Incremental building of machine description in GCC-4.0.2

Sameera Deshpande

September 10, 2006

Contents

1 Introduction

1.1 GCC Structure

GCC is collection of compilers for different high level languages generating code for various architectures. Because of this 'generic' attribute of GCC, GCC cannot contain machine or language dependent code. Instead, it parameterizes language or machine specific attributes, and contains the code which depends upon these parameters. This divides the GCC source broadly into

- Generator files: These files are used to adapt the generic GCC sources for the specified target architecture. These files take description of desired target architecture as an input at the time of building compiler for specified language-target pair, and generates machine specific sources which are required by actual coimpiler, `xgcc`.
- Build source files: These are the files that are required by final compiler `xgcc` being built. The build source files can further be categorized as
  - Existing files: These are the files which have language and machine independent code. These files are available in GCC source tree before build process begins.
  - Generated files: Generated files are machine or language dependent, and depending upon language or target architecture, these files are generated by generator files.

[1]

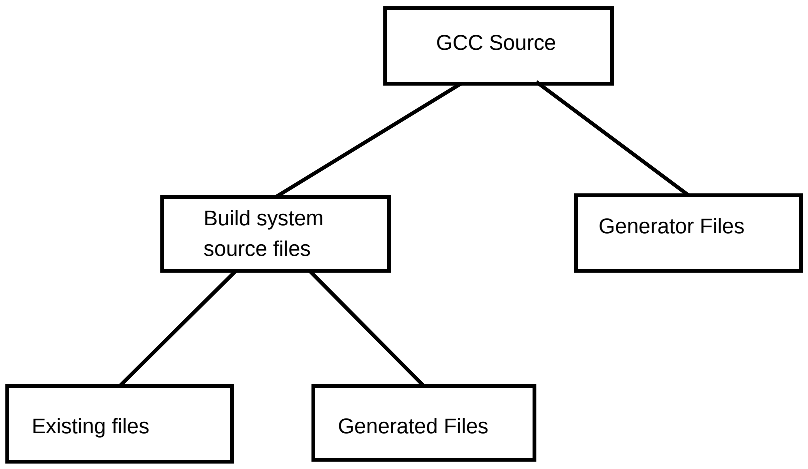


Figure 1: Structure of GCC source files

The figure 1 shows the overall structure in which GCC sources be divided.

The retargetability of compiler forces it to make use of intermediate representation in order to have better code reusability. Without intermediate language, compiler would need m x n translators for each combination of m languages and n target architectures supported by it. Hence for the compilers like GCC, aiming parameterization of target, an intermediate language is needed in order to describe higher level language in terms of generic machine properties.

Introduction of intermediate language gives rise to two main logical transformations on any input program given to the compiler:

[1]Here I wish to give list of files under each specific category... But that list is too big. And I don't know which files to name...

1. Higher level language to intermediate language transformation
2. Intermediate language to assembly language transformation

which must be semantic preserving. Each high level language construct must be translatted to semantically equivalent intermediate language statement(s), which get transformed to piece of code in assembly language. This mapping from high level language and intermediate language or intermediate language to assembly language must be known to the compiler. This is achieved by representing intermediate language in terms of 'pattens'.

In GCC register transfer language (RTL) is used as intermediate language. The high level language program represenated as parse tree is mapped to RTL language instructions through some 'standard pattern names' which are predefined by the compiler. The high level construct gets mapped to semantic preserving pattern associated with standard pattern name corresponding to that construct. This pattern is then used to find out assembly instruction(s) representing that pattern.

The GCC binary is generated by instanciating GCC source for spcific architecture. This binary can then be used to generate target specific assembly code for input source language program. The execution of this binary file can be divided into 3 stages:

- **Front End:** For each high level language there is language specific parser which takes source program as input and generates abstact syntax tree (AST) from it.
- **Middle End:** This stage gets AST as an input, and generates code in register transfer language (RTL) as an output.
- **Back End:** This part gets RTL code as an input and generates assembly code for specified target architecture from it.

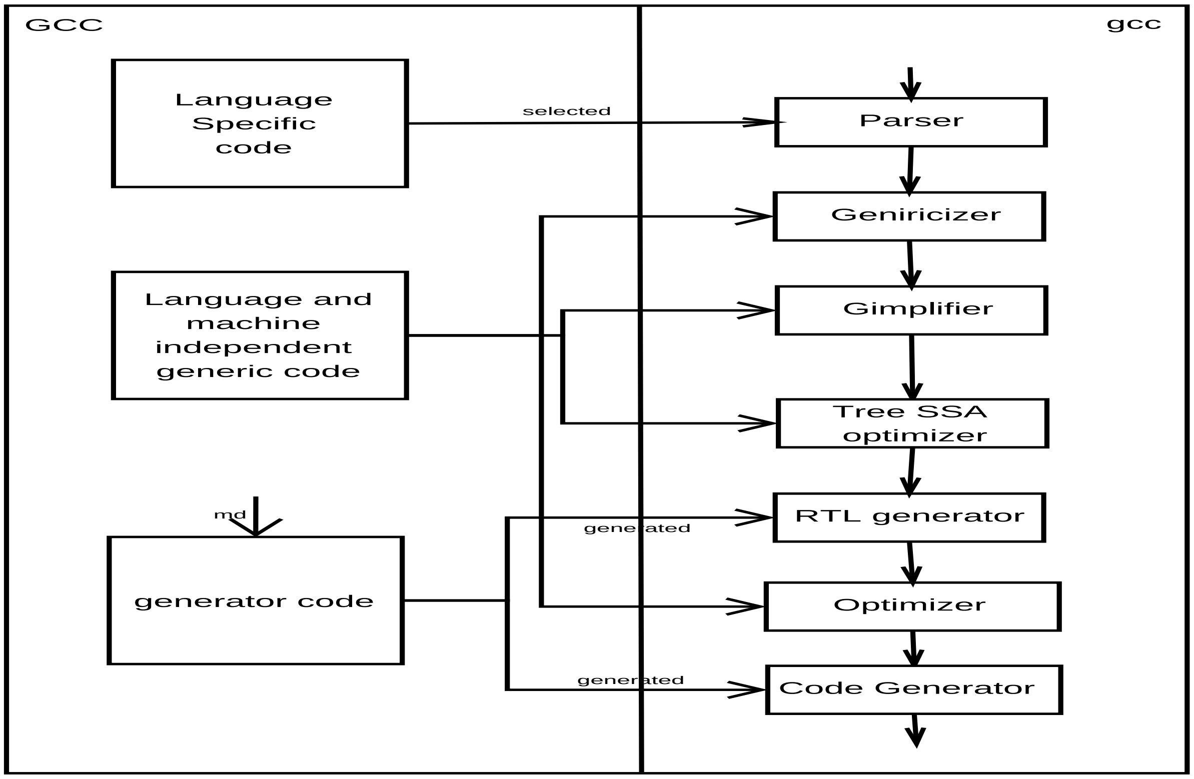


Figure 2: Structure of GCC compiler

The figure 2 gives the transformations input program undergoes when given to *xgcc*, which is the compiler built for specific language-target pair.

While compiling any high level language, the input program is scanned and parsed to generate abstract syntax tree (AST) using `lang_hooks.parse_file`. This AST is input language dependent, and it is reduced to representation understood by language independent parts of compiler : GENERIC trees. This translation is optional, in the sense, front ends for languages like Fortran convert the AST to generic and then lower it to GIMPLE at the time of compiling function, whereas there is no such translation in case of languages like C. For the languages like C, middle end invokes function `gimplify_function_tree` in `gimplify.c` to flatten the generated AST to GIMPLE code. The middle end then performs optimizations on GIMPLE code and finally converts gimplified function into RTL form. The file `cfgexpand.c` has the function `expand_gimple_basic_block` which convert the gimplified function into RTL form basic-block wise.

The back end then performs various optimizations on RTL code, and finally generates assembly code for desired underlying machine for which the compiler was built.

If one wants to include new higher level language in GCC, then it is sufficient to add new front end for that language. This front end should do lexical analysis, parse tree generation, and finally generic tree generation.

On the other hand, if one wants to add back-end for new architecture, then the information about target machine is provided to GCC in the form of machine description. Machine description has three parts:

- .md file : This file contains machine instruction patterns.
- .h file : The header file that define macros used to describe architecture of target machine.
- .c file : The C file defining functions.

The purpose of this project is to get to know about the information that is to be provided to GCC to port new target machine, and try to find reasoning behind inclusion of that information.

1.2 Classification of machine description

The information that is needed by GCC to build compiler for any given processor can be broadly divided into two groups:

- Machine Dependent information: This describes the architecture of processor under consideration. It is detailed description of computational, communication and data storage elements (hardware) of computer system, how those components interact (machine organization), and how they are controlled (Instruction set).
  - Instruction set architecture : This is the conceptual structure and functional behavior of the target. It controls the logic design and the physical implementation on the target. It describes the aspects of a computer architecture visible to a programmer, including the native datatypes, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. It determines computational characteristics of the processor. It mainly deals with design of instruction set. The attributes needed for designing the instruction set along with actual instruction set are included in this class. This class describes the structure of computer seen by programmer.
  - Microarchitecture : It is the set of processor design techniques used to implement the instruction set. It consists of low level details of a machine such as pipelining, memory management systems, specialized instructions processors, buses for communication within and between system, the design and layout of a microprocessor.

- Machine Independent information: This information does not directly depend upon target architecture. This may hold information that is needed by GCC, which depends upon selection of target machine but has nothing to do with target architecture. This information can be further divided as:
  - GCC specific information
  - function calling convention
  - Assembly file output format.
  - Layout of source language data-types.
  - Runtime target specifications.
  - Tool chain specific information.

Classifying the information required by GCC in this manner helps us in organizing the architectural details of target machine in better way.

1.3 What is incremental?

It is not sufficient to organize the machine description in systematic way, because even though machine description is organized successfully, it does not throw light on direct relation between higher level language constructs and architectural details of machine. Hence, here we propose building the compiler for small subset of C language, and then gradually go on increasing the scope of language supported by our port. For this purpose, we have levelised complete C language as follows:

Level 0 : The macros and RTLs that are necessary to build the compiler. This might not compile a single C program, but executable xgcc is generated.

Level 1 : The macros and RTLs that are necessary to perform assignment operation on integers.

Level 2 : The macros and RTLs needed for Arithmetic operations on integer data type.

Level 3 : The macros and RTLs needed for Control structure handling.

Level 4 : The macros and RTLs that are necessary for having various data types supported in machine.

Level 5 :The macros and RTLs needed for function handling. This involves designing activation record, also.

Each level c_i for source C code corresponds to level m_i of machine code generated.

In incremental build process, we are trying to find out the minimal information that is to be given by md writer in order to compile the programs written in some level i of C language, without crashing the compiler.

