Improving GCC Retargetability

Extraction of Machine Independent RTL patterns from GCC's Machine Description Files

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by
Saravana Perumal P
Roll No. : 10111033

under the guidance of
Dr. Amey Karkare

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

June, 2012

CERTIFICATE

It is certified that the work contained in this thesis entitled
"Improving GCC Retargetability, Extraction of Machine Independent RTL patterns from GCC's Machine Description Files.",
by *Saravana Perumal P(Roll No. 10111033)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 18/6/2012

(Dr. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

June, 2012

Abstract

GCC (GNU Compiler Collection) has been one of the most popular compiler infrastructure for many years. It is used across languages, architectures and operating systems. The availability of GCC ports for a large number of targets stands testimony to the success of the retargeting model of GCC. With a large number of embedded systems now being developed and released for use across various fields, the process of retargeting GCC assumes importance. To port GCC to a new architecture, a Machine Description(MD) file that has the mapping from GCC's intermediate form to the target assembly code is to be written.

Constructing an MD file is a difficult task partly because of its large size, but mainly due to the need to understand the intermediate representations of GCC, while simultaneously having a good grasp of the target architecture. Due to these difficulties, the process of writing machine descriptions has become an ad hoc one. Developers retargeting GCC tend to copy MD files of machines similar to the target machine and modify them, making the whole process a trial and error method.

In this thesis, we demonstrate that MD files of machines with similar architecture exhibit significant amount of similarities. We have created a tool MDParser, to extract RTL patterns from MD files of some known machines. Using this tool we compare the similarity of patterns across machines. We have further created a framework that can use these extracted patterns and with user intervention, can help in the construction of new RTL templates. We also show how this framework can be used to build a tool that can help a developer in the construction of MD files for a new architecture, thus simplifying the retargeting process.

Dedicated to
my parents and my brother.

Acknowledgement

I would like to express my sincere gratitude towards my thesis supervisor Dr.Amey Karkare for his constant support and encouragement. I am grateful for his patient guidance and advice in giving a proper direction to my efforts. I would also like to thank the faculty and staff of the Department of CSE for the beautiful academic environment they have created here.

I am indebted to all my friends of the Mtech 2010 Batch for making these past two years a memorable one for me. I have been fortunate to be with some really interesting, fun people. I especially thank Adarsh, Ajith, Anvesh, Ashendra, Chitti, Keerthi and Vinay with whom I shared many a good times. Their company and their ability to infuse humor into any situation helped me get through some tough times.

Last, but not the least, I would like to thank my parents and my elder brother for their love, constant support and encouragement. Without their support and patience this work would not have been possible.

Saravana Perumal

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are living in a time where electronic systems are pervasive in every aspect of our life, e.g., industrial automation, automobiles, telecommunication systems, consumer electronics, military, agriculture, to name a few. New computing architectures are coming into the market every day to meet this huge demand for programmable automated devices. Writing applications for these architectures, requires that the designer of such architectures, makes available a reliable compiler to begin with. The ever expanding market, places a huge demand for such devices and the need to beat competition, results in a very less turn-around time available for this task. So manufacturers usually go for a retargetable compiler. A retargetable compiler is one, which can be configured to produce code for a new architecture. GCC is one of the most popular retargetable compilers.

GCC stands for GNU Compiler Collection. GCC is an integrated distribution of compilers for several programming languages[11]. GCC is one of the most widely used compilers for developing applications that run across several different architectures and operating systems. One of the strengths of GCC is, it is highly portable. This is because GCC was built so that it can be easily ported to any machine where *int* is at least a 32-bit type[2]. GCC gets most of its information about the target machine from the Machine Description(MD) files written for that machine[2]. GCC does not have any machine specific code, but has parameters which depend on the target machine's features[2]. This is considered as an elegant way of alienating the machine specific details from the compiler design.

1.1 Motivation

GCC parses the source language, converts it to an intermediate representation GIMPLE. After a few internal transformations, GCC converts the GIMPLE representation to another intermediate representation, RTL(Register Transfer Language). Finally, the RTL representation is converted to the target assembly code after performing a few transformations. MD files have the mapping from GIMPLE representation to RTL notation and from RTL notation to the machine's assembly code. So constructing the MD file for the target architecture forms the most important step in porting GCC. The MD file represents the machine instructions in an algebraic formula notation. Writing a MD file for a new architecture needs a good understanding of the RTL notation, and the instruction set of the target architecture, which is very difficult. So, in general an MD file for a new architecture is constructed from an MD file of a similar architecture by making modifications to suit the needs. Describing instructions in this method is observed to be quite complex, verbose, repetitive and assumed to be more of a trial-and-error method.[7, 9, 10].

The MD files for popular architectures are typically huge(running into tens of thousands of lines). So, while writing MD files from scratch for a new architecture, a single mistake will result in the compiler producing wrong or worse, inefficient code, without the user realizing it quickly. The problem could be fixed if the process of writing MD files is fully automated. But given the complexity of the architectures and the variety of architectures available it is virtually impossible to fully automate this process.

The MD files are huge because, they not only tell about the target machine, they also contain details for generating the expander (GIMPLE to RTL converter, explained later in Chapter 2) and recognizer(RTL to target instruction generator). Apart from this, they have instructions that help in instruction scheduling, peephole optimizations, register allocation, etc.,[3]. RTL notation is used in achieving the above mentioned tasks. Hence, finding RTL expressions that are common across machines which can be reused in writing new machine descriptions will be of immense help.

1.2 Contribution of This Thesis

Typically, the intermediate languages used by compilers are assumed to be the language for an abstract machine. For a class of architectures serving similar purposes, say typical general purpose processors, we expected the intermediate language to show significant amount of similarities. So, if we can find out intermediate representations that are common across typical machines(minimal set), and group them logically, the process of retargeting can be made simpler and systematic. With the minimal set in hand, we believe parts of MD files for new architectures can be generated automatically. Take for example, the scenario where we wanted to port GCC to a new general purpose processor. If we have this minimal set for the class of general purpose processors, we can start off by first writing MD instructions that map this minimal set to the target architecture. Once this is done, we can deal with those intermediate representations that are not part of the minimal set and is unique to this architecture and map them to the target architecture.

