Introduction to RTL

GCC Resource Center

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

March 2010

Outline

- RTL: The Overall Perspective
- RTL: An External View
- RTL: An Internal View
- RTL: An Example Program to Manipulate RTL

Part 1

RTL: The Overall Perspective

What is RTL ?

RTL = Register Transfer Language

Assembly language for an abstract machine with infinite registers

Why RTL?

A lot of work in the back-end depends on RTL. Like,

- Low level optimizations like loop optimization, loop dependence, common subexpression elimination, etc
- Instruction scheduling
- Register Allocation
- Register Movement

Why RTL?

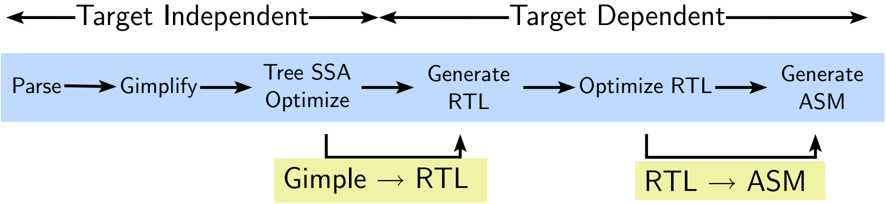
For tasks such as those, RTL supports many low level features, like,

- Register classes
- Memory addressing modes
- Word sizes and types
- Compare and branch instructions
- Calling Conventions
- Bitfield operations

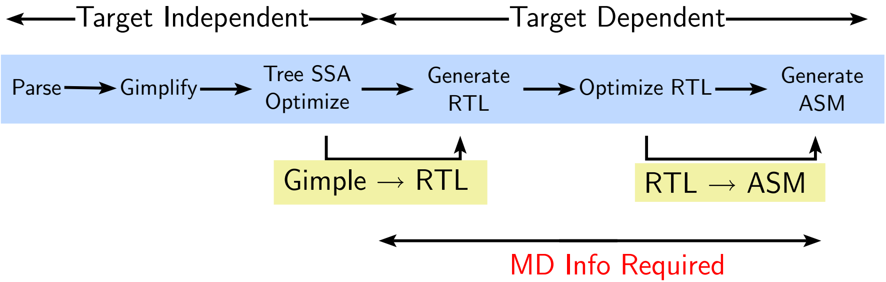
The Dual Role of RTL

- For specifying machine descriptions
  Machine description constructs:
  - define_insn, define_expand, match_operand
- For representing program during compilation
  IR constructs
  - insn, jump_insn, code_label, note, barrier

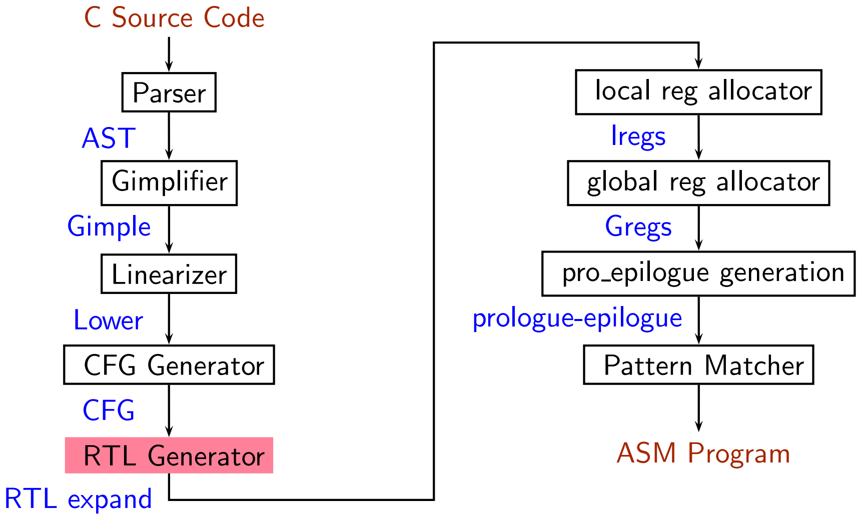
Role of Machine Descriptions in Translation