1.4 Why incremental?

The purpose behind using incremental strategy in building compiler is

- To include a back end for new target machine in GCC, complete knowledge of target architecture is required. From section 1.2, it can be seen, the information required to build complete port for target machine is huge, and managing that at a stretch becomes difficult.Divide and conquer strategy always helps in managing large tasks. So, this is an attempt to systematically divide the task of md writing, so as to concentrate on smaller part of md at a time.

- To know the information that is absolutely necessary to support some higher level language constructs for specific target machine. Because of incremental structure, each construct can be handled and observed carefully against variations in target machine.
- To extract underlying abstract machine assumed by GCC.

2 Role of md file in GCC

Here, we are trying to study role of md file, the target dependent source file, which is taken as input by generator files, and the target dependent source code fragments are generated which get linked to build final xgcc.

The machine description is used at three levels in generated compiler:

- To generate RTL code from gimple.
- To perform RTL to RTL transformations, during optimization passes.
- To emit assembly code corresponding to matched RTL pattern.

In this section, we will concentrate on first use of md file: to generate RTL code from gimple.

How the RTL patterns are generated.

There is list of standard RTL pattern names, which GCC recognizes at the time of RTL generation. GCC offers flexibility to md writer of defining RTL patterns for these standard names. User can omit some patterns' definition, depending upon the target architecture. This makes GCC to determine at build time, which patterns are defined, and how named patterns are generated if definition is found in md file.

The task of generating such information is carried our by generator files. Generator files take md file as an input, and scans it to extract desired information from it. Here is the list of files generated by generators, which holds such information extracted from the md file. The figure 3 gives different data structures created at the time of building compiler from MD file .

The files which are generated from .md file at the time of build and are used while GIMPLE to RTL transformation are as follows:

- `insn-codes.h` : This file is generated from source file `gencodes.c`, which goes through the md file, and generates `insn_code CODE_FOR_`*`pattern-name`* for each named pattern definition found in md file for target.

  ```
  enum insn_code {
    CODE_FOR_nop = 0,
    CODE_FOR_jump = 1,
    CODE_FOR_indirect_jump = 2,
            :
            :
            :
    CODE_FOR_nothing
  };
  ```

- `insn-output.c` : This file is generated from `genoutput.c` file, which goes through md file, and creates data structures `insn_ data` and `insn_operand_data` for all the `define_insn` and `define_expand` patterns. The data structure `insn_data` gives name of the function to be invoked in order to emit insn corresponding to named pattern defined in md file.

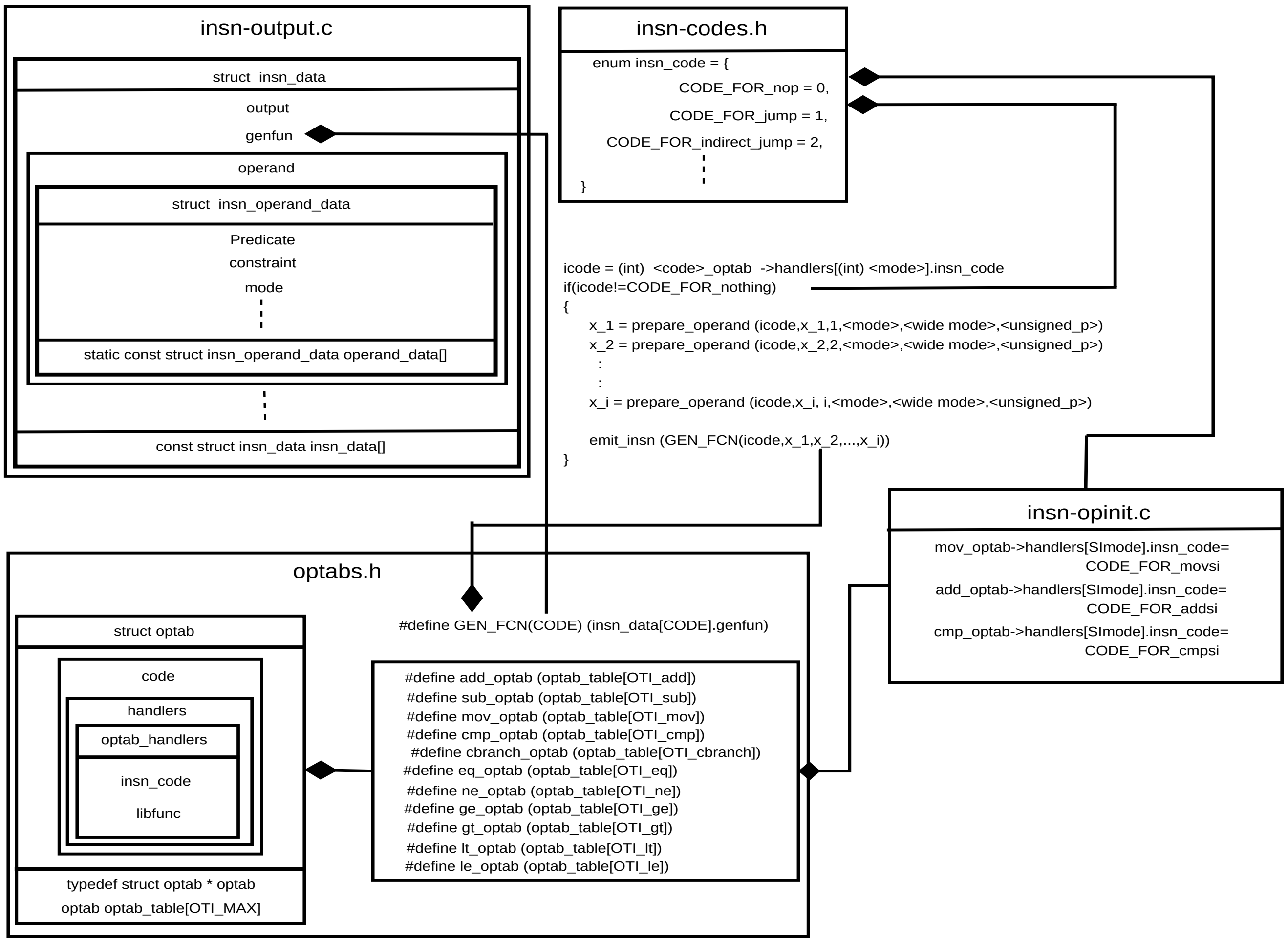

insn-output.c
struct insn_data
output
genfun
operand
struct insn_operand_data
Predicate
constraint
mode
static const struct insn_operand_data operand_data[]
const struct insn_data insn_data[]
insn-codes.h
enum insn_code = {
CODE_FOR_nop = 0,
CODE_FOR_jump = 1,
CODE_FOR_indirect_jump = 2,
}
icode = (int) <code>_optab ->handlers[(int) <mode>].insn_code
if(icode!=CODE_FOR_nothing)
{
x_1 = prepare_operand (icode,x_1,1,<mode>,<wide mode>,<unsigned_p>)
x_2 = prepare_operand (icode,x_2,2,<mode>,<wide mode>,<unsigned_p>)
x_i = prepare_operand (icode,x_i, i,<mode>,<wide mode>,<unsigned_p>)
emit_insn (GEN_FCN(icode,x_1,x_2,...,x_i))
}
insn-opinit.c
mov_optab->handlers[SImode].insn_code= CODE_FOR_movsi
add_optab->handlers[SImode].insn_code= CODE_FOR_addsi
cmp_optab->handlers[SImode].insn_code= CODE_FOR_cmpsi
optabs.h
struct optab
code
handlers
optab_handlers
insn_code
libfunc
typedef struct optab * optab
optab optab_table[OTI_MAX]
#define GEN_FCN(CODE) (insn_data[CODE].genfun)
#define add_optab (optab_table[OTI_add])
#define sub_optab (optab_table[OTI_sub])
#define mov_optab (optab_table[OTI_mov])
#define cmp_optab (optab_table[OTI_cmp])
#define cbranch_optab (optab_table[OTI_cbranch])
#define eq_optab (optab_table[OTI_eq])
#define ne_optab (optab_table[OTI_ne])
#define ge_optab (optab_table[OTI_ge])
#define gt_optab (optab_table[OTI_gt])
#define lt_optab (optab_table[OTI_lt])
#define le_optab (optab_table[OTI_le])

Figure 3: Use of MD file in GIMPLE to RTL conversion.

```
const struct insn_data insn_data[] =
{
  /* /home/sameera/myExperiments/gcc-4.0.2/gcc/config/myToy/myToy.md:4 */
  {
    "nop",
#if HAVE_DESIGNATED_INITIALIZERS
    { .single =
#else
    {
#endif
    "NOP\t\t!In define nop.",
#if HAVE_DESIGNATED_INITIALIZERS
    },
#else
    0, 0 },
#endif
    (insn_gen_fn) gen_nop,
    &operand_data[0],
    0,
      0,
      0,
      1
    },
    :
    :
    :
  }
```

- `insn-emit.c` : This file is generated from `genemit.c`. This file contains definition of all `gen*` functions which generate RTL pattern to be emitted for specific named pattern, as per defined by md writer.
- `insn-extract.c` : This is the file generated by `genextract.c` which extracts operands from the pattern.
- `insn-opinit.c` : This file is generated by `genopinit.c` file. It initializes `optab_table`.

```
add_optab->handlers[SImode].insn_code = CODE_FOR_addsi3;
mov_optab->handlers[SImode].insn_code = CODE_FOR_movsi;
cbranch_optab->handlers[SImode].insn_code = CODE_FOR_cbranchsi4;
bcc_gen_fctn[NE] = gen_bne;
         :
         :
         :
```

How the RTL patterns are emitted.

The figure 3 also gives how the information generated by generator files is used in order to emit RTL pattern matching to gimple code. `optabs.h` is the existing file, which defines array `optab_table` that holds information about standard patterns. This file associates standard names with each entry in `optab_table`. This association is machine independent, and hardcoded. the field `insn_code` plays important role in RTL pattern emission. This is the field which tells programmer if the pattern (s)he is looking for is defined by md writter or not. If the pattern is not defined, then this field is set to `CODE_FOR_nothing` at the time of building the compiler by function `new_optab` in existing file `optabs.c`. If the pattern is defined however, the field is set to have value `CODE_FOR_`*`pattern-name`* by generated file `insn-opinit.c` as disussed in previous section.

The macro definitions in `optabs.h` file are like

```
#define add_optab (optab_table[OTI_add])
#define mov_optab (optab_table[OTI_mov])
#define cbranch_optab (optab_table[OTI_cbranch])
#define eq_optab (optab_table[OTI_eq])
          :
          :
          :
```

which are then used to access appropriate `optab_table` entry whenever needed.

The file `optabs.h` also defines the macro `GEN_FCN` which invokes the function to emit the RTL pattern associated with the named pattern under consideration. Thus, though name of the function emitting RTL pattern is not hard-coded, the macro `GEN_FCN` can be used by the existing sorce files.

At the time of compiling input programs using built system, `tree_expand_cfg` is the function which expands basic blocks in GIMPLE to RTL. This function handles control transferring statements and other statements separately. If the GIMPLE code under consideration is control transfer statement, it eventually invokes the function `emit_cmp_and_jump_insn` otherwise it finally calls driving function `expand_expr_real_1`.