Davidson and Fraser[6] proposed a compiler architecture that was able to produce portable, optimizing compilers that can generate good code without the need for a machine-dependent optimizer. Like many portable compiler models, even in Davidson Fraser model, the front end parses the source code and produces an intermediate code. Machine-independent code optimizations are done on this intermediate code(code for abstract machine). Retargeting involves, writing the mapping from abstract machine code to target machine code. They made an interesting observation about the abstract machine. While deciding on this abstract machine, there are two choices. One is, letting the abstract machine support a set of features which is almost equivalent to the union of sets of features supported by typical real world machines. Such abstract machines are called *union machines.* The second choice is, letting the abstract machine support a set of features which is roughly equivalent to the intersection of features offered by typical machines. Such a machine is called an *intersection machine*. They argued that an intersection machine is easier to implement and more stable than a union machine. We believe our hypothesis about a minimal set for a class of machines, is equivalent to Davidson and Fraser's idea for an intersection machine.

To verify our hypothesis we did the following,

1. Read MD files for existing architectures from the back ends of GCC's source tree. We looked for patterns in them (RTL templates), which can be identified as machine-independent. A MD file parser was written for this purpose.

2. The collected patterns were classified logically, viz., arithmetic, logical, conditional branches, etc.

3. We tested the possibility of instantiating these patterns with machine-specific values to get back the RTL templates back and succeeded.

4. Rewrote MD files in terms of patterns and parameters, compiling them to get MD files with regular RTL templates from them.

We tested the above approach on MD files from five architectures, ARM, i386, MIPS, SPARC and VAX and the results were promising.

1.3 A Motivating Example

Let us look at two define_expand expressions, the first one is from MIPS and the next is from ARM. The examples given are modified for ease of explanation and to avoid referring to complex concepts not yet introduced.

Example 1.
```
(define_expand "addsi3"
  [(set (match_operand:SI 0 "register_operand")
(plus:GPR (match_operand:SI 1 "register_operand")
  (match_operand:SI 2 "arith_operand")))]
  "")

(define_expand "addsi3"
  [(set (match_operand:SI 0 "s_register_operand" "")
(plus:SI (match_operand:SI 1 "s_register_operand" "")
 (match_operand:SI 2 "reg_or_int_operand" "")))]
  "TARGET_EITHER"
  "
  if (TARGET_32BIT && GET_CODE (operands[2]) == CONST_INT)
    {
      arm_split_constant (PLUS, SImode, NULL_RTX,
                  INTVAL (operands[2]), operands[0], operands[1],
  optimize && can_create_pseudo_p ());
      DONE;
    }
  "
)
```

The RTL template of addsi3 expression from MIPS is depicted in figure 1.1. The table 1.1, splits this RTL template into RTL pattern and parameters.

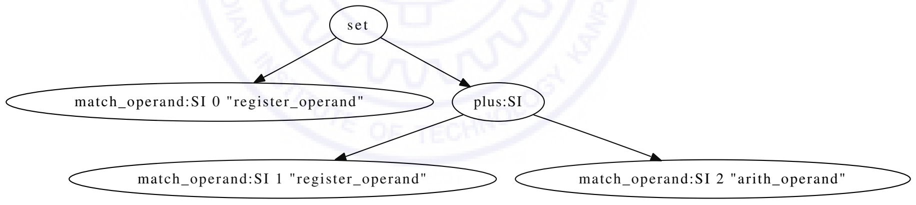

set
match_operand:SI 0 "register_operand"
plus:SI
match_operand:SI 1 "register_operand"
match_operand:SI 2 "arith_operand"

Figure 1.1: MIPS addsi3 Template

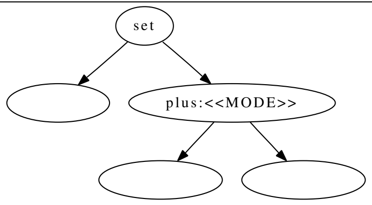
addsi3-Pattern	Parameter
	(match_operand:SI 0 ”register_operand”) SI (match_operand:SI 1 ”register_operand”) (match_operand:SI 2 ”arith_operand”)

Table 1.1: MIPS Patterns and Parameters

Similarly, figure 1.2 depicts the RTL template of addsi3 expression from ARM, while table 1.2 shows the RTL pattern and parameters for the same. We can observe from the tables that both these expressions, share the same pattern. They differ only in the parameters.

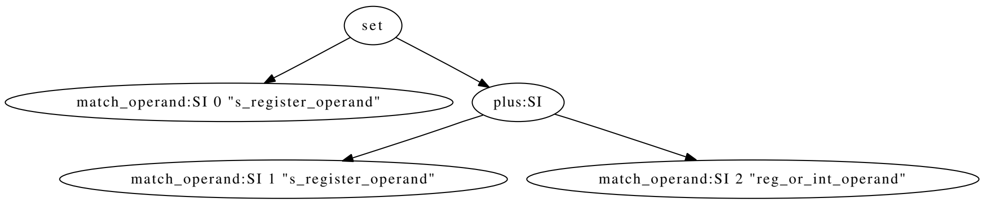

set
match_operand:SI 0 "s_register_operand"
plus:SI
match_operand:SI 1 "s_register_operand"
match_operand:SI 2 "reg_or_int_operand"

Figure 1.2: ARM addsi3 Template

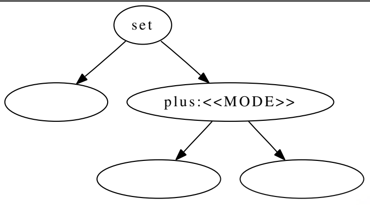
addsi3-Pattern	Parameter
	(match_operand:SI 0 ”s_register_operand”) SI (match_operand:SI 1 ”s_register_operand” ””) (match_operand:SI 2 ”reg_or_int_operand” ””)

Table 1.2: ARM Patterns and Parameters

We can see that, the pattern extracted in both the cases are the same. The pattern, extracted thus can go as part of the minimal set, more specifically to the list of arithmetic patterns. So while writing a new port, this is one of the arithmetic patterns that needs to be filled with machine-specific values.

Another possibility arises from the above method of splitting the RTL templates into patterns and parameters. The MD file's RTL templates exhibit significant amount of redundancies[9]. Patterns that repeats itself across several RTL templates can be declared once and used/instantiated with parameters every time it reoccurs. This can reduce the redundancies found in MD files. We'll look at this possibility in more detail in chapter 3.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2, reviews the compilation model of GCC and also the back-end organization of GCC. Chapter 3, presents the methodology we have adopted. Chapter 4, presents the results and an analysis of the same. Chapter 5, gives a brief description of the related work. The conclusion is provided in Chapter 6.