Role of Machine Descriptions in Translation



Generating RTL IR During Compilation



A Target Instruction in Machine Descriptions

```
(define_insn
    "movsi"
     (set
       (match_operand 0 "register_operand" "r")
       (match_operand 1 "const_int_operand" "k")
       )
    "" /*  C boolean expression, if required */
    "li %0, %1"
)
```

A Target Instruction in Machine Descriptions

Define instruction pattern

Standard Pattern Name

```
(define_insn
   "movsi"
   (set
     (match_operand 0 "register_operand" "r")
     (match_operand 1 "const_int_operand" "k")
     )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

RTL Expression (RTX):
Semantics of target instruction

target asm inst. =
Concrete syntax for RTX

A Target Instruction in Machine Descriptions

RTL operator

MD constructs

```
(define_insn
   "movsi"
   (set
     (match_operand 0 "register_operand" "r")
     (match_operand 1 "const_int_operand" "k")
     )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

Predicates

Constraints

An Example of Translation

```
(define_insn
   "movsi"
    (set
      (match_operand 0 "register_operand" "r")
      (match_operand 1 "const_int_operand" "k")
      )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

An Example of Translation

```
(define_insn
   "movsi"
    (set
      (match_operand 0 "register_operand" "r")
      (match_operand 1 "const_int_operand" "k")
      )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

Development

`D.1283 = 10;` ⇒

```
(set
    (reg:SI 58 [D.1283])
    (const_int 10:  [0xa])
)
```

⇒ `li $t0, 10`

Use

The Essence of Retargetability

When are the machine descriptions read?

The Essence of Retargetability

When are the machine descriptions read?

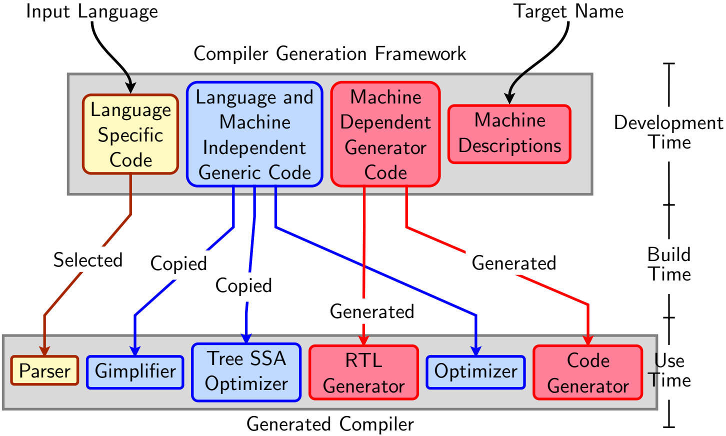
- During the build process

The Essence of Retargetability

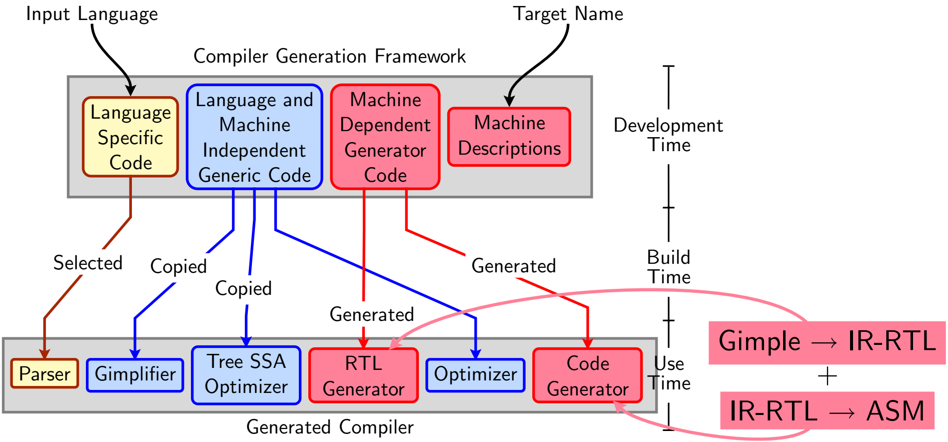
When are the machine descriptions read?

