

Level 1 completes the basic capabilities of the language. It includes all other remaining arithmetic operators like division, subtraction, etc.. For these there is nothing new from implementation of addition operator. But ternary operator {() ? : } uses if-then-else construct, which is explained below.

If then Else statement

```
int main()
{
        int n;
        int factorial;

        if (n == 0)
                factorial = 3;
        else
                factorial = 4;
        return factorial;
}
```



.
.
.
//PROLOGUE PART

//function begins
```
(note 3 11 19 NOTE_INSN_FUNCTION_BEG)
```

//
check whether the value at memory location pointed to by (base pointer – 8) i.e. value of 'n' is > 0 and store the difference of that value and 0 in a register (17 here)
reg 17 is a dedicated flag register used for compare operations
//
```
(insn 19 3 20 (set (reg:CCZ 17 flags)
      (compare:CCZ (mem/i:SI (plus:SI (reg/f:SI 6 bp)
            (const_int -8 [0xfffffff8])) [0 n+0 S4 A32])
            (const_int 0 [0x0]))) 0 {*cmpsi_ccno_1} (nil)
    (nil))

(jump_insn 20 19 22 (set (pc)
      //if register no. 17 does not contain 0 then jump to label 27 else
      //jump to current program counter (pc) i.e. instruction number 22.
      (if_then_else (ne (reg:CCZ 17 flags)
            (const_int 0 [0x0]))
            (label_ref 27)
            (pc))) 340 {*jcc_1} (nil)
    (nil))

(note 22 20 24 [bb 2] NOTE_INSN_BASIC_BLOCK)
```

//
the if part of the c code ..
Set the value in memory location pointed to by (base pointer - 4) i.e. value of 'factorial' to 3.
//
```
(insn 24 22 25 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
      (const_int -4 [0xfffffffc])) [0 factorial+0 S4 A32])
      (const_int 3 [0x3])) 35 {*movsi_1} (nil)
    (nil))
```

//unconditional jump
//jump to label 31
```
(jump_insn 25 24 26 (set (pc)
        (label_ref 31)) 354 {jump} (nil)
    (nil))
```

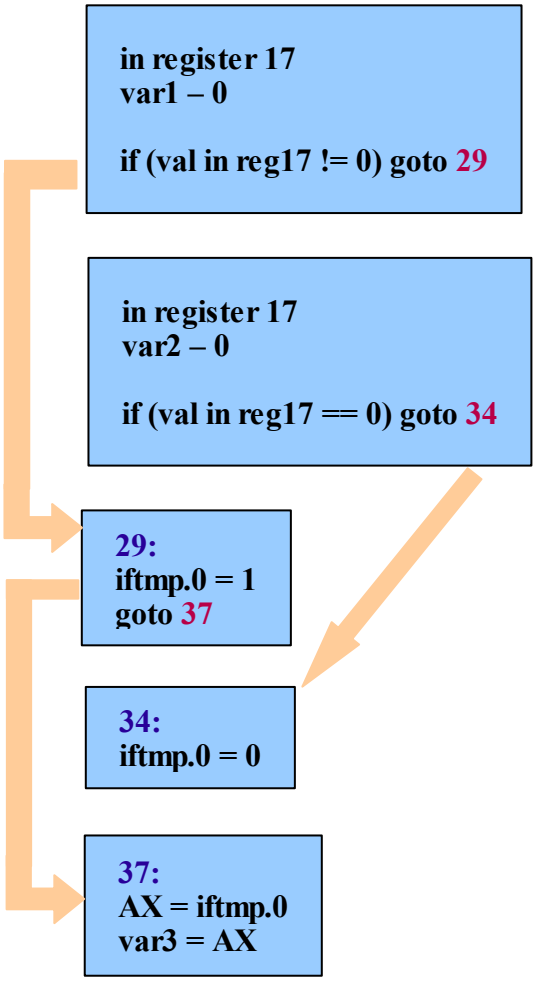
//control cannot flow past this barrier
```
(barrier 26 25 27)
```

```
//define label 27
(code_label 27 26 28 2 "" [1 uses])
(note 28 27 30 [bb 3] NOTE_INSN_BASIC_BLOCK)

//the else part of the c code
//set the value in memory location pointed to by (base pointer - 4) i.e. value of 'factorial'
//to 4
(insn 30 28 31 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -4 [0xfffffffc])) [0 factorial+0 S4 A32])
        (const_int 4 [0x4])) 35 {*movsi_1} (nil)
    (nil))

//define label 31
(code_label 31 30 32 4 "" [1 uses])
(note 32 31 34 [bb 4] NOTE_INSN_BASIC_BLOCK)


//set auxiliary register to the value in the memory location pointed to by (bp - 4) which is
the value of 'factorial'
//this is the return value of the function
(insn 34 32 38 (set (reg:SI 0 ax [orig:58 D.1168 ] [58])
        (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -4 [0xfffffffc])) [0 factorial+0 S4
A32])) 35 {*movsi_1} (nil)
    (nil))

//function ends
(note 38 34 47 NOTE_INSN_FUNCTION_END)

//EPILOGUE PART
.
.
.
```