The function `expand_expr_real_1` handles all basic GIMPLE codes. If the GIMPLE expression is not having target, then that expression might be dead, and need not be evaluated at all. But, there might be few operands in that expression, which may cause side-effects. So, in such case, just the operands causing side-effect are translated into RTL. Otherwise, it examines the code of GIMPLE tree, and invokes expanders to emit RTLs appropriate to those codes. These expander functions use the macro `GEN_FCN` to emit desired RTL pattern.

3 Levelised C and its correlation with md

3.1 level0

In level 0, no C language construct is supported. In this level, aim is to find the minimal information that is necessary to build the compiler successfully. The only input program that gets compiled using `xgcc` generated by this port is

```
void main()
{
}
```

3.1.1 Why some information should go to machine description?

To add new port for some target machine in GCC, the information about target architecture must be provided to GCC, through machine description. If the basic features of the target are not given to compiler, it fails to build. The basic features include

- Register set.
- Storage layout.
- Activation record design.
- Addressing Modes.
- Assembly file output format.

along with some miscellaneous parameters.

The `CODE_FOR_*` is insn code for standard name pattern. When `optabs.c` is executed at the time of building of compiler, it refers to `CODE_FOR_indirect_jump` which will not be defined if that RTL is not written in md file. Hence, it flashes error at build time.

The source files `cfgrtl.c, cse.c,expmed.c` which contribute in `cc1`, use the function `gen_jump` to generate insn corresponding to jump instruction which is generated by generator file `genemit.c`. If this pattern is not defined in machine description, the compiler, fails to build.

Hence, the RTL patterns `indirect_jump` and `jump` are defined in level0 md file for ‘minMd’.

The GCC documentation says that `NOP` insn is also necessary, but the compiler gets built successfully without that pattern.

3.1.2 The information that goes in machine description

There are only 2 RTLs necessary for target compiler to 'build' successfully:

1. JUMP : This standard pattern gives a jump inside a function. Operand 0 is the `label_ref` of the label to jump to. This pattern name is mandatory on all machines. This pattern is defined as follows:

   ```
   (define_insn "jump"
           [(set (pc) (label_ref (match_operand 0 "" "")))]
           ""
           {
                   return "JMP\\t%l0\\t\\t!In define jump.";
           }
   )
   ```

   This RTL gives unconditional jump instruction for intermediate abstract machine.

2. INDIRECT_JUMP : This standard pattern name gives instruction to jump to an address which is operand 0. This pattern name is mandatory on all machines. The pattern is defined as follows:

   ```
   (define_insn "indirect_jump"
           [(set (pc) (match_operand:SI 0 "register_operand" ""))]
           ""
           "JMPIN\\t%0\\t\\t!In define indirect jump."
   )
   ```

The macros that define target architecture that must be specified to build the compiler successfully are:

- **Register Usage**
  - Basic Characteristics of registers :
    - FIRST_PSEUDO_REGISTER : This macro gives total number of registers in target machine. 'MinMd' has 16 general purpose registers and 16 floating point registers. Hence, first pseudo register will be numbered as 32 0 to 31 being hardware registers.

      ```
      #define FIRST_PSEUDO_REGISTER \
      32
      ```
    - FIXED_REGISTERS : This macro gives the hardware registers which have fixed functionality hence, which cannot be allocated by register allocator. The registers fixed in 'MinMd' are:
      0-Return value,
      1-Struct return,
      14-Frame pointer,
      15-Stack pointer.
      Hence definition of this macro is

      ```
      #define FIXED_REGISTERS \
      {1,1,0,0,0,0,0,0, \
       0,0,0,0,0,0,1,1, \
       0,0,0,0,0,0,0,0, \
       0,0,0,0,0,0,0,0}
      ```

  * CALL_USED_REGISTERS : Function call may modify contents of some registers. Hence, if those registers are used to store data across function boundaries, the data contained in those registers, might get lost. Hence, GCC must know the registers which get clobbered across function calls. This macro is used to list registers, clobbered because of function call. All fixed registers must also be call clobbered. 'MinMd' does not have any call clobbered register than fixed registers. Hence, this macro is defined as follows:

    ```
    #define CALL_USED_REGISTERS \
    {1,1,0,0,0,0,0,0, \
     0,0,0,0,0,0,1,1, \
     0,0,0,0,0,0,0,0, \
     0,0,0,0,0,0,0,0}
    ```

- How values fit in registers :
  * HARD_REGNO_NREGS : This macro gives number of consecutive hard registers required to hold value of given mode, starting from register REGNO. On most of the processors, the size of registers is same. Hence this is the definition used by most of the processors. But if the size of registers in register set is different, then depending upon REGNO, the number of registers required to hold data of mode MODE will be computed.

    ```
    #define HARD_REGNO_NREGS(REGNO, MODE) \
    ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1)  \
                   / UNITS_PER_WORD)
    ```

  * HARD_REGNO_MODE_OK : This macro returns 1 if it is permissible to hold value of mode MODE in register REGNO. If data is real and size of data is greater than word size, then 2 consecutive floating point registers on even boundary are required to hold data. If the real data is of size less than unit word, then any floating point register will be allowed. Similarly, for integer data with size greater than word size, will need 2 consecutive general purpose registers aligned on even boundaries. And any general purpose register is suffice to hold integer value smaller than word size. The function `hard_regno_mode_ok` handles all these cases.

    ```
    #define HARD_REGNO_MODE_OK(REGN, MODE) \
    hard_regno_mode_ok (REGN, MODE)

    hard_regno_mode_ok (int REGN, enum machine_mode MODE)
    {
     if(GET_MODE_CLASS(MODE) == MODE_FLOAT)
     {
      if(GET_MODE_SIZE(MODE) > UNITS_PER_WORD)
      {
       if(REGN >= 16 && (REGN % 2) == 0)
       /*Floating point data with size = 64 bits.*/
        return IITB_YES;
       return IITB_NO;
      }
      else
      {
       if(REGN >= 16)
        return IITB_YES;
       return IITB_NO;
      }
    ```

```
  }
  else
  {
   if(GET_MODE_SIZE(MODE) >= UNITS_PER_WORD)
   /*Double Integer value.*/
   {
    if(REGN >= 0 && REGN <= 15 && (REGN % 2) == 0)
     return IITB_YES;
    return IITB_NO;
   }
   else
   {
    if(REGN >= 0 && REGN <= 15)
     return IITB_YES;
    return IITB_NO;
   }
  }
 }
```

  * MODES_TIEABLE_P : This macro gives C expression that is nonzero if a value of mode MODE1 is accessible in mode MODE2 without copying. This is is valid in 'MinMd' only if both modes are same or both modes are from same mode class and MODE1 is of smaller size.

```
#define MODES_TIEABLE_P(MODE1, MODE2) \
modes_tieable_p (MODE1,MODE2)

int
modes_tieable_p(enum machine_mode MODE1, enum machine_mode MODE2)
{
 if((MODE1 == MODE2)
  || (GET_MODE_SIZE(MODE1) <= GET_MODE_SIZE(MODE2)
   && GET_MODE_CLASS(MODE1) == GET_MODE_CLASS(MODE2)))
    return IITB_YES;
 return IITB_NO;
}
```

- **Register Classes** There may be some restrictions on use of registers in some instructions. The registers can be classified on the basis of their use in the instructions. The 'register class' divides register set according to their use. Once classified, one can specify register classes that are allowed as operands to particular instruction patterns. In the our processor, the basic classes considered are `NO_REGS` which indicates that no register can be used in the instruction under consideration. `ALL_REGS`, lists all registers in the processor, and indicates that the instruction can use any register. `GENERAL_REGS` lists general purpose registers, indicating that few instructions can use only registers in this class and cannot use special registers for that purpose. Moreover we have `INDEX_REGS`, `BASE_REGS` and `FLOAT_REGS` which gives registers that can be used for specific operation.

  Usage of registers in different addressing modes can be specified using different macros coming under this catagory.

  – `enum reg_class`: An enumerated type that must be defined with each register class name. `NO_REGS` must be first. `ALL_REGS` must be the last register class, followed by one more enumeral value, `LIM_REG_CLASSES`, which is not a register class but rather tells how many classes are there. In our example, we are assuming 6 register classes, dividing the register set into index registers, base registers, floating point registers and general purpose registers.

```
enum reg_class \
{\
        NO_REGS,\
        INDEX_REGS,\
        BASE_REGS,\
        GENERAL_REGS,\
        FLOAT_REGS,\
        ALL_REGS,\
        LIM_REG_CLASSES \
   };
```

– N_REG_CLASSES : This macro gives number of register classes. The hardware registers are managed as various classes of registers. This information can be used for register usage in various addressing modes. Moreover this information can be used to have constraints on registers used while using some assembly instructions. By default, this is LIM_REG_CLASSES.

```
#define N_REG_CLASSES LIM_REG_CLASSES
```

– REG_CLASS_NAMES : This macro gives the register class names in the sequence given in enum reg_class. these names are used in writing debugging dumps.

```
#define REG_CLASS_NAMES \
{\
        "NO_REGS",\
        "INDEX_REGS",\
        "BASE_REGS",\
        "GEN_REGS",\
        "FLOAT_REGS",\
        "ALL_REGS"\
}
```

– REG_CLASS_CONTENTS : This macro gives registers in specific register class. n^th entry in this array gives bitmask for contents of n^th class. We have decided to have Registers from 0 to 7 as index registers, from 8 to 15 as base registers, all index and base registers can be used as general purpose registers,and registers from 16 to 31 as single prcision floating point registers. Hence this macro is defined as

```
#define REG_CLASS_CONTENTS \
{0x00000000,0x000000ff,0x0000ff00,0x0000ffff,0xffff0000,0xffffffff}
```

If the number of registers is larger than size of integer (on host machine), then intgers are replaced by set of integers per register class.