- During the build process
- When a program is compiled by gcc the information gleaned from machine descriptions is consulted

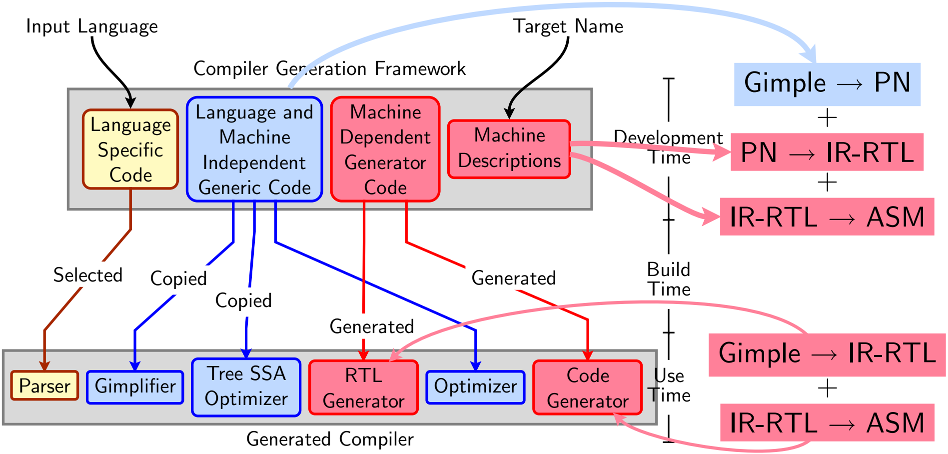
Retargetability Mechanism of GCC



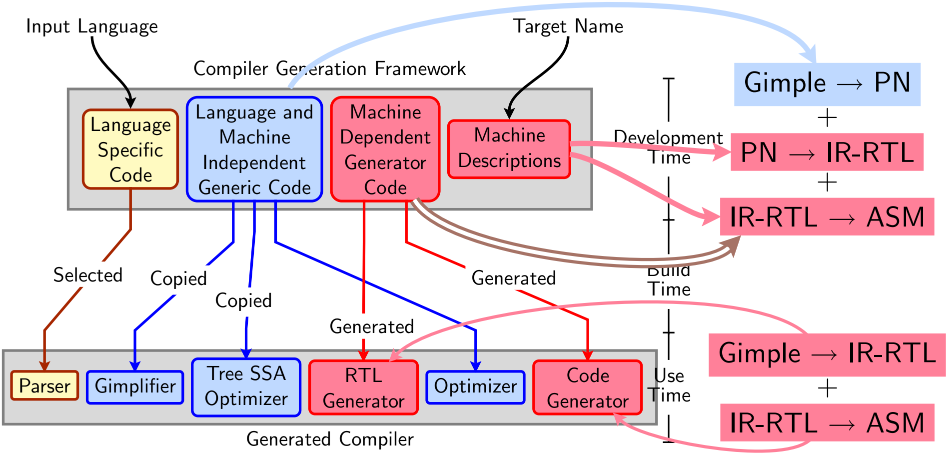
Retargetability Mechanism of GCC



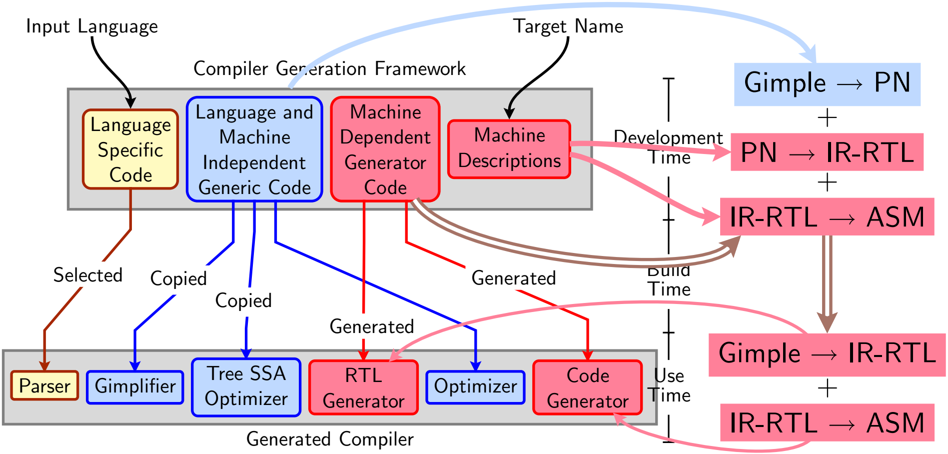
Retargetability Mechanism of GCC



Retargetability Mechanism of GCC



Retargetability Mechanism of GCC



Part 2

RTL: An External View

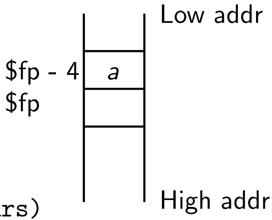
RTL for i386: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 10 (parallel [
    (set (mem/c/i:SI (plus:SI
             (reg/f:SI 54 virtual-stack-vars)
             (const int -4 [...])) [...])
        (plus:SI
             (mem/c/i:SI (plus:SI
                 (reg/f:SI 54 virtual-stack-vars)
                 (const int -4 [...])) [...])
             (const int 1 [...])))
   (clobber (reg:CC 17 flags))
   ]) -1 (nil))
```



Plus operation computes `$fp - 4` *as the address of variable* a

RTL for i386: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 10 (parallel[
    ( set (mem/c/i:SI (plus:SI
              (reg/f:SI 54 virtual-stack-vars)
              (const int -4 [...])) [...])
          (plus:SI
              (mem/c/i:SI (plus:SI
                  (reg/f:SI 54 virtual-stack-vars)
                  (const int -4 [...])) [...])
              (const int 1 [...])))
   (clobber (reg:CC 17 flags))
   ]) -1 (nil))
```

Set denotes assignment

RTL for i386: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 10 (parallel [
    ( set (mem/c/i:SI (plus:SI
                (reg/f:SI 54 virtual-stack-vars)
                (const int -4 [...])) [...])
          (plus:SI
                (mem/c/i:SI (plus:SI
                    (reg/f:SI 54 virtual-stack-vars)
                    (const int -4 [...])) [...])
                (const int 1 [...])))
   (clobber (reg:CC 17 flags))
   ]) -1 (nil))
```

1 is added to variable a

RTL for i386: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

stack(\$fp - 4) = stack(\$fp - 4) + 1