Binary Operators

OR

```
int main()
{
    int var1 = 1;
    int var2 = 0;
    int var3=var1 || var2;
}
```




```
(note 3 11 19 NOTE_INSN_FUNCTION_BEG)
```

// Set the value in memory location pointed to by (base pointer - 12) i.e. value of 'var1' to 1. //

```
(insn 19 3 21 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -12 [0xfffffff4])) [0 var1+0 S4 A32])
        (const_int 1 [0x1])) 35 {*movsi_1} (nil)
    (nil))
```

```
/* Set the value in memory location pointed to by (base pointer - 8) i.e. value of
'var2' to 0. */
(insn 21 19 23 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -8 [0xfffffff8])) [0 var2+0 S4 A32])
        (const_int 0 [0x0])) 35 {*movsi_1} (nil)
    (nil))

/*
check whether the value at memory location pointed to by (base pointer – 12) i.e. value
of 'var1' is > 0 and store the difference of that value and 0 in register 17.
reg 17 is a dedicated flag register used for compare operations
*/
(insn 23 21 24 (set (reg:CCZ 17 flags)
        (compare:CCZ (mem/i:SI (plus:SI (reg/f:SI 6 bp)
           (const_int -12 [0xfffffff4])) [0 var1+0 S4 A32])
            (const_int 0 [0x0]))) 0 {*cmpsi_ccno_1} (nil)
    (nil))

(jump_insn 24 23 26 (set (pc)
      //if register no. 17 does not contain 0 then jump to label 29 else
      //jump to current program counter (pc) i.e. instruction number 26.
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 29)
            (pc))) 340 {*jcc_1} (nil)
    (nil))

(note 26 24 27 [bb 2] NOTE_INSN_BASIC_BLOCK)

/*
check whether the value at memory location pointed to by (base pointer – 8) i.e. value
of 'var2' is > 0 and store the difference of that value and 0 in register 17.
reg 17 is a dedicated flag register used for compare operations
*/
(insn 27 26 28 (set (reg:CCZ 17 flags)
        (compare:CCZ (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -8 [0xfffffff8])) [0 var2+0 S4 A32])
            (const_int 0 [0x0]))) 0 {*cmpsi_ccno_1} (nil)
    (nil))

(jump_insn 28 27 29 (set (pc)
      //if register no. 17 contains 0 then jump to label 34 else
      //jump to current program counter (pc) i.e. instruction number 29.
        (if_then_else (eq (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 34)
            (pc))) 340 {*jcc_1} (nil)
    (nil))
```

```
//define label 29
(code_label 29 28 30 2 "" [1 uses])

(note 30 29 31 [bb 3] NOTE_INSN_BASIC_BLOCK)

/*
Create a temporary variable 'iftmp.0' and set the value in memory location pointed to by
(base pointer - 20) i.e. value of  'iftmp.0' to 1.
*/
(insn 31 30 32 (set (mem:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -20 [0xffffffec])) [0 iftmp.0+0 S4 A8])
        (const_int 1 [0x1])) 35 {*movsi_1} (nil)
    (nil))

//jump to label 37
(jump_insn 32 31 33 (set (pc)
        (label_ref 37)) 354 {jump} (nil)
    (nil))
//control cannot flow past this barrier
(barrier 33 32 34)

//define label 34
(code_label 34 33 35 4 "" [1 uses])

(note 35 34 36 [bb 4] NOTE_INSN_BASIC_BLOCK)

//set the value in memory location pointed to by (base pointer - 20) i.e. value of  'iftmp.0'
to 0.
(insn 36 35 37 (set (mem:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -20 [0xffffffec])) [0 iftmp.0+0 S4 A8])
        (const_int 0 [0x0])) 35 {*movsi_1} (nil)
    (nil))

//define label 37
(code_label 37 36 38 5 "" [1 uses])

(note 38 37 54 [bb 5] NOTE_INSN_BASIC_BLOCK)

//set the value in the auxiliary register to the value in memory location pointed to by (bp –
20)  i.e. the value of 'iftmp.0'
(insn 54 38 39 (set (reg:SI 0 ax)
        (mem:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -20 [0xffffffec])) [0 iftmp.0+0 S4 A8]))
35 {*movsi_1} (nil)
    (nil))
```

```
//set the value in memory location pointed to by (base pointer - 4) i.e. value of  'var3' to the value present in auxiliary register.. */
(insn 39 54 40 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -4 [0xfffffffc])) [0 var3+0 S4 A32])
        (reg:SI 0 ax)) 35 {*movsi_1} (nil)
    (nil))
//function ends here
(note 40 39 44 NOTE_INSN_FUNCTION_END)
```

Level 2

adds basic data types: character, integer array and single precision floating point.