– REGNO_REG_CLASS : This macro gives register class in which certain register lies. If register is member of more than one classes, then name of smaller class is returned.

```
#define REGNO_REG_CLASS(REGNO) \
regno_reg_class(REGNO)

enum reg_class
```

```
regno_reg_class(int REGN)
{
        if(is_index_reg(REGN))
                return INDEX_REGS;
        if(is_base_reg(REGN))
                return BASE_REGS;
        if(is_float_reg(REGN))
                return FLOAT_REGS;
        if(REGN<16)
                return GENERAL_REGS;
        return NO_REGS;
}
```

– BASE_REG_CLASS : This macro gives name of the class to which a valid base register must belong. We have seperate BASE_REGs class for base registers.

```
#define BASE_REG_CLASS \
BASE_REGS
```

– INDEX_REG_CLASS : This macro gives name of the class to which a valid index register must belong. We have seperate INDEX_REGs class for index registers.

```
#define INDEX_REG_CLASS \
INDEX_REGS
```

– REG_CLASS_FROM_LETTER : This macro defines constraint letter for each register class. If the letter is one of the character defiend by this function, then corresponding register class is returned, else, the macro returns NO_REGS. Constraint characters are used at the time of RTL matching and assembly code emission.

```
#define REG_CLASS_FROM_LETTER(ch)\
reg_class_from_letter (ch)

enum reg_class
reg_class_from_letter (char ch)
{
        switch(ch)
        {
                case 'b':return BASE_REGS;
                case 'x':return INDEX_REGS;
                case 'f':return FLOAT_REGS;
        }
        return NO_REGS;
}
```

– REGNO_OK_FOR_BASE_P : This macro gives a C expression which is nonzero if the register number given is suitable for use as a base register in operand addresses. In this example, if predicate `is_base_reg` defined by us is true, then the register can be used as base register, otherwise it cannot be used as base register.

```
#define REGNO_OK_FOR_BASE_P(REGNO)\
regno_ok_for_base_p (REGNO)
```

```
int
regno_ok_for_base_p (int REGN)
{
        if(is_base_reg(REGN)
                || (REGN >= FIRST_PSEUDO_REGISTER
                && is_base_reg(reg_renumber[REGN])))
                return IITB_YES;
        return IITB_NO;
}
```

- REGNO_OK_FOR_INDEX_P :This macro gives a C expression which is nonzero if the register number given is suitable for use as an index register in operand addresses. In this example, if predicate `is_index_reg` defined by us is true, then the register can be used as base register, otherwise it cannot be used as index register.

```
#define REGNO_OK_FOR_INDEX_P(REGNO)\
regno_ok_for_index_p (REGNO)

int
regno_ok_for_index_p (int REGN)
{
        if(is_index_reg(REGN)
                || (REGN >= FIRST_PSEUDO_REGISTER
                       && is_index_reg(reg_renumber[REGN])))
                return IITB_YES;
        return IITB_NO;
}
```

- PREFERRED_RELOAD_CLASS :A C expression that places additional restrictions on the register class to use when it is necessary to copy value 'X' into a register in class 'CLASS'. We don't put any restriction on the register class usage, hence, we assume default definition.

```
#define PREFERRED_RELOAD_CLASS(X, CLASS) \
CLASS
```

- CLASS_MAX_NREGS : A C expression for the maximum number of consecutive registers of class `CLASS` needed to hold a value of mode `MODE`.

```
#define CLASS_MAX_NREGS(CLASS, MODE) \
((GET_MODE_SIZE(MODE)+UNITS_PER_WORD-1)/UNITS_PER_WORD)
```

• **Stack layout and Calling convensions** The macros under this class give information about stack layout and actions to be taken during function calls and returns. We can specify register pointing to stack, a frame (i.e. activation record), direction of growth for stack, position of local variables and function arguments in activation record relative to the frame or stack pointers.

- Basic stack layout :
  - ∗ STACK_GROWS_DOWNWARD : Define this macro if pushing a word onto the stack moves the stack pointer to a smaller address. This macro is checked in #ifdef. So, if this macro is not defined, it is assumed that the stack growth is in upward direction.

    ```
    #define STACK_GROWS_DOWNWARD
    ```

  - ∗ FRAME_GROWS_DOWNWARD : Define this macro if the addresses of local variable slots are at negative offsets from the frame pointer. If this macro is not defined, it is assumed that the local variables are at positive offset from frame pointer.

    ```
    #define FRAME_GROWS_DOWNWARD
    ```

  - ∗ ARGS_GROW_UPWARD : This macro does not have any significance as such. The macro supported by GCC is `ARGS_GROW_DOWNWARD`. But for our stack design, that is not true. Hence, to indecate that, we have defined this new macro.

    ```
    #define ARGS_GROW_UPWARD
    ```

  - ∗ STARTING_FRAME_OFFSET : This macro gives offset from the frame pointer to the first local variable slot to be allocated. In our actvation record design, the local frame is pushed on return address and dynamic chain address. Hence, we add offset of 2 words in frame pointer to address first local variable slot.

    ```
    #define STARTING_FRAME_OFFSET \
    starting_frame_offset ()

    int
    starting_frame_offset ()
    {
            unsigned int num_words = 0;
            int i;
            num_words += 1;
            num_words += 1;
            return (-1 * num_words * UNITS_PER_WORD);
    }
    ```

  - ∗ STACK_POINTER_OFFSET :Offset from the stack pointer register to the first location at which outgoing arguments are placed. If not specified, the default value of zero is used. This is the proper value for most machines.

    ```
    #define STACK_POINTER_OFFSET \
    0
    ```

  - ∗ FIRST_PARM_OFFSET :Offset from the argument pointer register to the first argument's address. On some machines it may depend on the data type of the function.

    ```
    #define FIRST_PARM_OFFSET(FUN)\
    0
    ```

– Registers Used to access the Stack Frame :

  - ∗ STACK_POINTER_REGNUM :This macro gives register number of dedicated stack pointer register in target machine which must also be a fixed register specified through `FIXED_REGISTERS`

    ```
    #define STACK_POINTER_REGNUM \
    15
    ```

  - ∗ FRAME_POINTER_REGNUM : The register number of the frame pointer register which is used to access automatic variables in the stack frame.
    The macro `FRAME_POINTER_REGNUM` must be defined by every processor. But whether to use the register `FRAME_POINTER_REGNUM` to access the local variables of a function or eliminate it depends upon definition of macro `FRAME_POINTER_REQUIRED`.

    ```
    #define FRAME_POINTER_REGNUM \
    14
    ```

    * ARG_POINTER_REGNUM : The register number of the arg pointer register which is used to access the function's argument list. It is same as `FRAME_POINTER_REGNUM` in our processor.

      ```
      #define ARG_POINTER_REGNUM \
      FRAME_POINTER_REGNUM
      ```

    * FRAME_POINTER_REQUIRED :A C expression which is nonzero if a function must have and use a frame pointer. This expression is evaluated in the reload pass. If its value is nonzero the function will have a frame pointer. If dynamic stack allocation is there, then there is no way in which frame pointer can be eliminated. But, otherwise, if this macro holds non-zero value, compiler tries to address each stack element with the help of stack pointer.

      ```
      #define FRAME_POINTER_REQUIRED \
      1
      ```

    * INITIAL_FRAME_POINTER_OFFSET : A C statement to store in the variable DEPTH the difference between the frame pointer and the stack pointer values immediately after the function prologue.

      ```
      #define INITIAL_FRAME_POINTER_OFFSET(DEPTH)\
      DEPTH=initial_frame_pointer_offset (DEPTH)

      int
      initial_frame_pointer_offset (int DEPTH)
      {
              int size,i;
              size = get_frame_size() + 2 * UNITS_PER_WORD;
              for(i=0;i<FIRST_PSEUDO_REGISTER;i++)
              {
                      if(IITB_LIVE_REG_CONDITION(i))
                      {
                              size = size + UNITS_PER_WORD;
                      }
              }
              return (size);
      }
      ```

      According to our stack design, the frame pointer points to base of new activation record, and stack pointer points to top of the stack. Hence, the difference between frame pointer and stack pointer values is sizeof(register save area) + sizeof(local frame) + starting frame offset.

- Passing function arguments :

    * ACCUMULATE_OUTGOING_ARGS : This is a C expression. If nonzero, the maximum amount of space required for outgoing arguments will be computed and placed into the variable `current_function_outgoing_args_size`. In our processor, this value is set to 0, as we don't want to accumulate arguments, instead pass the arguments on the stack, as and when function is invoked.

      ```
      #define ACCUMULATE_OUTGOING_ARGS \
      0
      ```

    * RETURN_POPS_ARGS : This is again C expression that should indicate the number of bytes of its own arguments that a function pops on returning, or 0 if the function pops

no arguments and the caller must therefore pop them all after the function returns. This is true for our case. Hence, we set this macro to have value 0.

```
#define RETURN_POPS_ARGS(FUN, TYPE, SIZE) \
0
```

  * FUNCTION_ARG : This is C expression that controls whether a function argument is passed in a register, and which register. We are passing all the arguments on stack, hence the macro should have value 0.

    ```
    #define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) \
    0
    ```

  * FUNCTION_ARG_REGNO_P : A C expression that is nonzero if regno is the number of a hard register in which function arguments are sometimes passed.

    ```
    #define FUNCTION_ARG_REGNO_P(r) \
    0
    ```

  * CUMULATIVE_ARGS : A C type for declaring a variable that is used to summerise all information about previous arguments in `FUNCTION_ARG`.

    ```
    #define CUMULATIVE_ARGS \
    int
    ```

  * INIT_CUMULATIVE_ARGS : This is the C code fragment which initialises the variable CUM at the beginning of argument list.

    ```
    #define INIT_CUMULATIVE_ARGS(CUM, FNTYPE, LIBNAME, FNDECL, NAMED_ARGS)  \
    {\
    CUM = 0;\
    }
    ```

  * FUNCTION_ARG_ADVANCE : This macro gives a C statement to update the summarizer variable cum to advance past an argument in the argument list.

    ```
    #define FUNCTION_ARG_ADVANCE(cum, mode, type, named) \
    cum++
    ```

– How Scalar Function Values Are Returned :

  * FUNCTION_VALUE : A C expression to create an RTX representing the place where function returns a value of data type `VALTYPE`. As the value can be returned in return-register, this gives us the RTX which sets the value returned by function to fixed register Return-pointer.

    ```
    #define FUNCTION_VALUE(valtype, func)\
    function_value()

    rtx
    function_value ()
    {
            //Return register is register 0 when value is of type SImode.
            return (gen_rtx_REG(SImode,0));
    }
    ```

  * LIBCALL_VALUE : A C expression to create an RTX representing the place where a library function returns a value of mode mode. If the precise function being called is known, func is a tree node (`FUNCTION_DECL`) for it; otherwise, func is a null pointer.

    ```
    #define LIBCALL_VALUE(MODE) \
    function_value()
    ```

    * FUNCTION_VALUE_REGNO_P : A C expression that is nonzero if regno is the number of a hard register in which the values of called function may come back.

      ```
      #define FUNCTION_VALUE_REGNO_P(REGN) \
      ((REGN) == 0)
      ```

- **Addressing modes**
  - CONST_OK_FOR_LETTER_P : This macro gives constraint characters for constant values. If the value lies in specified range for some character, the macro returns True, False otherwise.

    ```
    #define CONST_OK_FOR_LETTER_P(VALUE, CH)\
    const_ok_for_letter_p (VALUE, CH)

    int
    const_ok_for_letter_p(int VALUE,char CH)
    {
            switch(CH)
            {
                    case 'I':if((unsigned)VALUE < 0x256) return IITB_YES;
                                      return IITB_NO;
                    case 'J':if((unsigned)VALUE < 0x1024) return IITB_YES;
                                      return IITB_NO;
                    case 'K':if((unsigned)VALUE < 0x8192) return IITB_YES;
                                      return IITB_NO;
            }
    }
    ```

  - CONST_DOUBLE_OK_FOR_LETTER_P :A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of const_double values (G or H).