|| flags=?

```
(insn 12 11 10 (parallel [
   ( set (mem/c/i:SI (plus:SI
            (reg/f:SI 54 virtual-stack-vars)
            (const int -4 [...])) [...])
        (plus:SI
            (mem/c/i:SI (plus:SI
                (reg/f:SI 54 virtual-stack-vars)
                (const int -4 [...])) [...])
            (const int 1 [...])))
  (clobber (reg:CC 17 flags))
  ]) -1 (nil))
```

Condition Code register is clobbered to record possible side effect of plus

Flags in RTL Expressions

Meanings of some of the common flags

`/c`	memory reference that does not trap
`/i`	scalar that is not part of an aggregate
`/f`	register that holds a pointer

RTL for spim: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

RTL for spim: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

RTL for spim: Examples

Translation of $a = a + 1$

Dump file: `test.c.131r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

RTL for i386: Examples

What does this represent?

```
(jump_insn 15 14 16 4 p1.c:6 (set (pc)
     (if_then_else (lt (reg:CCGC 17 flags)
         (const_int 0 [0x0]))
        (label_ref 12)
        (pc))) (nil)
     (nil))
```

RTL for i386: Examples

What does this represent?

```
(jump_insn 15 14 16 4 p1.c:6 (set (pc)
     (if_then_else (lt (reg:CCGC 17 flags)
         (const_int 0 [0x0]))
        (label_ref 12)
        (pc))) (nil)
     (nil))
```

$$pc = r17 < 0 \; ? \; label(12) : pc$$

RTL for i386: Examples

Translation of `if (a > b)`

Dump file: `test.c.131r.expand`

```
(insn 8 7 9 test.c:7 (set (reg:SI 61)
   (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
   (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])) -1 (nil))
(insn 9 8 10 test.c:7 (set (reg:CCGC 17 flags)
   (compare:CCGC (reg:SI 61)
      (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 b+0 S4 A32]))) -1 (nil))
(jump_insn 10 9 0 test.c:7 (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0]))
        (label_ref 0)
        (pc))) -1 (nil))
```

RTL for i386: Examples

Translation of `if (a > b)`

Dump file: `test.c.131r.expand`