Character : `char var2='A';`

```
(insn 21 19 44 (set (mem/i:QI (plus:SI (reg/f:SI 6 bp)
                (const_int -5 [0xfffffffb])) [0 var2+0 S1 A8])
        (const_int 65 [0x41])) 44 {*movqi_1} (nil)
    (nil))
```

Above code intializes var2 to A. Note following:

- mode of mem is QI => quarter integer, because a character occupies just 1 byte as compared to 4 bytes of interger.
- We have allocated space of var2 as bp-5 => since character takes only one bit therefore its memory allocated space is just bp-5. (space till bp-4 was taken by some int var declared before it)
- const_int 65 => it stores ascii value of A

There is one more important thing to be noted. If we store just one character variable in memory it actually consumes 4 bytes. Meaning to say, even if we have space from bp-6 free in above example, still next int will be alloted space from bp-9 to bp-12. In between space are used only if we have more character variables to be stored.

Float : `float var3 = 2.345;`

```
(insn 46 23 26 (set (reg:SF 0 ax [70])
        (const_double:SF -1777042688 [0x96147b00]
2.34500002861022949218750e+0 [0x0.96147bp+2])) 60 {*movsf_1} (nil)
    (nil))

(insn 26 46 28 (set (mem/i:SF (plus:SI (reg/f:SI 6 bp)
                (const_int -4 [0xfffffffc])) [0 var3+0 S4 A32])
        (reg:SF 0 ax [70])) 60 {*movsf_1} (nil)
    (expr_list:REG_DEAD (reg:SF 0 ax [70])
        (nil)))
```

- Mode used is SF => Single Floating
- float uses 4 bytes bp-4
- While storing value in ax register, some arbit precision has been added. Also we need to identify a float constant as exponential form in our scanner.

Arrays:
int var2[10];
var2[0]=90;
var2[9]=1;

```
(insn 19 3 21 (set (mem/s/j:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -44 [0xffffffd4])) [0 var2+0 S4 A32])
        (const_int 90 [0x5a])) 35 {*movsi_1} (nil)
    (nil))

(insn 21 19 22 (set (mem/s/j:SI (plus:SI (reg/f:SI 6 bp)
                (const_int -8 [0xfffffff8])) [0 var2+36 S4 A32])
        (const_int 1 [0x1])) 35 {*movsi_1} (nil)
    (nil))
```

- Array size * width of one element is the memory size allocated.
- var2[0] is the last element, farthest from bp (bp-44)
- No error checking is done for array bound. For example following code is generated for var[10]=1;

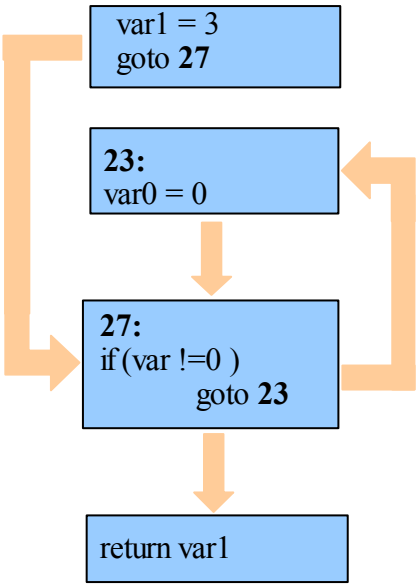
```
(insn 21 19 22 (set (mem/s/j:SI (plus:SI (reg/f:SI 6 bp)
      (const_int -4 [0xfffffffc])) [0 var2+40 S4 A32])
      (const_int 1 [0x1])) 35 {*movsi_1} (nil)
      (nil))
```

Level 3

Loops and if-then-else

WHILE LOOP

```
int main()
{
    int var1=3;
    while (var1)
    {
        int var0=0;
    }
    return var1;
}
```



//PROLOGUE PART

//Function begins

```
(note 3 11 19 NOTE_INSN_FUNCTION_BEG)
```

//set the value at memory location pointed to by (base pointer – 8) to value of 'var1' i.e. 3

```
(insn 19 3 47
    (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -8 [0xfffffff8])) [0 var1+0 S4 A32])
        (const_int 3 [0x1])) 35 {*movsi_1} (nil)
    (nil))
```

//While loop begins

```
(note 47 19 21 NOTE_INSN_LOOP_BEG)
```

```
//unconditional jump instruction
//jump to label 27
(jump_insn 21 47 22 (set (pc)
        (label_ref 27)) 349 {jump} (nil)
    (nil))

//placed after unconditional jump instruction
//control cannot flow past this barrier
(barrier 22 21 23)

//define label 23
(code_label 23 22 24 3 "" [1 uses])

(note 24 23 26 [bb 2] NOTE_INSN_BASIC_BLOCK)

//set the value at memory location pointed to by (base pointer – 4) to value of 'var0' i.e. 0
(insn 26 24 27
     (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
          (const_int -4 [0xfffffffc])) [0 var0+0 S4 A32])
          (const_int 0 [0x0])) 35 {*movsi_1} (nil)
     (nil))
//define label 27
(code_label 27 26 28 2 "" [1 uses])

(note 28 27 30 [bb 3] NOTE_INSN_BASIC_BLOCK)

/*
check whether the value at memory location pointed to by (base pointer – 8) ie value
of 'var1' is > 0 and store the difference of that value and 0 in a register (17 here).

Reg 17 is a dedicated flag register used for compare operations.
CCZ(mode used for integer comparisons against zero such that only zero flag is valid)
similarly, CCFPEQ(for floating point equality comparisons), CCNO (for integer
comparisons against zero that requires overflow flag to be unset)
*/
(insn 30 28 31 (set (reg:CCZ 17 flags)
        (compare:CCZ (mem/i:SI (plus:SI (reg/f:SI 6 bp)
          (const_int -8 [0xfffffff8])) [0 var1+0 S4 A32])
          (const_int 0 [0x0]))) 0 {*cmpsi_ccno_1} (nil)
    (nil))

(jump_insn 31 30 48 (set (pc)
      //if register no. 17 does not contain 0 then jump to label 23 else
      //jump to current program counter (pc) ie instruction number 48.
      (if_then_else (ne (reg:CCZ 17 flags)
            (const_int 0 [0x0]))
            (label_ref 23)
            (pc))) 335 {*jcc_1} (nil)
```

```
    (nil))
```