    ```
    #define CONST_DOUBLE_OK_FOR_CONSTRAINT_P(op,ch,p) \
    1
    ```

  - CONST_DOUBLE_OK_FOR_CONSTRAINT_P :A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of const_double values (G or H).

  - CONSTANT_ADDRESS_P : A C expression that is 1 if the RTX x is a constant which is a valid address.

    ```
    #define CONSTANT_ADDRESS_P(X) \
    constant_address_p(X)

    int
    constant_address_p (rtx X)
    {
            return (CONSTANT_P(X) &&
                    GET_CODE(X)!=CONST_DOUBLE);
    }
    ```

    In case of 'minMd', if the RTX is constant, and not double, we accept it as valid address.

- MAX_REGS_PER_ADDRESS : A number, the maximum number of registers that can appear in a valid memory address.

  ```
  #define MAX_REGS_PER_ADDRESS \
  2
  ```

  This number depends upon addressing modes supported by the processor. We support all the addressing modes, including register indirect addressing mode, base-index addressing mode, base displacemet addressing mode, hence maximum number of registers required are 2.

- GO_IF_LEGITIMATE_ADDRESS : A C compound statement with a conditional goto label; executed if x (an RTX) is a legitimate memory address on the target machine for a memory operand of mode mode.

  ```
  #ifdef REG_OK_STRICT
  #define GO_IF_LEGITIMATE_ADDRESS(mode,x,label) \
  {\
  if (go_if_legitimate_address1(mode,x))\
          goto label;\
  }
  #else
  #define GO_IF_LEGITIMATE_ADDRESS(mode,x,label) \
  {\
          if (go_if_legitimate_address2(mode,x))\
                          goto label;\
  }
  #endif
  ```

  This macro has 2 varients: strict varient and non-strict varient. The strict variant is used in the reload pass. It must be defined so that any pseudo-register that has not been allocated a hard register is considered a memory reference. If the address is valid base register or if address is addition of base register and index register or base register and constant, then this macro returns True, False otherwise.

- REG_OK_FOR_BASE_P :A C expression that is nonzero if x is valid for use as a base register. This macro also has strict and nonstrict varients.

  ```
  #ifdef REG_OK_STRICT
  #define REG_OK_FOR_BASE_P(x) \
  reg_ok_for_base_p1(x)
  #else
  #define REG_OK_FOR_BASE_P(x)  \
  reg_ok_for_base_p2(x)
  #endif
  ```

- REG_OK_FOR_INDEX_P ::A C expression that is nonzero if x is valid for use as an index register. This macro also has strict and nonstrict varients.

  ```
  #ifdef REG_OK_STRICT
  #define REG_OK_FOR_INDEX_P(x) \
  reg_ok_for_index_p1(x)
  #else
  #define REG_OK_FOR_INDEX_P(x) \
  reg_ok_for_index_p2(x)
  #endif
  ```

- LEGITIMIZE_ADDRESS : A C compound statement that attempts to replace x with a valid memory address for an operand of mode mode. win will be a C statement label elsewhere in the code; the macro definition may use

  ```
          GO_IF_LEGITIMATE_ADDRESS (mode, x, win);
  ```

  to avoid further processing if the address has become legitimate.

  ```
  #define LEGITIMIZE_ADDRESS(x,oldx,mode,win) \
  {\
  rtx Samop1;\
  Samop1=legitimize_address(x,oldx,mode);\
  if(memory_address_p(mode,Samop1)){\
          x=Samop1;\
  goto win;}\
  }
  ```

  Here, if address is represented as addition of 2 operands, then both operands are checked, and if they are not legitimate, then those are forced in some register, to make them legitimate.

- LEGITIMATE_CONSTANT_P : A C expression that is nonzero if x is a legitimate constant for an immediate operand on the target machine.

  ```
  #define LEGITIMATE_CONSTANT_P(x) \
  legitimate_constant_p(x)

  int
  legitimate_constant_p (rtx X)
  {
          return (GET_CODE(X)!= CONST_DOUBLE);
  }
  ```

- MOVE_MAX : This macro gives maximum number of bytes that can be moved

  ```
  #define MOVE_MAX \
  4
  ```

- SLOW_BYTE_ACCESS :Define this macro as a C expression which is nonzero if accessing less than a word of memory is no faster than accessing a word of memory, i.e., if such access require more than one instruction or if there is no difference in cost between byte and (aligned) word loads.

  ```
  #define SLOW_BYTE_ACCESS 0
  ```

- GO_IF_MODE_DEPENDENT_ADDRESS : If the memory address addr has different meanings for different modes, then goto label is executed. In this processor, there is no such condition, hence, unconditionally label is visited.

  ```
  #define GO_IF_MODE_DEPENDENT_ADDRESS(addr,label) \
  goto label;
  ```

• **Storage Layout**

  - BITS_BIG_ENDIAN : This macro is set to 1 if most significant bit has lowest value. As this level doesn't support bit-wise operations, value of this macro is not significant, but this macro must be defined. I have set this macro to have value 1.

- BYTES_BIG_ENDIAN : This macro is set to 1 if most significant byte in a word holds lowest number. This is not true in our case, i.e larger address holds higher order bits of data. Hence, this macro holds value 0 in myToy.
- WORDS_BIG_ENDIAN : This macro is set to have the value 1 if, in a multiword object, the most significant word has the lowest number. This applies to both memory locations and registers. This is true in our case, so we set this macro to 1 in myToy.
- BITS_PER_UNIT : This macro gives number of bits in addressable meory unit. This macro defaults to 8.

  ```
  #define BITS_PER_UNIT \
  8
  ```

- BITS_PER_WORD : This macro gives number of bits in single word. I am assuming the word size to be 32 on my machine. This allows me to define all patterns in md file in SImode.

  ```
  #define BITS_PER_WORD \
  32
  ```

- UNITS_PER_WORD : This is sort of redundant macro. Because its value is always BITS_PER_WORD/BITS_PER_UNIT. But this is mandetory, hence defined to have value 4.

  ```
  #define UNITS_PER_WORD \
  4
  ```

- PARM_BOUNDARY : Normal alignment required while passing parameters for function on stack (in bits).

  ```
  #define PARM_BOUNDARY \
  32
  ```

- STACK_BOUNDARY : This macro gives minimum boundary enforced by hardware on stack pointer. On myToy this boundary is same as PARM_BOUNDARY.

  ```
  #define STACK_BOUNDARY \
  32
  ```

- FUNCTION_BOUNDARY : This macro gives alignment needed by function entry point, in bits. I have considered all alignmets to be 32.

  ```
  #define FUNCTION_BOUNDARY \
  32
  ```

- BIGGEST_ALIGNMENT : This gives the maximum alignment that any data item needs. This is also 32 bits for my hypothtical machine.

  ```
  #define BIGGEST_ALIGNMENT \
  32
  ```

- STRICT_ALIGNMENT : This macro is defined to be the value 1 if instructions will fail to work if given data not on the nominal alignment. Even though this is not very common feature, for simplicity I have assumed strict alignment.

  ```
  #define STRICT_ALIGNMENT \
  1
  ```

- **targetm variable** : This is the structure which contains pointers to the functions and data describing target machine. This variable must be defined in <machine>.c file. The definition of the structure can be found in `target.h` file. The file `target-def.h` defines macro `TARGET_INITIALIZER` which is used to initialize this structure to default function pointers. We can change this mapping by overriding definitions of default macros. There are only 2 macros from this structure, which must be defined as default values for those macros are not available, which makes compiler to crash, if those definitions are not given by md writer. Care must be taken however, while defining macros hooked into targetm; these macros must be defined in <machine>.c file and not in <machine>.h file. Otherwise the desired definitions of these macros get overridden by default definitions.

- TARGET_STRUCT_VALUE_RTX : This macro gives location of structure value address[2]. I have dedicated a register to hold structure value address. so, my definition for this macro is:

```
#undef TARGET_STRUCT_VALUE_RTX
#define TARGET_STRUCT_VALUE_RTX \
target_struct_value_rtx

rtx
target_struct_value_rtx(tree fndecl, int incoming)
{
        //register 1 is used to return pointer to
//large memory locations.
        return gen_rtx_REG(Pmode, 1);
}
```

- TARGET_ASM_GLOBALIZE_LABEL : This macro emits commands that will make label global, i.e available for reference from all files. I have attached "gl" to original label to indicate that label is global.

```
#undef TARGET_ASM_GLOBALIZE_LABEL
#define TARGET_ASM_GLOBALIZE_LABEL \
target_asm_globalize_label

void
target_asm_globalize_label(FILE *stream, const char *name)
{
        fprintf(stream,"gl%s",name);
        return;
}
```

- **Runtime Target Specification**
  - TARGET_SWITCHES : This macro defines names of command options to set and clear bits in target_flags. This macro is of no use, when the compiler is used to generate assembly code. Hence, this macro is set to have insignificant information.

    ```
    #define TARGET_SWITCHES                                              \
     {"", 0, ""}
    ```

[2]The address of block of memory where return value of structure is stored, is called structure value address

– TARGET_VERSION : This macro defines stream to display on stderr describing specific target machine choice.

```
fprintf(stderr,"IIT Bombay, MyToy");
```

- **Assembly File Output Formating**

  – Output of assembler instructions :

    * PRINT_OPERAND : A C compound statement to output to stdio stream `stream` the assembler syntax for an instruction operand `x`. `x` is an RTL expression.`Code` is a value that can be used to specify one of several ways of printing the operand. It is used when identical operands must be printed differently depending on the context. `code` comes from the '%' specification that was used to request printing of the operand in .md file.

```
#define PRINT_OPERAND(STREAM, X, CODE)                                  \
         print_operand(STREAM, X, CODE)
```

    * PRINT_OPERAND_ADDRESS : A C compound statement to output to stdio stream `stream` the assembler syntax for an instruction operand that is a memory reference whose address is `x`. This macro must handle all the addressing modes supported by `GO_IF_LEGITIMATE_ADDRESS`.

```
#define PRINT_OPERAND_ADDRESS(STREAM, X)                                \
         print_operand_address(STREAM, X)
```

    * REGISTER_NAMES : This macro gives the names by which the registers in target machine are known to assembler. This macro is defined as

```
#define REGISTER_NAMES \
{"RET","%r1","%r2","%r3","%r4","%r5","%r6","%r7", \
  "%r8","%r9","%r10","%r11","%r12","%r13","SP","FP", \
  "%f0","%f1","%f2","%f3","%f4","%f5","%f6","%f7", \
  "%f8","%f9","%f10","%f11","%f12","%f13","%f14","%f15" \
 }
```

  – Assembly output and Generation of Labels :

    * ASM_GENERATE_INTERNAL_LABEL : A C statement to store into the string `string` a label whose name is made from the string `prefix` and the number `num`.

```
#define ASM_GENERATE_INTERNAL_LABEL(STRING, PREFIX, NUM)                \
        asm_generate_internal_label(STRING, PREFIX, NUM)

void
asm_generate_internal_label(char *STRING,char *PREFIX,int NUM)
{
        sprintf(STRING,"*%s%d", PREFIX,NUM);
}
```

    * ASM_OUTPUT_INTERNAL_LABEL : A C statement to output to the stdio stream `stream` a label whose name is made from the string `prefix` and the number `num`.

```
#define ASM_OUTPUT_INTERNAL_LABEL(STREAM, PREFIX)                   \
        asm_output_internal_label(STREAM, PREFIX)

void
asm_output_internal_label(FILE *STREAM,char *PREFIX)
{
```

```
                fprintf(STREAM,"%s :",PREFIX);
        }
```

- Overall framework of Assembly :
  - TEXT_SECTION_ASM_OP : This is C expression whose value is a string, including spacing, containing the assembler operation that should precede instructions and read-only data.