Chapter 2

Background

This chapter explains about the traditional compilation model, the Davidson Fraser model and how GCC differs from the traditional model and the effect of this on the retargeting process of GCC. This also introduces some basic concepts needed to understand GCC's machine description files and the RTL expressions used in them.

2.1 Aho Ullman Model

The Traditional Aho Ullman Model of compilation[4] advocates grouping the various phases of compilation into three logical units, the front end, the optimizer and the back end as seen in figure 2.1

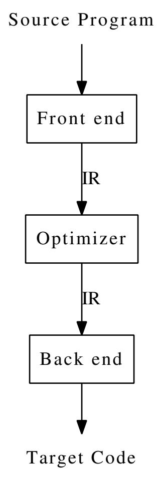


Figure 2.1: Aho Ullman Model of Compilation

The front end, converts the source language into an intermediate representation which is both machine-independent and language independent. The intermediate language, can be considered as a language for an abstract machine. The optimizer, works on the intermediate language and improves on it. The back end, also known as the code generation phase, does the work of instruction selection, instruction scheduling, register allocation. The front end isolates the source language specific issues while the back end isolates the machine dependent features from the rest of the compiler. This is supposed to make the process of retargeting such a compiler relatively easy. The optimizer which works on the language and machine-independent intermediate representation can be reused across any combinations of source language and target machine.

In this model, to retarget the compiler to a new machine, only the back end has to be rewritten. To be more specific, the instruction selection part of the back end has to be rewritten. The instruction selector does a pattern matching over the compiler's intermediate representation and executes a code fragment associated with this pattern, which produces the assembly code. So a knowledge of the compiler's intermediate representation and the target machine's instruction set is needed to write this new instruction selector. In this model, the back end has to be written very carefully, since the target code generated by it, determines the efficiency of the source program as no further optimizations are done over it.

2.2 Davidson Fraser Model

The Davidson Fraser Model[6], pitched for a compiler that optimizes the code generated by the code generator.

As seen from figure 2.2, the front end here converts the source code into an abstract machine code. The abstract machine code is machine independent. The expander converts the abstract machine code to a set of register transfers. The expander is so named because it takes a single input instruction and converts it into a sequence of register transfers. Register transfer is an intermediate representation

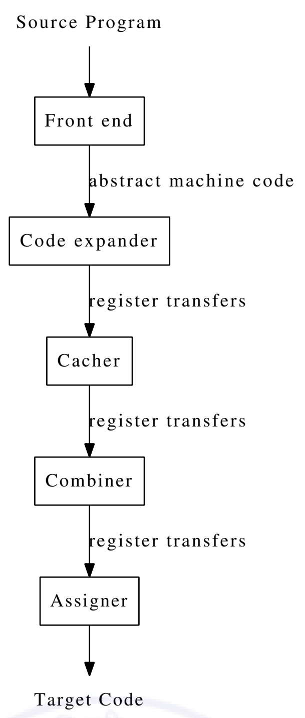


Figure 2.2: Davidson Fraser Model of Compilation[6]

that is machine dependent. It gets its machine dependent parameters from machine description files. Since the expander is followed by the Cacher, which is an optimizer that acts on register transfers, the expander can be written in a simple way so that it outputs a naive sequence of instructions. The combiner's work is to combine a sequence of register transfer instructions and produce a single instruction wherever possible. The assigner finally produces the assembly code.

Here the cacher is machine independent, though it works on register transfers which is a machine dependent code. This is because of the fact that the register transfers are machine specific but their form is machine independent[6]. The cacher takes advantage of this fact and works on the form of the register transfers. This object code optimizer makes it possible to write a simple expander, which need not produce an efficient code and only has to produce the correct code. Retargeting involves rewriting the expander and the assigner, which are easy tasks since both use simple algorithms.

2.3 GCC's Model

GCC's compilation model is a modified version of the Davidson Fraser compilation model. The various phases in its compilation process can be split into three phases[1],

- Front end
- Target independent code transformations
- Code generation

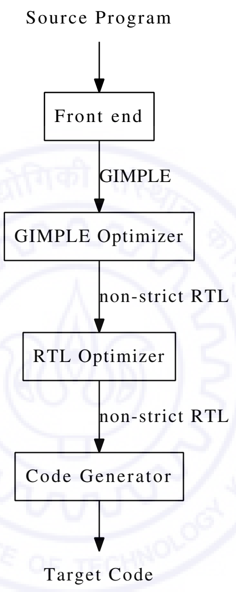


Figure 2.3: GCC's Model of Compilation

Like in the Davidson Fraser Model, the front end here transforms the source code to a machine independent representation, GIMPLE. The second phase, performs several SSA based optimizations on the GIMPLE code, then performs expansion, i.e., conversion of GIMPLE code to what is called the non-strict Register Transfer Language(RTL). The non-strict RTLs are so named because, in the second phase of compilation, the RTL patterns matched from constructs like *define_insn*s are matched without their corresponding operand *constraints* being satisfied. Finally optimizations are performed over this RTL and given to the next phase. The code generation phase converts the non-strict RTL to strict RTL (unlike non-strict RTL, in strict RTL it is made sure that the patterns that match also satisfies all the operand constraints), schedules instructions , performs peephole optimizations and finally generates assembly code.

The machine description files are used in expansion and code generation operations. In the phase where the expand operation is done, the RTL templates from the named define_insns and define_expands are used. In the code generation phase, where the strict RTLs are converted to assembly code, the names are ignored and the define_insn which contains the current RTL pattern as its template is matched and the corresponding assembly code is emitted.

2.4 GCC's Back End Organization

A back end for a target `machine` in GCC must contain the following files and directories apart from other files which are not of interest now,

1. A directory named `machine` under gcc/config and the following files under this directory `machine`.

2. A file named `machine`.md.

3. A file named `machine`.c.

4. A header file `machine`.h and `machine`-protos.h.

The file `machine`.md contains a list of instruction patterns that the machine supports. The header file `machine`.h, contains macros which describes the machine's properties and which could not be included in the `machine`.md file. The source file `machine`.c contains a variable targetm, which is a structure with pointers to datastructures and functions that are related to the target machine. Also it must contain definitions of the datastructures and functions that are not defined elsewhere.