//End of while loop

```
(note 48 31 34 NOTE_INSN_LOOP_END)

(note 34 53 35 [bb 4] NOTE_INSN_BASIC_BLOCK)
```

//Set auxiliary register to the value of the memory location pointed to by (bp – 8) which is the value of 'var1'
//this is the return value of the function

```
(insn 35 33 39 (set (reg:SI 0 ax [orig:58 D.1171 ] [58])
        (mem/i:SI (plus:SI (reg/f:SI 6 bp)
        (const_int -8 [0xfffffff8])) [0 var1+0 S4 A32])) 35
{*movsi_1} (nil)
    (nil))
```

//End of function

```
(note 34 48 44 NOTE_INSN_FUNCTION_END)
```

//EPILOGUE PART

If-then-else is already covered in level1

Level 4

Functions

FUNCTION CALLS

```
int fact(int a)
{
    return a*a;
}

int main()
{
    int i=9;
    int var2=fact(9);
}
```

// Function fact begins

```
(note 3 2 6 NOTE_INSN_FUNCTION_BEG)

(note 6 3 27 [bb 1] NOTE_INSN_BASIC_BLOCK)
```

//decrement stack pointer by the length of a SImode value and use the result to address a Simode value which in this case is base pointer register.

```
(insn/f 27 6 28
    (set (mem:SI (pre_dec:SI (reg/f:SI 7 sp)) [0 S4 A8])
        (reg/f:SI 6 bp)) -1 (nil)
    (nil))
```

//set base pointer to stack pointer

```
(insn/f 28 27 29 (set (reg/f:SI 6 bp)
        (reg/f:SI 7 sp)) -1 (nil)
    (nil))

(note 29 28 8 NOTE_INSN_PROLOGUE_END)
```

//set auxiliary register to the value in the memory pointed to by (bp + 8) which is the value of 'a'

```
(insn 8 29 9 (set (reg:SI 0 ax [orig:60 a ] [60])
        (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                (const_int 8 [0x8])) [0 a+0 S4 A32])) 35
{*movsi_1} (nil)
    (nil))
```

//now set the auxiliary register to the result of the multiplication of value in auxiliary register and value in the memory pointed to by (bp+8) which is the value of 'a'

```
(insn 9 8 13 (parallel [
       (set (reg:SI 0 ax [orig:58 D.1119 ] [58])
           (mult:SI (reg:SI 0 ax [orig:60 a ] [60])
               (mem/i:SI (plus:SI (reg/f:SI 6 bp)
                   (const_int 8 [0x8])) [0 a+0 S4 A32])))
            (clobber (reg:CC 17 flags))
        ]) 173 {*mulsi3_1} (nil)
    (nil))

(note 13 9 22 NOTE_INSN_FUNCTION_END)
```

//value in auxiliary register ie result is needed at this point in the program
//the compiler will not attempt to delete previous instructions whose only effect is to store a value in 'ax'

```
(insn 22 13 30 (use (reg/i:SI 0 ax [ <result> ])) -1 (nil)
    (nil))

(note 30 22 31 NOTE_INSN_EPILOGUE_BEG)
```

//set base pointer to stack pointer and set sp to (sp + 4)

```
(insn 31 30 32 (parallel [
            (set (reg/f:SI 6 bp)
```

```
                (mem:SI (reg/f:SI 7 sp) [0 S4 A8]))
            (set (reg/f:SI 7 sp)
                (plus:SI (reg/f:SI 7 sp)
                    (const_int 4 [0x4])))
        ]) -1 (nil)
    (nil))
```

//jump to return address
//(return) is logically equivalent to (set (pc) (return))
//it represents a return from the current function

```
(jump_insn 32 31 33 (return) -1 (nil)
    (nil))

(barrier 33 32 26)

(note 26 33 0 NOTE_INSN_DELETED)
```

// Function main

```
(note 2 0 16 NOTE_INSN_DELETED)

(note 16 2 39 [bb 0] NOTE_INSN_BASIC_BLOCK)
```