    ```
    #define TEXT_SECTION_ASM_OP                                                    \
    ".text"
    ```
  - DATA_SECTION_ASM_OP : This is C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as writable initialized data.

    ```
    #define DATA_SECTION_ASM_OP                                                    \
    ".data"
    ```

- **Miscellaneous Parameters**
- TRULY_NOOP_TRUNCATION :A C expression which is nonzero if on this machine it is safe to ”convert” an integer of `INPREC` bits to one of `OUTPREC` bits (where `OUTPREC` is smaller than `INPREC`) by merely operating on it as if it had only `OUTPREC` bits.

  ```
  #define TRULY_NOOP_TRUNCATION(in,out) \
  1
  ```
- Pmode :An alias for the machine mode for pointers. The pointers are of mode `SImode` in ‘minMd’.

  ```
  #define Pmode SImode
  ```
- FUNCTION_MODE :An alias for the machine mode used for memory references to functions being called, in ‘call’ RTL expressions. It is `QImode` for ‘minMd’.

  ```
  #define FUNCTION_MODE SImode
  ```
- CASE_VECTOR_MODE :An alias for a machine mode name. This is the machine mode that elements of a jump-table should have.

  ```
  #define CASE_VECTOR_MODE SImode
  ```
- DEFAULT_SIGNED_CHAR : This macro is set to one if type char is considered as signed by default. This is not the case, in our processor, hence, set to value 0.

  ```
  #define DEFAULT_SIGNED_CHAR 0
  ```
- TARGET_CPU_CPP_BUILTINS :This function-like macro expands to a block of code that defines built-in preprocessor macros and assertions for the target cpu, using the functions **`builtin_define`**, **`builtin_define_std`** and **`builtin_assert`**. I haven’t understood use of this macro, so I have defined this macro so as to have the functions stated in document.

  ```
  #define TARGET_CPU_CPP_BUILTINS()\
  do                                              \
    {                                             \
  ```

```
      builtin_define_std ("myToy");            \
      builtin_assert ("cpu=myToy");            \
  }                                            \
  while (0)
```

- HAS_INIT_SECTION : If this macro is defined, then `main` does not call `__main` function. As this machine description will support level 0 language features, it will not have call instruction, which will make compiler to fail even for simple program like

  ```
  void main()
  {
  }
  ```

  which will not be correct. Hence, This macro is defined even though it is not one of the *necessary* macros in minimal set of macros.

- **Generating Code For Profiling**

  – FUNCTION_PROFILER : this is C statement or compound statement to output to file some assembler code to call the profiling subroutine mcount.

  ```
  #define FUNCTION_PROFILER(file,lab) \
  function_profiler(file,lab)

  void
  function_profiler
          (asm_file, labelno)
       FILE *asm_file;
       int labelno;
  {
    fprintf (asm_file, "\t");
    fprintf (asm_file, "call\t _mcount");
    fprintf (asm_file, "\n");
  }
  ```

- **Trampolines For Nested Functions**

  – INITIALIZE_TRAMPOLINE : This macro is used to initialize various parts of trampoline. As I am not supporting nested functions, may be trampolines will not prove necessary to me. But definition of this macro is neccessary. Hence I have defined this macro as

  ```
  #define INITIALIZE_TRAMPOLINE \
  initialize_trampoline(a,b,c)
  void
  initialize_trampoline()
  {
          printf("This is trampoline body.");
          return;
  }
  ```

– TRAMPOLINE_SIZE : This gives a C expression for the size in bytes of the trampoline, as an integer.

```
#define TRAMPOLINE_SIZE 32
```

3.2 level1

Level 1 handles the assignment operations on integer data. So, the patterns are added to support assignment operation.

3.2.1 Mapping from `modify_expr` to *movm*

As discussed in section 1.1, each high level construct should be mapped to set of semantically equivalent intermediate language statements and this association must be known to the compiler at build time. The standard pattern names are listed in existing source file `optabs.h` along with maximum number of operands the intermediate instruction can have. The mapping of tree node and the RTL pattern name is given by a function in `expr.c` depending upon the operands of the tree-node. Thus, when a tree-node is found in the tree dump of high level language program, the pattern associated with mapped name is emitted to generate intermediate code.

the assignment operation in C is translated to `modify_expr` when represented as a parse tree. GCC associates standard name `mov`*m* with `MODIFY_EXPR`. This association is hardcoded in the driving function `expand_expr_real_1`. It first gets lhs and rhs of assignment statement, and then invokes the function `expand_assignment` as shown in the snippet given below by passing lhs and rhs as parameters.

```
case MODIFY_EXPR:
        tree lhs = TREE_OPERAND (exp, 0);
        tree rhs = TREE_OPERAND (exp, 1);
        gcc_assert (ignore);
             ://The code here is for future developements,
             ://and in current gcc version, it is not visited.
             ://Hence, it is removed here for simplicity in
             ://explaining MODIFY_EXPR to movm conversion.
        expand_assignment (lhs, rhs);
        return const0_rtx;
```

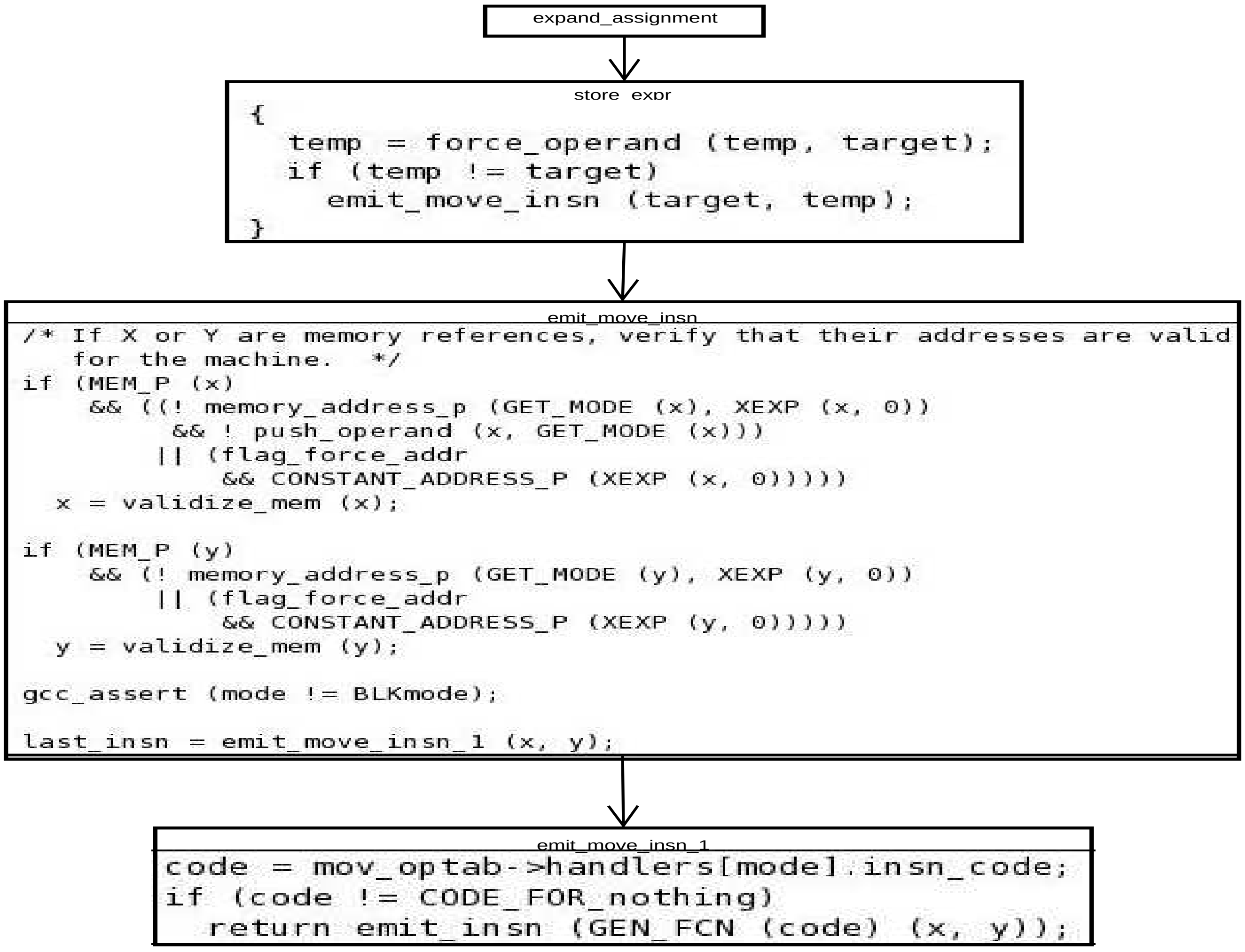
The figure **??**ig1 gives the hierarchy of functions called at the time of converting `MODIFY_EXPR` into *movm* RTL pattern. The function `expand_assignment` handles assignment of structure components, array elements, call to functions, and other assignments seperately. But the function it invokes to actually perform move operation is the function `store_expr`.

The function `store_expr` gets 3 parameters: the source expression, the target RTL in which the expression is to be stored, and a flag. First the source expression is converted into RTL using the function `expand_expr_real`. It then emits the appropriate 'move' instruction to move sorce to target.

The function `emit_move_insn` checks whether the operands are memory operands or not, and if they are, then checks whether the memory addresses are legitimate or not. If the addresses are not legitimate, the memory is legitimized, and then final `emit_move_insn_1` function is invoked to emit the instruction.