The machine description file contains a set of expressions of the form `define_*`. They are `define_insn`, `define_peephole`, `define_split`, `define_insn_and_split`, `define_peephole2`, `define_expand`, `define_delay`, `define_asm_attributes`, `define_cond_exec`, `define_predicate`, `define_special_predicate`, `define_register_constraint`, `define_constraint`, `define_memory_constraint`, `define_address_constraint`, `define_cpu_unit`, `define_query_cpu_unit`, `define_bypass`, `define_automaton`, `define_reservation`, `define_insn_reservation`, `define_attr` and `define_enum_attr`.

Here we are interested in those expressions which as part of their format have RTL templates to match with. So the list above shrinks to

- define_insn
- define_peephole
- define_split
- define_insn_and_split
- define_peephole2
- define_expand

2.4.1 RTL Expression Codes

The RTL templates in these expressions tell which instructions match these patterns and also how the operands can be found[2]. The current work involves parsing this RTL template and extracting out patterns from them, which can be instantiated with parameters. RTL uses five different types of objects, viz., expressions, integers, wide integers, strings and vectors. Among the five listed above, expressions are of interest to us. The gcc internals documentation provides a good description of RTL expressions and also the file rtl.def from the GCC source lists all the possible expressions. Each RTL expression has an expression code(RTX code) associated with it. The RTX code is machine independent. The RTX code gives information like, how many operands the expression contains and what are the types of each operand.

The RTX codes are grouped into classes based on the type of operation, operands. For example, the class RTX_COMM_ARITH, contains all the RTX codes which perform commutative binary operations like PLUS, AND, etc. The file rtl.def contains a series of DEF_RTL_EXPR() macro statements, each of which defines an RTL expression. The macro has four operands,

1. The internal name of the RTX.
2. The name of the RTX in ASCII format.
3. The print format of the RTX.
4. The class of the RTX.

The third operand is a sequence of characters called the format. This format enumerates the number of objects that the RTL expression contains and the type of each of the objects. For example, the format for define_insn is sEsTV. Each character in a format has a particular meaning. The meaning of some of the most used format characters are listed below,

s - A string.

e - An RTL expression.

i - An integer.

E - A vector of RTL expressions.

V - This is same as E, but this object is optional.

T - This is like strings, but treated speacially by rtl_reader used internally by gcc.

The format of the define_insn expression can be listed as below.

```
(define_insn string
    [RTL Expression Vector]
    string
```

string
[Optional RTL Expression Vector]
)

From the above list, it is understood that, define_insn(format is sESTV, as stated above) has a string, followed by an RTL expression vector, then two strings and finally an optional RTL expression vector. This is explained in more detail in the subsequent section. In summary, the file rtl.def, gives the syntax of each RTL expression while the semantics can be understood from the comments in rtl.def and GCC's documentation[2].

2.5 Standard Pattern Names

Names are given to patterns defined by define_expand expressions and optionally to define_insn expressions also. Two different patterns can not use the same names i.e., the names are unique. GCC provides a set of standard pattern names, whose semantics are well defined and is used in the RTL generation phase. If the name matches a standard pattern name, gcc calls the function gen_name implicitly that generates the RTL pattern for that name. Names can start with a *, so that they can be traced in debugging operations, as such names can be easily traced in RTL dumps while debugging. Such names won't be used in RTL generation phase.

2.6 RTL Templates

An RTL template is a vector of incomplete rtl expressions, as it contains a number of unresolved operands which have to be resolved by expressions like match_operand, match_dup, etc. If the vector has only one RTL expression, then that expression is the template of the instruction pattern. If the vector has multiple expressions, then it is as though there is a *parallel* construct that executes all the expressions in parallel. An RTL template tells which instructions match the particular pattern and how to find its operands. For a named pattern, it also tells how to construct the instruction from the operands. To construct an instruction, the operands must be substituted into a copy of the template. An RTL template uses rtl expressions like match_operand, match_dup, match_operator,etc. Let us look at the format of a match_operand expressions

```
(match_operand:m n predicate constraint)
```

This expression acts as a placeholder for the operand number 'n' of the instruction. If an instruction is being constructed, the operand number 'n' will be substituted here. If an instruction is being matched, the operand appearing at this position must satisfy the predicate mentioned, else the instruction won't be matched. Predicate is a string that is a name of a function that accepts two arguments, an expression and the machine mode 'm'. If the predicate is empty, then no tests need to be done. Constraints allows a more detailed selection after the operand satisfies the predicate condition. Constraints can be used to decide on reloading and the register class choices. The operand numbers are numbered from zero. For example, let us look at the following match_operand expression,

Example 2. `(match_operand:SI 0 ‘‘register_operand’’ ’’r’’)`

This expression will try to match operand zero with machine mode Single Integer mode(SI). The predicate function to be used is `register_operand` and the constraint to be tested is `r`. The constraint here means that the operand must be a register and it must be one of the general registers.

Now let us look at the define_∗ expressions which have RTL expressions in them and see how they are used.

2.6.1 define_insn Expression

The define_insn expression is very important because this describes an instruction pattern. Let us look at what constitutes a define_insn expression and the meaning of each of its constituents. A define_insn expression contains,

1. An optional name. If the name is present, and if this matches one of the standard names that the compiler is aware of, this define_insn will be used in the RTL generation phase. If the name is not present or if the name is not a known name, then this define_insn will not be used in the RTL generation phase, but can be used in other phases.

2. An RTL template. This is same as the RTL template that was discussed in the previous section.

3. A condition. This is a C expression, which is the final test to be performed to decide whether the instruction body matches this pattern.

4. The output template. This is a string that says how to output the assembly code for the matching instructions.

5. An optional vector of attributes.

A sample define_insn expression is listed below.

Example 3.

```
(define_insn "add<mode>3"
  [(set (match_operand:ANYF 0 "register_operand" "=f")
(plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
   (match_operand:ANYF 2 "register_operand" "f")))]
  ""
  "add.<fmt>\t%0,%1,%2"
  [(set_attr "type" "fadd")
   (set_attr "mode" "<UNITMODE>")])
```

2.6.2 define_expand Expression

A define_expand expression looks similar to a define_insn expression, but the former is used only in the RTL generation phase and it can produce more than one RTL instruction. The RTL instruction generated by a define_expand expression must match the RTL template of some define_insn expression. Otherwise, the compiler will crash when trying to generate code or during optimization. A define_expand expression contains,

1. A name. In a define_expand expression, the name is not optional, as in the case of a define_insn expression.

2. An RTL Template. The RTL template here has one different meaning. If the RTL template has more than one RTL expression, then it is not assumed that there is an implicit PARALLEL surrounding them.