```
(insn 8 7 9 test.c:7 (set (reg:SI 61)
   (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
   (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])) -1 (nil))
(insn 9 8 10 test.c:7 (set (reg:CCGC 17 flags)
   (compare:CCGC (reg:SI 61)
      (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 b+0 S4 A32]))) -1 (nil)
(jump_insn 10 9 0 test.c:7 (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0]))
        (label_ref 0)
        (pc))) -1 (nil))
```

RTL for i386: Examples

Translation of `if (a > b)`

Dump file: `test.c.131r.expand`

```
(insn 8 7 9 test.c:7 (set (reg:SI 61)
   (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
   (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])) -1 (nil))
(insn 9 8 10 test.c:7 (set (reg:CCGC 17 flags)
   (compare:CCGC (reg:SI 61)
      (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 b+0 S4 A32]))) -1 (nil))
(jump_insn 10 9 0 test.c:7 (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0]))
        (label_ref 0)
        (pc))) -1 (nil))
```

Part 3

RTL: An Internal View

RTL Objects

- Types of RTL Objects
  - Expressions
  - Integers
  - Wide Integers
  - Strings
  - Vectors
- Internal representation of RTL Expressions
  - Expressions in RTX are represented as trees
  - A pointer to the C data structure for RTL is called `rtx`

RTX Codes

RTL Expressions are classified into RTX codes :

- Expression codes are names defined in rtl.def
- RTX codes are C enumeration constants
- Expression codes and their meanings are machine-independent
- Extract the code of a RTX with the macro `GET_CODE(x)`

RTL Classes

RTL expressions are divided into few classes, like:

- `RTX_UNARY : NEG, NOT, ABS`
- `RTX_BIN_ARITH : MINUS, DIV`
- `RTX_COMM_ARITH : PLUS, MULT`
- `RTX_OBJ : REG, MEM, SYMBOL_REF`
- `RTX_COMPARE : GE, LT`
- `RTX_TERNARY : IF_THEN_ELSE`
- `RTX_INSN : INSN, JUMP_INSN, CALL_INSN`
- `RTX_EXTRA : SET, USE`

RTX Codes

The RTX codes are defined in `rtl.def` using cpp macro call `DEF_RTL_EXPR`, like :

- `DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)`
- `DEF_RTL_EXPR(SET, "set", "ee", RTX_EXTRA)`
- `DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)`
- `DEF_RTL_EXPR(IF_THEN_ELSE, "if_then_else", "eee", RTX_TERNARY)`

The operands of the macro are :

- Internal name of the `rtx` used in C source. It's a tag in enumeration ‘‘`enum rtx_code`"
- name of the `rtx` in the external ASCII format
- Format string of the `rtx`, defined in `rtx_format[]`
- Class of the `rtx`

RTX Formats

```
DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)
```

- `i` : Integer
- `u` : Integer representing a pointer
- `B` : Pointer to basic block
- `e` : Expression

RTL statements

- RTL statements are instances of type `rtx`
- RTL insns contain embedded links
- Types of RTL insns :
  - `INSN` : Normal non-jumping instruction
  - `JUMP_INSN` : Conditional and unconditional jumps
  - `CALL_INSN` : Function calls
  - `CODE_LABEL`: Target label for JUMP_INSN
  - `BARRIER` : End of control Flow
  - `NOTE` : Debugging information

Basic RTL APIs

- XEXP,XINT,XWINT,XSTR
  - Example: XINT(x,2) accesses the 2nd operand of rtx x as an integer
  - Example: XEXP(x,2) accesses the same operand as an expression
- Any operand can be accessed as any type of RTX object
  - So operand accessor to be chosen based on the format string of the containing expression
- Special macros are available for Vector operands
  - XVEC(exp,idx) : Access the vector-pointer which is operand number idx in exp
  - XVECLEN (exp, idx) : Access the length (number of elements) in the vector which is in operand number idx in exp. This value is an int
  - XVECEXP (exp, idx, eltnum) : Access element number "eltnum" in the vector which is in operand number idx in exp. This value is an RTX

RTL Insns

- A function's code is a doubly linked chain of INSN objects
- Insns are `rtxs` with special code
- Each insn contains atleast 3 extra fields :
  - Unique id of the insn , accessed by `INSN_UID(i)`
  - `PREV_INSN(i) accesses the chain pointer to the INSN preceeding i`
  - `NEXT_INSN(i) accesses the chain pointer to the INSN succeeding i`
- The first insn is accessed by using `get_insns()`
- `The last insn is accessed by using get_last_insn()`

Part 4

RTL: An Example Program to Manipulate RTL

Sample Demo Program

Problem statement : Counting the number of SET objects in a basic block by adding a new RTL pass

- Add your new pass after `pass_expand`
- `new_rtl_pass_main` is the main function of the pass
- Iterate through different instructions in the doubly linked list of instructions and for each expression, call `eval_rtx(insn)` for that expression which recurse in the expression tree to find the set statements