//decrement stack pointer by the length of a SImode value and use the result to address a SImode value which in this case is base pointer register.

```
(insn/f 39 16 40
     (set (mem:SI (pre_dec:SI (reg/f:SI 7 sp)) [0 S4 A8])
        (reg/f:SI 6 bp)) -1 (nil)
    (nil))
```

//set base pointer to stack pointer

```
(insn/f 40 39 41 (set (reg/f:SI 6 bp)
        (reg/f:SI 7 sp)) -1 (nil)
    (nil))

(note 41 39 5 NOTE_INSN_PROLOGUE_END)

(note 3 11 19 NOTE_INSN_FUNCTION_BEG)
```

//set the memory pointed to by (bp - 8) ie value of 'i' to 9

```
(insn 19 3 21 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
            (const_int -8 [0xfffffff8])) [0 i+0 S4 A32])
        (const_int 9 [0x9])) 35 {*movsi_1} (nil)
    (nil))
```

//set the value of the memory pointed to by the stack pointer to 9 (arguement passed to fact)

```
(insn 21 19 22 (set (mem:SI (reg/f:SI 7 sp) [0 S4 A32])
        (const_int 9 [0x9])) 35 {*movsi_1} (nil)
    (nil))
```

//used for instructions that do function calls.
//It is important to distinguish these instructions because they imply that certain registers and memory locations may be altered unpredictably.

```
(call_insn 22 21 24

    (set (reg:SI 0 ax)
```

//the 1st argument represents the address label of the subroutine

```
     (call (mem:QI (symbol_ref:SI ("fact") [flags 0x3]
<function_decl 0x40397dec fact>) [0 S1 A8])
```

//the second argument to call represents the number of bytes of argument
//data being passed to the subroutine ie 4 in this case

```
            (const_int 4 [0x4])
     )
    ) 481 {*call_value_0} (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (nil))
```

//set the value at the memory poited to by (bp - 4) ie value of var2 to the value present in auxiliary register

```
(insn 24 22 25 (set (mem/i:SI (plus:SI (reg/f:SI 6 bp)
  (const_int -4 [0xfffffffc])) [0 var2+0 S4 A32])
  (reg:SI 0 ax [orig:58 D.1125 ] [58])) 35 {*movsi_1} (nil)
    (nil))

(note 25 24 29 NOTE_INSN_FUNCTION_END)
```

//EPILOGUE PART

Note: If we are not using a return value for a function, then the following instruction in caller function

```
(call_insn 22 21 24
    (set (reg:SI 0 ax)
     (call (mem:QI (symbol_ref:SI ("fact") [flags 0x3]
<function_decl 0x40397dec fact>) [0 S1 A8])
            (const_int 4 [0x4]))) 481 {*call_value_0} (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (nil))
```

is replaced by

```
(call_insn 22 21 24
     (call (mem:QI (symbol_ref:SI ("fact") [flags 0x3]
<function_decl 0x40397dec fact>) [0 S1 A8])
            (const_int 4 [0x4])) 481 {*call_value_0} (nil)
    (expr_list:REG_EH_REGION (const_int 0 [0x0])
        (nil))
    (nil))
```

This means that 'call' need not return some value to store, as it was doing in case above.

Limitations of RTL/Improvements in GCC-4.0.x

However, some analyzes and transformations need higher level information about the program that is not possible (or very difficult) to obtain from RTL (for example, array references, data types, references to program variables, control flow structures). In general, too many target features are exposed in RTL. For instance, if the target cannot handle more than 32-bit quantities in a register, the RTL representation splits values bigger than 32 bits into multiple values. Also, things like function calling conventions are exposed in detail in RTL, so a single function call may span multiple RTL instructions. All these details make it difficult to implement analyzes and optimizations that don't really need to concern themselves with target details. Over time, some of these transformations have been implemented in RTL, but the end result is code that is excessively convoluted, hard to maintain, and error prone.

Overcoming Architectural Limitations

Since the existing optimization framework in GCC was not flexible enough, several approaches were considered and settled on the idea of starting with the tree representation. Trees contain detailed information about data types, variables, and control structures of the original program. Optimizing the tree representation in GCC is appealing because (a) it facilitates the implementation of new analyzes and optimizations closer to the source and (b) it simplifies the work of the RTL optimizers, potentially speeding up the compilation process or improving the generated code.

The new framework is based on a compiler formalism called Static Single Assignment (SSA), which is a representation used internally by the compiler to keep track of how data flows through the program. This information is essential for the compiler to guide its optimization decisions. Given that the framework was implemented on top of the Tree data structure and uses SSA for data flow analysis, it is called Tree SSA.

Tree SSA

Although GCC parse trees provide very detailed information about the original program, they are not suitable for optimization for two main reasons:

1. **Lack of a common representation**

There is no single tree representation shared by all the front ends. Notice in Figure , how

every front end generates its own flavor of trees. A tree optimizer would have to be implemented once for every Front End supported by GCC. This would be a maintenance nightmare and would create a significant burden on every new Front End.