3. A condition. This is same as in the define_insn expression, but the condition here depends more on the machine specific flags.

4. Preparatory statements. This is a string containing a set of C statements, which has to executed before the RTL code is emitted.

A sample define_expand expression is listed below.

Example 4.
```
(define_expand "mul<mode>3"
  [(set (match_operand:GPR 0 "register_operand")
(mult:GPR (match_operand:GPR 1 "register_operand")
  (match_operand:GPR 2 "register_operand")))]
  ""
{
  if (TARGET_LOONGSON_2EF || TARGET_LOONGSON_3A)
    emit_insn (gen_mul<mode>3_mul3_loongson (operands[0],
                                 operands[1], operands[2]));
  else if (ISA_HAS_<D>MUL3)
    emit_insn (gen_mul<mode>3_mul3 (operands[0],
                                  operands[1], operands[2]));
  else if (TARGET_FIX_R4000)
    emit_insn (gen_mul<mode>3_r4000 (operands[0],
                                  operands[1], operands[2]));
  else
    emit_insn
      (gen_mul<mode>3_internal (operands[0],
                                  operands[1], operands[2]));
  DONE;
})
```

2.6.3 define_split Expression

A define_split expression tells the compiler how to split a complex instruction into two or more simpler instructions. This splitting is necessiated under two circumstances. One is the case, where the machine may have instructions that require delay slots in between. The second case is, the output of some instructions won't be available for multiple cycles. In both these cases, the optimization phase of the compiler needs the ability to place instructions in slots that are empty. This is supported by the define_split expression. A define_split expression contains,

1. An instruction pattern(RTL Template). This is the pattern that is to be split.
2. A condition. The condition here is same as in a define_insn expresion.
3. Output instructions. This is a list of instruction patterns that will be generated. That is the input instruction pattern is split into a list of instructions given by this output instructions.
4. Preparatory statements. This is similar to those in a define_expand expression.

A sample define_split expression is listed below.

Example 5. `(define_split`
```
  [(set (match_operand:GPR 0 "d_operand")
(const:GPR (unspec:GPR [(const_int 0)] UNSPEC_GP)))]
  "TARGET_MIPS16 && TARGET_USE_GOT && reload_completed"
  [(set (match_dup 0) (match_dup 1))]
  { operands[1] = pic_offset_table_rtx; })
```

2.6.4 define_insn_and_split Expression

This is used, when the instruction pattern of a define_split expression matches exactly with that of a define_insn expression. So this expression contains all the fields from the define_insn expression and define_split expression. This will have two separate conditions, one for instruction splitting and another for instruction matching (used in define_insn).

A sample define_insn_and_split expression is listed below.

Example 6.
```
(define_insn_and_split "loadgp_newabi_<mode>"
  [(set (match_operand:P 0 "register_operand" "=d")
(unspec:P [(match_operand:P 1)
   (match_operand:P 2 "register_operand" "d")]
  UNSPEC_LOADGP))]
  "mips_current_loadgp_style () == LOADGP_NEWABI"
  { return mips_must_initialize_gp_p () ? "#" : ""; }
  "&& mips_must_initialize_gp_p ()"
  [(set (match_dup 0) (match_dup 3))
   (set (match_dup 0) (match_dup 4))
   (set (match_dup 0) (match_dup 5))]
{
  operands[3] = gen_rtx_HIGH (Pmode, operands[1]);
  operands[4] = gen_rtx_PLUS (Pmode, operands[0], operands[2]);
  operands[5] = gen_rtx_LO_SUM (Pmode, operands[0], operands[1]);
}
  [(set_attr "type" "ghost")])
```

2.6.5 define_peephole Expression

The use of this expression is deprecated.

2.6.6 define_peephole2 Expression

Peephole optimizations are done after the register allocation phase but before the instruction scheduling phase. The format of this expression is almost similar to the define_split expression except that the instruction pattern to be matched in this case is a sequence of instructions instead of a single instruction. This also tells, what additional scratch registers are required(if any) and their lifetimes.

A sample define_peephole2 expression is listed below.

Example 7.
```
(define_peephole2
  [(parallel
       [(set (match_operand:SI 0 "lo_operand")
     (match_operand:SI 1 "macc_msac_operand"))
(clobber (scratch:SI))])
```

```
   (set (match_operand:SI 2 "d_operand")
(match_dup 0))]
  ""
  [(parallel [(set (match_dup 0)
   (match_dup 1))
      (set (match_dup 2)
   (match_dup 1))])])
```

Chapter 3

Methodology

In this chapter we look at what is the form(pattern) of an RTL expression and how this could be separated from an RTL template. Then we see how MD files were tested for similarities. Finally we see, how a pattern can be instantiated with machine-specific parameters to construct a new RTL template.

3.1 Extracting the Form of an RTL Expression

Even though the Register Transfer Expressions in an MD file represents machine instructions for a specific machine, their form is machine-independent[6]. This section explains what is the form and how this form can be extracted from an RTL expression.

Let us take for example, a define_insn expression from mips.md file

Example 8.
```
(define_insn "add<mode>3"
  [(set (match_operand:ANYF 0 "register_operand" "=f")
(plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
   (match_operand:ANYF 2 "register_operand" "f")))]
  ""
  "add.<fmt>\t%0,%1,%2"
  [(set_attr "type" "fadd")
   (set_attr "mode" "<UNITMODE>")])
```

Here the RTL template is

```
 [(set (match_operand:ANYF 0 "register_operand" "=f")
(plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
   (match_operand:ANYF 2 "register_operand" "f")))]
```

This can be represented in a tree form as,

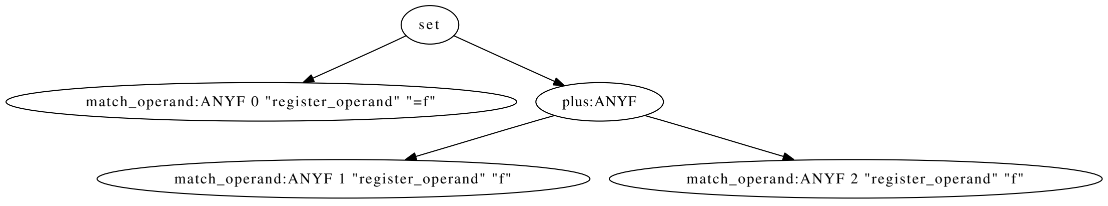

set
match_operand:ANYF 0 "register_operand" "=f"
plus:ANYF
match_operand:ANYF 1 "register_operand" "f"
match_operand:ANYF 2 "register_operand" "f"