3.2.2 What goes in Level 1?

At this level, the pattern is added to support assignment operation. The standard name to define pattern for assignment operation is *movm*. This instruction pattern moves data from operand 1 to



```
{
  temp = force_operand (temp, target);
  if (temp != target)
    emit_move_insn (target, temp);
}
```

```
/* If X or Y are memory references, verify that their addresses are valid
   for the machine.  */
if (MEM_P (x)
    && ((! memory_address_p (GET_MODE (x), XEXP (x, 0))
         && ! push_operand (x, GET_MODE (x)))
        || (flag_force_addr
            && CONSTANT_ADDRESS_P (XEXP (x, 0)))))
  x = validize_mem (x);

if (MEM_P (y)
    && (! memory_address_p (GET_MODE (y), XEXP (y, 0))
        || (flag_force_addr
            && CONSTANT_ADDRESS_P (XEXP (y, 0)))))
  y = validize_mem (y);

gcc_assert (mode != BLKmode);

last_insn = emit_move_insn_1 (x, y);
```

```
code = mov_optab->handlers[mode].insn_code;
if (code != CODE_FOR_nothing)
  return emit_insn (GEN_FCN (code) (x, y));
```

Figure 4: Process of converting from GIMPLE to RTL using *movm* pattern

operand 0. This instruction pattern is important in the sense, it is also used in reload pass to generate insn which copies stack slot into register (load operation). Hence, no extra temporary register should be used. We are also interested in exploring the way in which define_expand and define_insn can be used. It is observed that when GIMPLE to RTL transformation is to be performed at compile time, named pattern for corresponding GIMPLE code is first searched, and if the pattern is available, the operands are modified so as to satisfy predicates, and then actual RTL is emitted. The ways in which define_insn and define_expand are used in GIMPLE to RTL conversion differ a lot. In case of define_insn, the insn given in pattern cannot be modified, hence number of operands in the define_insn for standard pattern name must be same as number of operands that are actually supported by that pattern. This is not the case with define_expand. One can extend number of operands as much as needed, but in that case (s)he must take care of actually assigning desired value to these operands in preparation statements of define_expand. Secondly, in define_expand, one can modify the pattern of generated insn, or one even use define_expand as define_insn. Lets see this with the help of pattern definitions.

- *movsi* using `define_expand` :
  Consider the *movsi* pattern written using `define_expand`.

```
(define_expand "movsi"
 [(set (match_operand:SI 0 "nonimmediate_operand" "")
       (match_operand:SI 1 "general_operand" "")
  )
 ]
 ""
 "{
    if(!reload_in_progress)
    /*This is to see, if we are not in reload pass.*/
    {
      if(GET_CODE(operands[0]) == MEM && GET_CODE(operands[1]) != REG)
      {
        operands[1] = force_reg(SImode,operands[1]);
        emit_insn(gen_rtx_SET(SImode,operands[0],operands[1]));
        DONE;
      }
    }
 }"
)
```

  The RTL template sets non-immediate operand 0 to general operand 1. The condition section is kept null, so no explicit check will be there to match the pattern. The preparation section is C code. It checks if the current pass is reload pass or not. If it is reload pass, then `force_reg` function cannot be used, as `force_reg` function uses temporary intermediate register. If reload pass is not in progress, then one can safely use `force_reg`. Here, if the destination (operand 0) is memory operand, and source (operand 1) is not in register, then the source operand is first moved into temporary register, and then that register is moved into destination by emitting the set insn explicitly. The call to macro `DONE` is used to indicate end of pattern generation for that insn.

  So, when move RTL is to be generated, if destination is in memory and source is non-register entity, 2 insns get emitted, one forcing source into temporary register, and second emitted by `emit_insn`; otherwise, single pattern matching to RTL template will be emitted.

  When the compiler is built, in this example, the code that is generated for this insn is :

```
rtx
gen_movsi (rtx operand0,
        rtx operand1)
{
  rtx _val = 0;
  start_sequence ();
  {
    rtx operands[2];
    operands[0] = operand0;
    operands[1] = operand1;
+----------------------------------------------------------------------+
|{                                                                     |
|  if(!reload_in_progress)                                             |
|    /*This is to see, if we are not in reload pass.*/                 |
|  {                                                                   |
|    if(GET_CODE(operands[0]) == MEM && GET_CODE(operands[1]) != REG) |
|    {                                                                 |
|            operands[1] = force_reg(SImode,operands[1]);              |
|            emit_insn(gen_rtx_SET(SImode,operands[0],operands[1]));  |
|            DONE;                                                     |
|    }                                                                 |
|  }                                                                   |
|}                                                                     |
+----------------------------------------------------------------------+
    operand0 = operands[0];
    operand1 = operands[1];
  }
  emit_insn (gen_rtx_SET (VOIDmode,
        operand0,
        operand1));
  _val = get_insns ();
  end_sequence ();
  return _val;
}
```

The marked region indicates the code written by me while writing md file. When there is need to move data from one operand to another (explicitly in case of tree node MODIFY_EXPR, or implicitly in case of force_reg), the function `gen_movsi` is invoked. There are few assumptions regarding MOVE insn. As MOVE is considered as one of the basic insns, which must be present on the machine, without which data movement is not possible, the predicate checking is **_NOT_** done for *movm* pattern.

- *movsi* using `define_insn` :
  The named pattern written using define_insn is used at the time of RTL generation as well as assembly code emission. There is no provision to modify the RTL insn pattern matched as in define_expand. Hence, if the standard named pattern is defined using define_insn, then the pattern getting generated is same as the pattern getting matched at the time of assembly emission. Lets have a look at this example:

```
(define_insn "movsi"
        [(set (match_operand:SI 0 "nonimmediate_operand" "")
              (plus:SI (match_operand:SI 1 "general_operand")
              (const_int 0)))
        ]
        ""
        "MOV\\t %0,%1"

)
```

Here, *movsi* instruction is represented as addition of source and constant value 0. In this level, RTL for addition is not added. Still the code works for same set of C source programs as previous one because we have just modified the pattern to be generated and matched. As far as number of operands in definition of *movsi* in md file and actual number of operands defined for standard pattern *movm* are same, this definition is also going to work fine. This is mainly because, when the `gen_movsi` function is generated, the pattern is emitted as it was there in md file. Whenever the compiler tries to emit assembly output, the pattern matched is still the same. Hence the compiler never fails. But we still cannot compile source C program having '+' operation, because, still we have not defined *addsi* insn, which is needed by compiler in order to generate RTL matching to the node PLUS_EXPR in GIMPLE. Thus even though RTLs generated by

```
(define_insn "movsi"
        [(set (match_operand:SI 0 "nonimmediate_operand" "")
              (plus:SI (match_operand:SI 1 "general_operand")
              (const_int 0)))
        ]
        ""
        "MOV\\t %0,%1"

)
```

and

```
(define_insn "addsi"
        [(set (match_operand:SI 0 "nonimmediate_operand" "")
              (plus:SI (match_operand:SI 1 "general_operand" "" )
              (const_int 0)))
        ]
        ""
        "ADD -> %0 = %1 + %2"

)
```

are just the same, the standard pattern names referred in order to generate these RTLs are quite different.

3.3 level2

Level 2 supports arithmetic operations on integer data. Hence, the patterns are added to support integer addition, subtraction, multiplication and division. It should be noted that defining patterns for multiplication and division are not necessary, as there are library functions defined to take care of those operations. But in level2, we haven't supported function calls, hence calls to library functions cannot get emitted. Hence, in this level we have decided to add the patterns for these operations also.

3.3.1 What goes in Level 2?

We define the standard insn *addm3* which adds operand 2 and operand 1, storing the result in operand 0. All operands must have mode *m*. This can be used even on two-address machines, by means of constraints requiring operands 1 and 0 to be the same location.

```
(define_insn "addsi3"
        [(set (match_operand:SI 0 "register_operand" "")
              (plus:SI (match_operand:SI 1 "register_operand" "")
                       (match_operand:SI 2 "register_operand" "")
              )
         )
        ]
        ""
        "ADD %0 = %1 + %2"
)
```

Here, we allow addition of register operands only. This forces the compiler, to first invoke `prepare_operand` to satisfy predicates, and then emit the RTL which satisfies predicates for all of its operands. *subm3, mulm3, divm3* are similar to *addm3* insn. These patterns are defined as:

```
(define_insn "subsi3"
        [(set (match_operand:SI 0 "register_operand" "")
              (minus:SI (match_operand:SI 1 "register_operand" "")
                       (match_operand:SI 2 "register_operand" "")
              )
         )
        ]
        ""
        "SUB %0 = %1 - %2"
)

(define_insn "mulsi3"
        [(set (match_operand:SI 0 "register_operand" "")
              (mult:SI (match_operand:SI 1 "register_operand" "")
                       (match_operand:SI 2 "register_operand" "")
              )
         )
        ]
        ""
        "MUL %0 = %1 * %2"
)

(define_insn "divsi3"
        [(set (match_operand:SI 0 "register_operand" "")
              (div:SI (match_operand:SI 1 "register_operand" "")
                       (match_operand:SI 2 "register_operand" "")
```

```
                )
          )
         ]
         ""
         "DIV %0 = %1 * %2"
)
```

3.4 level3

In level3, we have reshuffled the original sequence of levels intentionally. This is because, we can see functions as LHS of any assignment statement. Hence, to successfully complete assignment statements, we decided to move function handling at level 2. Secondly, we wanted to have 'MINIMAL' set of RTLs in 'MinMd' machine description. By this we mean, the absolutely necessary RTLs at level i are added in the machine description. Remaining operations supported in c_i are handled by library functions. When we tried to eliminate MUL, DIV operations, the program crashed because we was not having these insns in my machine description and compiler was not able to generate library function call because CALL insn was not supported either. Hence, function handling was done after arithmatic operations.

We define standard pattern *call* which is subroutine call instruction returning no value. Operand 0 is the name of the function to call; operand 1 is the number of bytes of arguments pushed as a const_int; operand 2 is the number of registers used as operands. In most of the processors operand 2 is not stored in RTL pattern.

Our definition for this pattern is as follows:

```
(define_insn "call"
        [(call (match_operand:SI 0 "symbolic_operand" "")
               (match_operand:SI 1 "immediate_operand" ""))
        ]
        ""
        "CALL %0 with %c1 number of parameters"
)
```

As we are not allowing parameters to be passed in register, operand 2 is not emitted.

The *call_value* is subroutine call instruction returning a value. Operand 0 is the hard register in which the value is returned. Operand 1 is the name of the function to call; operand 2 is the number of bytes of arguments pushed as a const_int, operand 3 is the number of registers used as operands. The definition of this pattern is:

```
(define_insn "call_value"
        [(set (match_operand:SI 0 "register_operand" "")
              (call (match_operand:SI 1 "symbolic_operand" "")
                    (match_operand:SI 2 "immediate_operand" "")))
        ]
        ""
        "%0=CALL %1 with %c2 number of parameters"
)
```

The pattern for return has predefined structure. It sets the program counter of intermediate abstract machine to return pointer. This pattern has the form

```
(define_insn "return"
        [(set (pc) (return))]
        ""
        "RETURN\\t\\t!return from call."
)
```

One thing is to be noted in case of call and return patterns. Call and return insns are not handled same as other instructions. There is no `CODE_FOR_*` macro defined for these insn patterns. When a tree node has code `CALL_EXPR`, the function `expand_call` is invoked. This function takes care of parameter passing issues and then calls function `emit_call_1`. Following given is the part of function `emit_call_1` which emits call insn depending upon, the function returns some value or not.

```
if (HAVE_call && HAVE_call_value)
    {
      if (valreg)
        emit_call_insn (GEN_CALL_VALUE (valreg,
                                        gen_rtx_MEM (FUNCTION_MODE, funexp),
                                        rounded_stack_size_rtx, next_arg_reg,
                                        NULL_RTX));
      else
        emit_call_insn (GEN_CALL (gen_rtx_MEM (FUNCTION_MODE, funexp),
                                  rounded_stack_size_rtx, next_arg_reg,
                                  struct_value_size_rtx));
    }
```

where GEN_CALL and GEN_CALL_VALUE are set to values

```
#define GEN_CALL(A, B, C, D) gen_call ((A), (B))
#define GEN_CALL_VALUE(A, B, C, D, E) gen_call_value ((A), (B), (C))
```

in file insn-flags.h, generated from genflags.c file in source code.