2. Structural complexity

The parse trees generated by the Front End are an almost exact representation of the original input program. As such, expressions can be combined in almost infinite ways & compilers, being the simple-minded automata they are, cannot easily cope with them. To overcome these limitations, two new tree-based intermediate representations called GENERIC and GIMPLE were introduced. GENERIC addresses the lack of a common tree representation among the various front ends. GIMPLE solves the complexity problems that facilitate the discovery of data and control flow in the program.

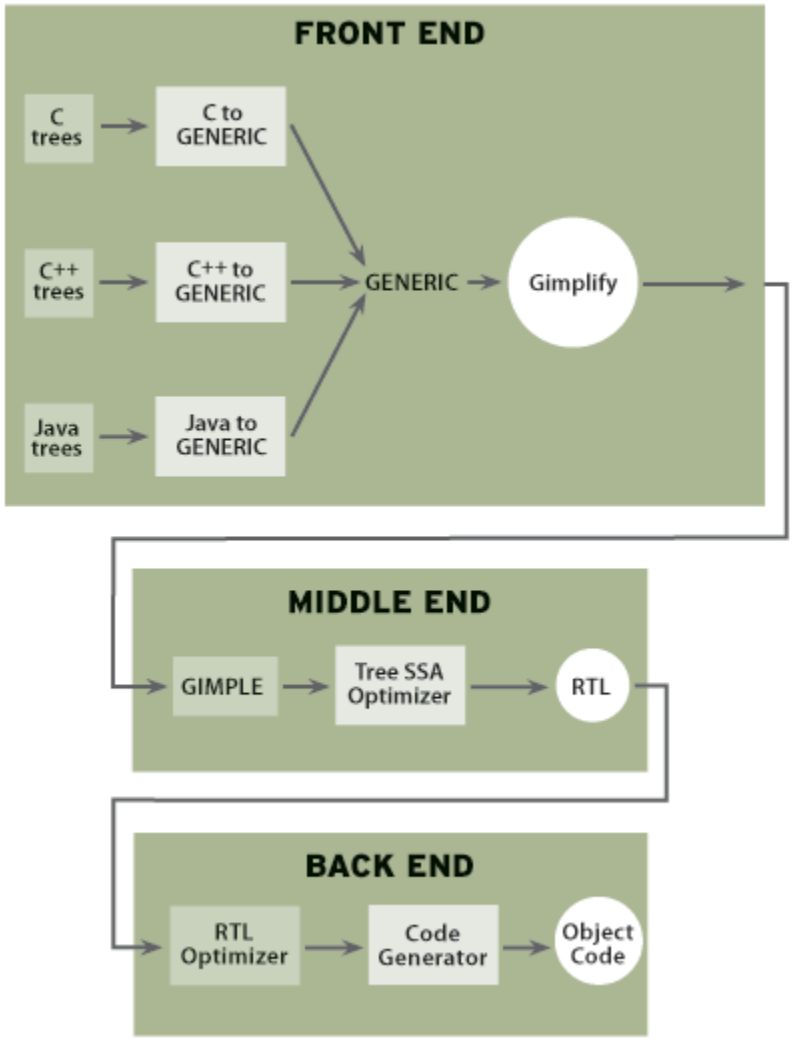


Fig 3. *GCC's Compilation Process with Tree SSA*

GENERIC and GIMPLE

Although some Front Ends share the same tree representation, there is no single representation used by all GCC Front Ends. Every Front End is responsible from translating its own parse trees directly into RTL. To address this problem, a new representation, named GENERIC, that is merely a superset of all the existing tree representations in GCC, was introduced. Instead of generating RTL, every Front End is responsible for converting its own parse trees into GENERIC trees. Since GENERIC trees are language-independent, all the semantics of the input language must be explicitly described by each Front End. Some of these semantics are translated in the "genericize" pass, while others are translated in the "gimplification" pass.

The conversion to GENERIC removes language dependencies from the program representation. However, GENERIC trees still are structurally complex, so the next step is to break them down into more manageable pieces. This new representation, called GIMPLE, is lexically identical to GENERIC but it imposes structural restrictions on statements. For instance, no expression can contain more than 3 operands, arguments to a function call can only be variables (i.e., function calls cannot be nested), the predicate of an if statement can only contain a single comparison, etc.

The following table consists the GENERIC and GIMPLE forms of the same code. Notice how the two versions are the same program, but the GIMPLE form of the program is more concise (albeit longer). This is the key property that makes things easier to the optimizer because it can make a variety of simplifying assumptions that, in the end, make for a compiler that is easier to maintain and improve.

```
GENERIC                          GIMPLE

if (foo (a + b, c))              t1 = a + b
  c = b++ / a                    t2 = foo (t1, c)
endif                            if (t2 != 0)
return c                           t3 = b
                                   b = b + 1
                                   c = t3 / a
                                 endif
                                 return c
```

Static Single Assignment Form

A program is in static single assignment (SSA) form when every variable is assigned to at only one location. The example below…

```
if (j != 7) {
  i = 1;
  f(j);
} else {
  i = 2;
}
printf ("%d", i + j)
```

… is not in **SSA** form, because there are two assignments to **`i`**. To transform the program into **SSA** form, you need to somehow arrange that there is only one assignment to **`i`**. To accomplish that, you must introduce a magical construct called the *phi operator*. The phi operator assigns a different value to **`i2`** depending on which edge was used to reach the bottom block. If the code traversal came from the block on the left, the value in **`i0`** is assigned to **`i2`**; otherwise, traversal came from the block on the right and the value of **`i1`** is assigned to **`i2`**.