Figure 3.1: Add Template

The form of the above RTL template can be represented as,

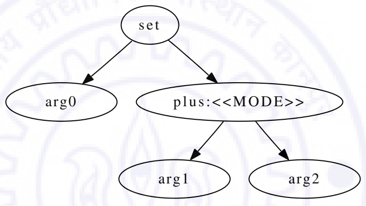


Figure 3.2: Add - Form of the template

The nodes that are labeled arg0, arg1, arg2, etc., can be termed as holes which can be filled by suitable parameters. The mode of the operator plus is also not part of the form, because, the modes supported are machine specific. Similarly, the match_operand expressions are machine-specific because, they have fields like predicate, constraint and mode. These fields tells how the operand should be chosen in a machine-specific way. We can see that the form depicted in figure 3.2, can be used by any machine, as it has no machine-specific parameters. Any machine can use this form after filling in details about the arguments for the operators set and plus and the mode supported by the operator plus.

Definition 1. An RTL template stripped off its machine-specific parameters, containing only the form of the template is called its *RTL pattern* or simply *pattern*.

The patterns when filled(instantiated) with suitable machine-specific parameters like modes or match_operand expressions can give the original RTL template.

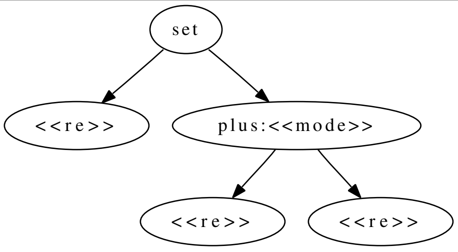
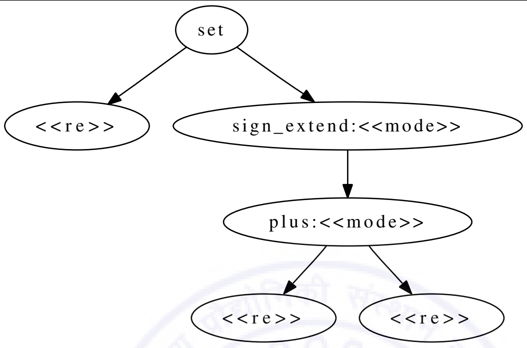
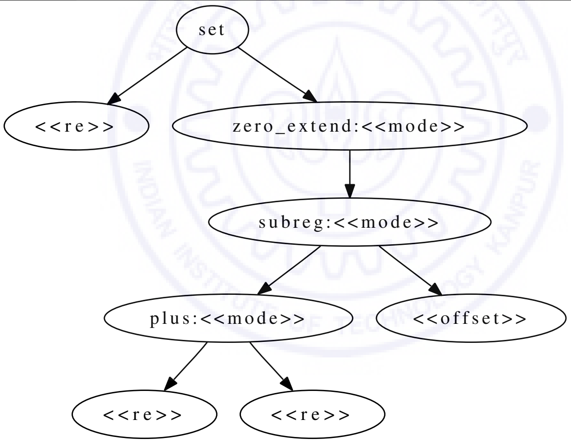
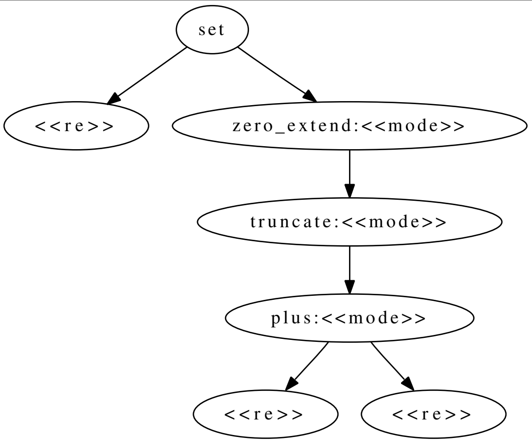
S.No	Pattern	No. of Occurrence
1		17
2	 set <<re>> sign_extend:<<mode>> plus:<<mode>> <<re>> <<re>>	2
3	 set <<re>> zero_extend:<<mode>> subreg:<<mode>> plus:<<mode>> <<offset>> <<re>> <<re>>	2
4	 set <<re>> zero_extend:<<mode>> truncate:<<mode>> plus:<<mode>> <<re>> <<re>>	1

Table 3.1: Summary of Patterns in Add instructions from mips.md

A look at the list of MD file instructions of MIPS, that deal exclusively with addition tells that, there are 7 define_insn instructions, 4 define_split instructions and one each of define_expand and define_insn_and_split instructions. These instructions together have 22 RTL templates. But when we extract the patterns from each of these 22 RTL templates, there are only 4 unique RTL patterns in total.

We see from table 3.1, that pattern1 repeats itself 17 times, with varying machine-specific mode and operand constraints. Extracting patterns from the addition instructions of the mips machine description was relatively simple. It was enough to extract the operators set, plus, zero_extend, sign_extend and truncate and excluding the modes and match_operand and match_dup expressions. But to extract patterns from the whole of machine description files, we need to classify what forms part of the pattern and what should be ignored.

The RTL expressions in rtl.def are classified into several classes. The classification is based on the type of their operation and their operands. Below is a list of RTL classes along with a brief description about each of them[2].

- RTX_OBJ: An RTX code that represents an actual object, such as a register (reg) or a memory location (mem, symbol_ref).
- RTX_CONST_OBJ: An RTX code that represents a constant object.
- RTX_COMPARE: An RTX code for a non-symmetric comparison, such as geu and lt.
- RTX_COMM_COMPARE: An RTX code for a symmetric (commutative) comparison, such as eq, ordered, etc.
- RTX_UNARY: An RTX code for a unary arithmetic operation, such as neg, not, abs, etc. This category also includes value extension (sign or zero) and conversions between integer and floating point.
- RTX_COMM_ARITH: An RTX code for a commutative binary operation, such as plus, and, ne, eq, etc.

- RTX_BIN_ARITH: An RTX code for a non-commutative binary operation, such as minus, div, ashiftrt, etc.

- RTX_BITFIELD_OPS: An RTX code for a bit-field operation. Currently only zero_extract and sign_extract are part of it.