```
int new_rtl_pass_main(void){
   basic_block bb;
   rtx last,insn,opd1,opd2;
   int bbno,code,type;
   count = 0;
   for (insn=get_insns(), last=get_last_insn(),
           last=NEXT_INSN(last); insn!=last; insn=NEXT_INSN(insn))
   {    int is_insn;
        is_insn = INSN_P (insn);
        if(flag_dump_new_rtl_pass)
           print_rtl_single(dump_file,insn);
        code = GET_CODE(insn);
        if(code==NOTE){ ... }
        if(is_insn)
        {    rtx subexp = XEXP(insn,5);
             eval_rtx(subexp);
        }
   }
   ...
}
```

```
int new_rtl_pass_main(void){
   basic_block bb;
   rtx last,insn,opd1,opd2;
   int bbno,code,type;
   count = 0;
   for (insn=get_insns(), last=get_last_insn(),
           last=NEXT_INSN(last); insn!=last; insn=NEXT_INSN(insn))
   {    int is_insn;
        is_insn = INSN_P (insn);
        if(flag_dump_new_rtl_pass)
           print_rtl_single(dump_file,insn);
        code = GET_CODE(insn);
        if(code==NOTE){ ... }
        if(is_insn)
        {    rtx subexp = XEXP(insn,5);
             eval_rtx(subexp);
        }
   }
   ...
}
```

```
void eval_rtx(rtx exp)
{ rtx temp;
  int veclen,i,
  int rt_code = GET_CODE(exp);
  switch(rt_code)
  {   case SET:
       if(flag_dump_new_rtl_pass){
           fprintf(dump_file,"\nSet statement %d : \t",count+1);
           print_rtl_single(dump_file,exp);}
       count++; break;
     case PARALLEL:
       veclen = XVECLEN(exp, 0);
       for(i = 0; i < veclen; i++)
       {    temp = XVECEXP(exp, 0, i);
            eval_rtx(temp);
       }
       break;
     default: break;
   }
}
```

```
void eval_rtx(rtx exp)
{ rtx temp;
  int veclen,i,
  int rt_code = GET_CODE(exp);
  switch(rt_code)
  {   case SET:
       if(flag_dump_new_rtl_pass){
           fprintf(dump_file,"\nSet statement %d : \t",count+1);
           print_rtl_single(dump_file,exp);}
       count++; break;
      case PARALLEL:
       veclen = XVECLEN(exp, 0);
       for(i = 0; i < veclen; i++)
       {    temp = XVECEXP(exp, 0, i);
            eval_rtx(temp);
       }
       break;
      default: break;
   }
}
```

```
void eval_rtx(rtx exp)
{ rtx temp;
  int veclen,i,
  int rt_code = GET_CODE(exp);
  switch(rt_code)
  {   case SET:
       if(flag_dump_new_rtl_pass){
           fprintf(dump_file,"\nSet statement %d : \t",count+1);
           print_rtl_single(dump_file,exp);}
       count++; break;
     case PARALLEL:
       veclen = XVECLEN(exp, 0);
       for(i = 0; i < veclen; i++)
       {    temp = XVECEXP(exp, 0, i);
            eval_rtx(temp);
       }
       break;
     default: break;
   }
}
```