Of course, real hardware does not have a phi operator, so trying to construct an SSA form representation from a machine-level representation (like RTL) is very hard. There are additional technical difficulties, like the fact that at the machine level it's sometimes hard to figure out what's being assigned. For example, many machines have instructions that assign to only part of a register; that can make it very difficult to obey the SSA constraint that each register be assigned only once.

The Static Single Assignment form is a representation that is laid on top of GIMPLE that exposes very explicitly the flow of data within the program. The central idea is that of versioning. Every time a variable X is assigned a new value, the compiler creates a new version of X and the next time that variable X is used, the compiler looks up the latest version of X and uses that. Notice that this representation is completely internal to the compiler, it is not something that shows up in the generated code nor could be observed by the debugger.

```
        GIMPLE program              SSA form

1       a = 3                       a_1 = 3
2       b = 9                       b_2 = 9
3       c = a + b                   c_3 = a_1 + b_2
4       a = b + 1                   a_4 = b_2 + 21
5       d = a + c                   d_5 = a_4 + c_3
6       return d                    return d_5
```

Notice that every assignment generates a new version number for the variable being modified. And every time a variable is used inside an expression, it always uses the latest version. So, the use of variable **a** in line 3 is modified to use **a_1**, while the use of a in line 5 uses **a_4**, instead.

So, why is this advantageous to the optimizer? Consider, for instance, a very common optimization called *constant propagation*. In this optimization, the compiler tries to compute at compile time as many expressions as possible using constant values. Since the program is in SSA form and all the variables have version numbers, the compiler can simply build arrays indexed by version number to keep track of constant values. Suppose we had an array **CST** for this purpose. In this case, **CST[1]** would hold **3** and **CST[2]** would hold **9**. So, when the compiler examines the statement in line 3, **c_3 = a_1 + b_2**, it can deduce that **c_3** must always be **12** (that is, it stores **12** in **CST[3]**). After visiting all the statements in this fashion, we end up with

```
1       a_1 = 3
2       b_2 = 9
```

```
3       c_3 = 12
4       a_4 = 30
5       d_5 = 42
6       return 42
```

And now, after having propagated all the constants, we find out that none of the statements in lines 1 through 5 are really useful at all. This program merely computes and returns the value 42.

Now, what happens when it is not possible to determine what the latest version of a variable is? Real programs are seldom straight line code, there are loops and conditionals that alter the flow of control. For instance, in the following program it may be impossible for the compiler to determine what branch of the conditional will be taken at runtime:

```
x = foo ()
if (x > 10)
  a = 92
else
  a = 23
endif
return a
```

When the compiler tries to convert the program above into **SSA** form, it does not know which of the two versions for variable **a** to use. To solve this ambiguity, the compiler creates a third version for variable **a** that is the *merge* of the two conflicting versions. This merge operation is known as a *PHI function*:

```
x_1 = foo ()
if (x_1 > 10)
  a_2 = 92
else
  a_3 = 23
endif
a_4 = PHI (a_2, a_3)
return a_4
```

Since the compiler does not know in advance which of **a_2** or **a_3** will be returned, it creates **a_4** as the merge of the two. This indicates to the optimizers that **a_4** can be one of the other two versions, but we don't know which.

Tree SSA provides a new optimization framework to make it possible for GCC to implement high-level analyzes and optimizations. The initial implementation is available with the release of GCC 4.0. This framework is still in active development, the next release will include several new features including pointer and array bound checking (-fmudflap), support for Fortran 90, auto vectorization, various high-level loop transformations and most traditional scalar optimization passes.

Optimizations

In all versions of GCC before GCC 4.0.0, all “interesting” optimizations occurred in RTL. First, the compiler would perform high-level optimizations, such as moving a computation out of a loop if that computation produced the same result every time. Then,

the compiler performed low-level optimizations, including instruction scheduling, in which a pre selected sequence of machine instructions are ordered so that they run as fast as possible on a particular processor.

Now, in GCC 4.0.0, many high-level optimizations are now performed at the Tree level rather than at the RTL level. The next section explains why.

Some basic problems:

It was impractical or impossible to do as much optimization as was desired on the RTL representation. The most fundamental data structure used for optimization is the control flow graph (CFG). The CFG is a directed graph in which each node is a section of code that has only one entry point and only one exit point. In other words, once you start executing the first instruction in the block, you are guaranteed to execute every instruction in the block. However, there may be many ways to get to the beginning of the block, and many places to go after you leave the block. There is an edge from one block to another if you can reach the second block directly after leaving the first block.