- RTX_TERNARY: An RTX code for other three input operations. Currently only if_then_els, vec_merge, sign_extract, zero_extract, and fma are included.

- RTX_INSN: An RTX code for an entire instruction.

- RTX_MATCH: An RTX code for something that matches in insns, such as match_dup. These only occur in machine descriptions.

- RTX_AUTOINC: An RTX code for an auto-increment addressing mode, such as post_inc.

- RTX_EXTRA: All other RTX codes. This category includes the remaining codes used only in machine descriptions (define_*, etc.). It also includes all the codes describing side effects (set, use, clobber, etc.) and the non-insns that may appear on an insn chain, such as note, barrier, and code_label. subreg is also part of this class.

From the above list, the obvious choices for classes of RTL expressions which will be part of the patterns are,

- RTX_COMPARE

- RTX_COM_COMPARE

- RTX_UNARY

- RTX_COMM_ARITH

- RTX_BITFIELDS_OPS

- RTX_TERNARY

- RTX_AUTOINC

Apart from this, the form of an RTL expression contains a few RTL expressions from the class RTX_EXTRA. They are called *Side Effect Expressions*. They are so named, because they change the state of the machine. Side effect expressions include operators `set`, `return`, `call`, `clobber`, `use`, `parallel`, `cond_exec`, `sequence`, `asm_input`, `unspec`, `unspec_volatile`, `addr_vec` and `addr_diff_vec`.

An *internal node* of a tree is that node that has child nodes. We have seen quite a few examples where the RTL expressions and templates where represented as trees. The RTL expressions that are part of the pattern are those that appear as internal nodes in the tree form of the RTL expressions. The RTL expressions which are leaves in an RTL tree form, are constants, registers or memory locations which are chosen in a machine specific way. In summary, the RTL expressions which are part of the pattern, tells what operation is being done, and this has the same meaning across machines.

So, to extract the pattern from an RTL template, we must traverse the tree form of it and retain all nodes which form part of the pattern and create holes in place of nodes which are machine specific.

The list of unique patterns encountered in a MD file are listed in a file, along with the following details.

- The code iterators used in the machine. This is important, because the code iterators acts as aliases to operators and this alias is specific to a particular machine. So reading the patterns back won't make much sense without the code iterators listed.

- The height of the pattern. The height here is height of the tree that represents this pattern.

- The number of times this particular pattern is encountered in the files that are parsed.

- The total number of RTL templates considered for parsing.

A function, AddToPatternList, is used to get the list of unique patterns found in the machine description files of a machine. This function(refer Algorithm1) maintains a list of patterns, sorted in the increasing order of their heights. When a new pattern is encountered, it is compared with patterns of same height for equality and added to the list, if it is not already present. The patterns are sorted by their heights, so that unwanted comparison between patterns of dissimilar heights are avoided.

Algorithm 1 AddToPatternList (p)

```
$height := get\_height(p)$
if plist is empty then
    Add p to plist. {plist, is a global variable and points to a list of patterns.}
else
    Skip over all patterns,pat, such that $get\_height(pat) < height$.
    if There are patterns pat, such that $get\_height(pat) \geq height$ then
        while As long as plist has more patterns and current pattern, $pat \rightarrow height = height$ do
            Compare the input pattern p and pat for equality.
            if $p = pat$ then
                Break out of loop.
            end if
        end while
        if plist has more patterns then
            if $pat \rightarrow height = height$ then
                The input pattern, p, is already added to the list.So just increase the count for this pattern.
            else
                The input pattern p, is not in the list. Insert this pattern to the list.
            end if
        else
            None of the patterns in the list matches the input pattern and it is safe to add this pattern to the end of the list.
        end if
    else
        All the patterns in the plist are of height less than the input pattern. So insert this pattern to the end of the list.
    end if
end if
```

The input to the above function is a pattern p. To get the pattern p, from an RTL Template of an instruction, say a define_insn expression, the RTL template is passed to a function extractPattern, which accepts an RTL template and returns the corresponding pattern. The pattern thus returned is passed to the above function.

3.2 Output Format

After parsing the machine description files of a machine, the output is written on to a file. Initially, the code iterators found in the machine are listed down. Then the patterns are listed one by one in the increasing order of their heights. Along with the patterns, their height and the number of times that pattern occurs in the MD file are also listed.

```
Pattern_format : code_iterators patterns
               ;
code_iterators : //Empty pattern
                |
                 code_iterators code_iterator
                ;
code_iterator   : [SNO] code_iterator_expression
                ;
patterns ://Empty Pattern
                |
                 patterns pattern
                ;
pattern  :[SNO][HEIGHT][COUNT] pattern_expression
                ;
```

A grammar is written to read code_iterator_expression and also the pattern_expression, so that they can be read back from file and represented as trees in memory for further use.

For the addition instructions of the machine MIPS, the patterns extracted and printed will look like this

```
[1][2][17] (set <<:m>> (<<re>>)(plus <<:m>> (<<re>>)(<<re>>)))
```

```
[2][3][2] (set <<:m>> (<<re>>)(sign_extend <<:m>>
                                 (plus <<:m>> (<<re>>)(<<re>>))))
[3][4][2] (set <<:m>> (<<re>>)(zero_extend <<:m>>
                                (subreg <<:m>>
                                (plus <<:m>> (<<re>>)(<<re>>))
                                 <<offset>>)))
[4][4][1] (set <<:m>> (<<re>>)(zero_extend <<:m>>
                                 (truncate <<:m>>
                                 (plus <<:m>> (<<re>>)(<<re>>)))))
```

The instructions defining addition operation don't use code iterators. So the code iterators are not listed above. In the first line, the value [1] represents the index, the value [2] represents the height of the pattern and the value [17] represents the number of times this pattern occurs in the input file that was parsed. Followed by this, the pattern is listed. Any text within the two angle brackets represents machine specific values which are to be filled to get the RTL template. The textual form of the patterns is similar to the textual form of the RTL expressions found in machine description files. They are written in similar fashion, with the parentheses used in the same way as in a regular RTL expression. In the textual form of RTL, any RTL expression is written within a pair of parenthesis. An RTL vector, which is a vector of RTL expressions, is enclosed within a pair of square brackets. The only difference in the pattern's representation is that any machine-specific value is removed, and is replaced by a meaningful text enclosed within two angled brackets. So anyone familiar with RTL expression's textual form can easily read and understand the pattern files.