3.5 level4

In level4, we need to add macro `NOTICE_UPDATE_CC` in order to support conditional code handling. This macro is a C compound statement to set the components of cc_status appropriately for an insn insn whose body is exp. It is this macro's responsibility to recognize insns that set the condition code as a byproduct of other activity as well as those that explicitly set (cc0).

```
#define NOTICE_UPDATE_CC(exp,insn) \
        notice_update_cc(exp,insn)
```

The compare and jump operation can be handled in two different ways in md file :

- *cbranchm4*
- *cmpm and bcond*

Each patterns are discussed in detail in next subsections.

3.5.1 *cbranchsi4*

This standard pattern defines conditional branch instruction combined with a compare instruction. Operand 0 is a comparison operator. Operand 1 and operand 2 are the first and second operands of the comparison, respectively. Operand 3 is a label_ref that refers to the label to jump to.

```
(define_insn "cbranchsi4"
        [ (set (pc) ( if_then_else
                        (match_operator:SI 0 "comparison_operator"
                        [ (match_operand:SI 1 "register_operand" "")
                          (match_operand:SI 2 "register_operand" "") ])
                         (label_ref (match_operand 3 "" ""))
                         (pc)))]
        ""
        {
                char str[15],str1[40];
                sprintf(str,"%s",rtx_name[GET_CODE(operands[0])]);
                sprintf(str1,"DCBRANCH %s %%1 %%2 %%l3",str);
                switch(GET_CODE(operands[0]))
                {
                        case EQ: return "CBRANCH EQ %1 %2 %l3";
                        case NE: return "CBRANCH NE %1 %2 %l3";
                /*      case LT: return "CBRANCH LT %1 %2 %l3";
                        case LE: return "CBRANCH LE %1 %2 %l3";
                        case GT: return "CBRANCH GT %1 %2 %l3";
                        case GE: return "CBRANCH GE %1 %2 %l3";*/
                        default: return str1;
                }
        }
        )
```

3.5.2 Combination of *cmpsi* and *bcond*

cmpm is standard pattern name to compare operand 0 and operand 1, and set the condition codes. The RTL pattern should look like this:

```
          (set (cc0) (compare (match_operand:m 0 ...)
                              (match_operand:m 1 ...)))
```

bcond is conditional branch instruction. Operand 0 is a label_ref that refers to the label to jump to. Jump if the condition codes meet condition *cond*.

To have the effect of *cbranchm4* instruction, we have to use power of `define_expand` construct, to modify the structure of RTL to be emitted. Lets see the following code.

```
(define_expand "cmpsi"
        [(set (cc0)
              (compare:SI (match_operand:SI 0 "register_operand" "")
                          (match_operand:SI 1 "register_operand" "")))]
        ""
        {
                compare_op0 = operands[0];
                compare_op1 = operands[1];
                DONE;
        }
)
```

Here, when `gen_cmpsi` is invoked, the contents of `operands[0]` and `operands[1]` are stored in global md writer defined variables `compare_op0` and `compare_op1` respectively. Because of the use of macro

DONE, cmpsi pattern is *not* generated. Now, we can use the stored information about operands when we want to generate conditional branch statements. Lets see the following example to generate *branch if equal* insn:

```
(define_expand "beq"
        [(set (pc)
              (if_then_else (eq (cc0)(const_int 1))
                            (label_ref (match_operand 0 "" ""))
                            (pc)))
        ]
        ""
        "{
           emit_jump_insn(gen_rtx_SET(VOIDmode,
                             pc_rtx,
                             gen_rtx_IF_THEN_ELSE(VOIDmode,
                                gen_rtx_EQ(VOIDmode,
                                   compare_op0,
                                   compare_op1),
                                gen_rtx_LABEL_REF(VOIDmode,
                                   operands[0]),
                                pc_rtx)));
                DONE;
        }"
)

(define_insn "*beq"
        [(set (pc)(if_then_else (eq (match_operand:SI 1 "" "")
                                    (match_operand:SI 2 "" "")
                                )
                                (label_ref (match_operand 0 "" ""))
                                (pc)
                  )
        )]
        ""
        "BEQ (%1 == %2) %l0"
)
```

GCC internal manual says that, *bcond* pattern takes single argument, the code label where jump is to be emitted. But, in our case, desired pattern must take three arguments: the operands to be compared, and label to be jumped if comparison evaluates to be True. We can achieve this using 2 different methods.

Above given is one method to define any pattern. In this, we match the pattern with any arbitrary pattern, and then **unconditionally** correct it, by emitting the desired pattern in code area of define_expand, where desired pattern contains reference to previously saved `compare_op0` and `compare_op1`.

There is another method to generate desired conditional branch statement.

```
(define_expand "blt"
        [(set (pc)(if_then_else (lt (match_dup 1)
                                    (match_dup 2))
                                (label_ref (match_operand 0 "" ""))
                                (pc)))]
        ""
        "{
                operands[1] = compare_op0;
                operands[2] = compare_op1;
        }"
)


(define_insn "*blt"
        [(set (pc)(if_then_else (lt (match_operand:SI 1 "" "")
                                    (match_operand:SI 2 "" "")
                                )
                                (label_ref (match_operand 0 "" ""))
                                (pc)
                  )
        )]
        ""
        "BLT (%1 < %2) %l0"
)
```

Here, we make use of the fact that, in `define_expand`, we can extend number of operands as much as needed. But, one has to be very careful, because if the extra operands are not assigned any value, the compiler may crash.

So, in above given definition of *branch if less than*, we write the pattern as

```
[(set (pc)(if_then_else (lt (match_dup 1)
                                    (match_dup 2))
                                (label_ref (match_operand 0 "" ""))
                                (pc)))]
```

and then, set `operands[0]` and `operands[1]` to `compare_op0` and `compare_op1` respectively. As, here we are not calling macro `DONE`, the function `gen_blt` emits the default pattern by replacing `match_dup`'s with newly assigned patterns.

3.5.3 How control structures are handled?

Control structures are handled quite differently than other language constructs. Actual language constructs are represented as given in following definitions while converting them from C to Parse tree.

```
/* Used to represent a ‘for’ statement. The operands are
   FOR_INIT_STMT, FOR_COND, FOR_EXPR, and FOR_BODY, respectively.  */
DEFTREECODE (FOR_STMT, "for_stmt", tcc_expression, 4)

/* Used to represent a 'while' statement. The operands are WHILE_COND
   and WHILE_BODY, respectively.  */
DEFTREECODE (WHILE_STMT, "while_stmt", tcc_expression, 2)

/* Used to represent a 'do' statement. The operands are DO_BODY and
   DO_COND, respectively.  */
DEFTREECODE (DO_STMT, "do_stmt", tcc_expression, 2)
```

```
/* Used to represent a 'break' statement.  */
DEFTREECODE (BREAK_STMT, "break_stmt", tcc_expression, 0)

/* Used to represent a 'continue' statement.  */
DEFTREECODE (CONTINUE_STMT, "continue_stmt", tcc_expression, 0)

/* Used to represent a 'switch' statement. The operands are
   SWITCH_STMT_COND, SWITCH_STMT_BODY and SWITCH_STMT_TYPE, respectively.  */
DEFTREECODE (SWITCH_STMT, "switch_stmt", tcc_expression, 3)
```

These intermediate tree nodes are then flattened while gimplification. This introduces `goto_expr` and `cond_expr` in the gimplified code, which is used to generate RTL. The gimple to RTL conversion is carried out basic-blockwise and control structures alter the flow of program. Hence the `cond_expr` generated because of such language constructs needs to handle specially.

The function `expand_gimple_cond_expr`, subroutine of `expand_gimple_basic_block` handles these `cond_exprs`. It first extracts information about true and false destinations of basic block, and expressions associated with else and then part of the conditional code. It then invokes the function `jumpif` or `jumpifnot` depending upon fall-through conditions. If there is no fall-through, `jumpif` function is called.

The functions `jumpif, jumpifnot` are defined as follows:

```
void
jumpifnot (tree exp, rtx label)
{
  do_jump (exp, label, NULL_RTX);
}

void
jumpif (tree exp, rtx label)
{
  do_jump (exp, NULL_RTX, label);
}
```

which call function `do_jump` with appropriate true and false labels.

The function `do_jump` is the function which examines the conditional code, and if conditional code is other than basic conditional operators like EQ, LT, GT etc; then it first expands the conditional code, and then invokes the function `do_compare_rtx_and_jump`. Otherwise it invokes function `do_compare_and_jump` with appropriate conditional code.

3.6 Level 5

4 Problems in writting incremental machine description

Even though incremental md writing has many advantages over writing machine description at a stretch, as we have not fixed target architecture, we are still facing a lot of problems while writing incremental md file. Few issues are resolved, but there are few we are still working on.

- Size of long int : When we were trying to handle arrays at some level of C language, we got **internal compiler error: in `instantiate_virtual_regs_lossage`, at `emit_move_insn`:3092** when input program had variable to index into array. The program

```
int main()
{
   int arr[5];
   arr[3]=15;
   return arr[3];
}
```

would generate code successfully, but program

```
int main()
{
   int arr[5],i;
   arr[i]=15;
   return arr[i];
}
```

used to crash giving internal compiler error. It was then found that because we had set size of long int type as 64 bits, and we had not defined patterns for *DImode*, the compiler was crashing. There were 3 solutions to solve this problem, all of which work perfectly fine.

1. Define patterns for *DImode*.
2. Define `LONG_TYPE_SIZE` as 32 bits instead of 64 bits.
3. Define macro SIZE_TYPE which gives name of data type to use for size values when tree is built to be unsigned int.

- Control structure handling : The control structure handling can be done either using old method which combines *cmpsi* and *bcond* patterns to generate conditional code or using *cbranchsi4* pattern. But when we try to fuse these two constructs, the compiler behaves in unusual way. If we define a pattern for *cmpsi* and *cbranchsi4*, the compiler does not recognize the pattern *cmpsi*, and hence calls library routine `__cmpsi2`. If the pattern definition for *cmpsi* is removed, the call to this library routine is not emitted at all. We are currently going through the source code, but we haven't found any thing unusual in it because of this compiler is behaving in unexpected manner.

- Floating point operations : This problem is somewhat critical. If we do not define any floating point operation, the compiler works perfectly by converting floating point numbers into long and then moving them into desired register. But if we define floating point move insn for `SFmode`, the compiler crashes giving error. We are still fixing this error.