For example, in the CFG for the code snippet…

```
if (j != 7) {
        i = 1;
        f(j);
} else {
        i = 2;
}

printf ("%d", i + j);
```

… there are two successor blocks to the j!=7 block, and two predecessor blocks to the printf block. Each "branch" of the conditional has only one predecessor block and only one successor block. The CFG data structure is incredibly powerful, because the compiler can use it to figure out how to move computations around and how to simplify them.

For example, if the compiler can see that no matter how you reach a particular block, the variable i has the value 3, the compiler can replace i with 3 in that block, even if it is possible for i to have other values in other blocks. Replacing a variable with a constant may then may permit further simplifications.

Or, if there is no way to reach a particular block, then that code can be eliminated completely. Or, as another example, if the same computation is performed in two blocks and every path that reaches either one of the blocks goes through some predecessor block, then the computation can be moved to the predecessor block, an optimization that shrinks the size of the program. And there are many more optimizations that can be performed on a control flow graph.

Unfortunately, building an accurate control flow graph at the RTL level is somewhat difficult. For example, at the assembly code level, you sometimes see "conditionally executed" instructions which, as their name implies, may or may not actually be executed depending on the state of some predicate register. The prologue code for functions in shared libraries often contains code that appears to be a control-flow transfer, but which

is not really such a transfer. These kinds of details reduce the accuracy of the control flow graph. It's possible to build a CFG for RTL, and GCC does in fact do so, but it's difficult.

A fundamental tenet of compiler design is that the more the compiler knows, the better the code it can generate. Every single fact that the compiler can deduce about the program can result in better code. But a lot of information is lost in translation to the RTL level. Therefore, doing optimizations on RTL necessary results in inferior code.

In particular, the RTL representation has no notion of "type," in the sense that programmers usually use that term. At the RTL level, the only types are things like "16-bit integer" or "32-bit floating-point value." There's no easy way to even tell whether a particular value is a signed integer or an unsigned integer, and there's no way to tell that three consecutive 32-bit floating-point values stored in memory are actually a structure representing a point in three-dimensional euclidean space. For the underlying processor, it's just data, and the same is true for RTL. (Processors do not know that a particular piece of data is signed or unsigned; the assembly code simply tells them whether to perform a signed or unsigned operation on the data.) Type information is just one kind of information that is unavailable at the RTL level, despite being available in the source program and useful for optimization.

Another example is knowledge about what memory is part of an array. At the RTL level, array accesses are transformed into pointer dereferences. As a result, it may no longer be obvious to the compiler that a series of pointer dereferences actually refer to the same array, therefore resulting in inferior code.

Another problem is that by the time RTL has been generated, the compiler has committed to "code shape." For example, the compiler may have decided that some structure variables must live in memory on the stack, because they are too big to fit in registers. Even if the compiler can subsequently prove that part of the variable is not used by the program, and that, therefore, the value could fit in a register, it's very difficult to find and correct all the places in the function that assume that the variable will be on the stack.

Finally, it's nearly impossible to convert the RTL representation into static single assignment (SSA) form. Most modern optimization algorithms work most efficiently and most effectively on programs in SSA form.

Improvements in GCC 4.0.x

There are a number of ways in which 4.0.x is a definitive improvement over any previous release of GCC.
For example, GCC now performs an optimization called ***scalar replacement of aggregates***, which is extremely important for C++ programs. In a program that uses small structures, like…

```
struct Point {
  double x, y, z;
};
```

… previous versions of GCC would have insisted on storing such a structure in memory and would copy the structure from place to place whenever it was passed as an argument to functions. Now, GCC is capable of splitting up the structure into separate "scalar" variables, so that they can be handled independently. As a result, the contents of the structure are much more likely to be placed into registers and much more likely to be copied. This kind of improvement is known as "***reducing the abstraction penalty***" because it allows programmers to write shorter, more abstract programs without sacrificing performance.

Another new optimization in GCC 4.0.x is ***partial redundancy elimination (PRE)***. This optimization looks for ways to move code into locations in the CFG that allow duplicate code appearing in multiple subsequent blocks to be eliminated. There are also a number of new optimizations that provide better optimization of loops. (Optimizing loops is very important, because most programs spend most of their time in relatively few loops. If you can make those loops more efficient, you make the whole program more efficient.)

As a rule of thumb, CPU bound programs might hope to see as much as a 10 percent improvement in performance when using GCC 4.x.

TO BE ADDED IN THIS REPORT

1. RTL explaination for &&, <<, switch statement (using integer & char in case)
2. What is there in PROLOGUE & EPILOGUE parts of a RTL file ?
3. Explanation for implementation of various levels of interpreter.
4. Brief overview of RTL Activation record

References:

[1] http://altair.snu.ac.kr/newhome/kr/course/introduction-to-compiler/2003/portingGcc.pdf
[2] http://www.linux-mag.com/index.php?option=com_content&task=view&id=2438&Itemid=2346
[3] http://www.redhat.com/magazine/002dec04/features/gcc/
[4] http://sinhagad.cse.iitb.ac.in/~gcc_docs/onlinedocs/GCC/internals/4.0.0/