3.3 Finding Common Patterns between two machines

One interesting application of the parsing discussed above is, given two machines, we can get an idea about the number of patterns that are common between them. Note that, with each pattern p_i, we associate a count. This count, tells us the number of actual templates that have this pattern as its form, or conversely, the number of templates that can be formed on this machine by supplying machine-specific parameters to this pattern.

Given two files, each containing the pattern list of a machine, a list of common patterns is output into a file with the following details,

- The list of code_iterators common between the two machines.
- The list of patterns that are common between the two machines.

The above listing has the same syntax as the pattern list file discussed in the previous section. The count associate with each pattern in this case means the number of templates that are common in both the machines.

3.4 Instantiating Patterns with Parameters

This section describes how to instantiate a pattern with machine-specific parameters to generate a complete RTL template. For this we need the list of patterns which was output from the *parse* option or the *intersect* option. We refer to the pattern to be instantiated with a number which indicates the index at which the pattern is listed in the pattern file. The first pattern listed gets the number 1, the second 2 and so on. To instantiate a pattern we use the notation,

```
[$$INDEX ARGUMENTS]
```

The INDEX above represents the index of the pattern in the pattern_list. The ARGUMENTS, is list of machine-specific values that are to be used by the pattern indicated by index to complete the pattern to get the complete RTL Template. The arguments are listed in an in-order fashion. The use of parenthesis and square brackets is similar to the RTL's textual representation. The arguments for an RTL expression are listed within a pair of parenthesis, while that of an RTL vector are enclosed within square brackets.

Let us illustrate this with an example. The addition specific instructions from the MD file of the MIPS machine is parsed and we get the below list of patterns.

```
[1][2][17] (set <<:m>> (<<re>>)(plus <<:m>> (<<re>>)(<<re>>)))
[2][3][2] (set <<:m>> (<<re>>)(sign_extend <<:m>>
                                (plus <<:m>> (<<re>>)(<<re>>))))
[3][4][2] (set <<:m>> (<<re>>)(zero_extend <<:m>> (subreg <<:m>>
                                (plus <<:m>> (<<re>>)(<<re>>))<<offset>>)))
[4][4][1] (set <<:m>> (<<re>>)(zero_extend <<:m>>
                                (truncate <<:m>>
                                 (plus <<:m>> (<<re>>)(<<re>>)))))
```

To instantiate one of the patterns above, say pattern 3, with operands, we can use the below construct,

```
[$$3(null (match_operand:SI 0 "register_operand" "=d")
        (SI (QI
    (SI (match_operand:SI 1 "register_operand" "d")
 (match_operand:SI 2 "register_operand" "d"))
     3)))]
```

In the above statement, $$3 corresponds to the pattern 3 in the pattern list, listed above. On reading the pattern, a tree, say T1 as shown in Figure3.3, of the pattern is created internally. Followed by this, the arguments to this pattern are listed. A grammar has been written to parse the arguments. This grammar reads the arguments and creates a tree say T2 as shown in Figure3.4, for this argument. The next step will be to merge these two trees T1 and T2, the pattern and the argument to get the actual template with the complete machine-specific values as shown in Figure3.5.

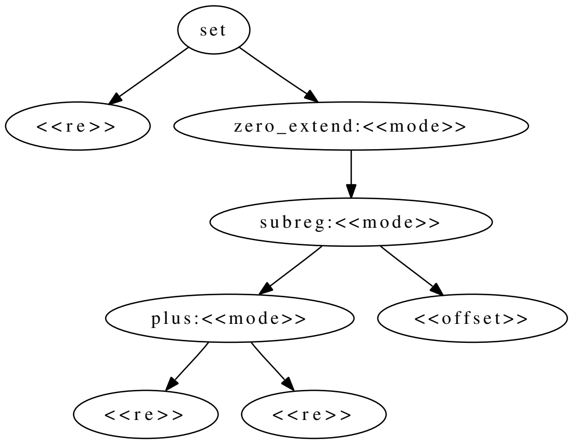


Figure 3.3: Pattern-Tree T1

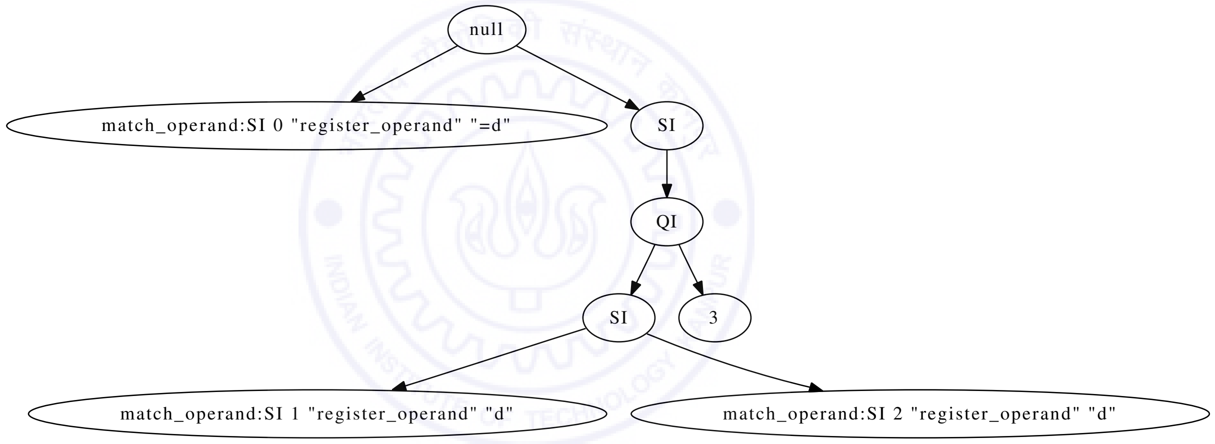

null
match_operand:SI 0 "register_operand" "=d"
SI
QI
SI
3
match_operand:SI 1 "register_operand" "d"
match_operand:SI 2 "register_operand" "d"

Figure 3.4: Argument-Tree T2

3.5 Merging Pattern Files

This section describes how a set union of patterns is performed given a list of files with pattern lists. Previously we saw that a grammar was written to read pattern files using which common patterns between two machines were listed. The same grammar is used to find the union of patterns given two pattern files. If we have the pattern-list files for a set of architectures, with our capability to perform union and intersection operation on patterns, we can create a basis set of patterns that are common to a majority of the architectures.

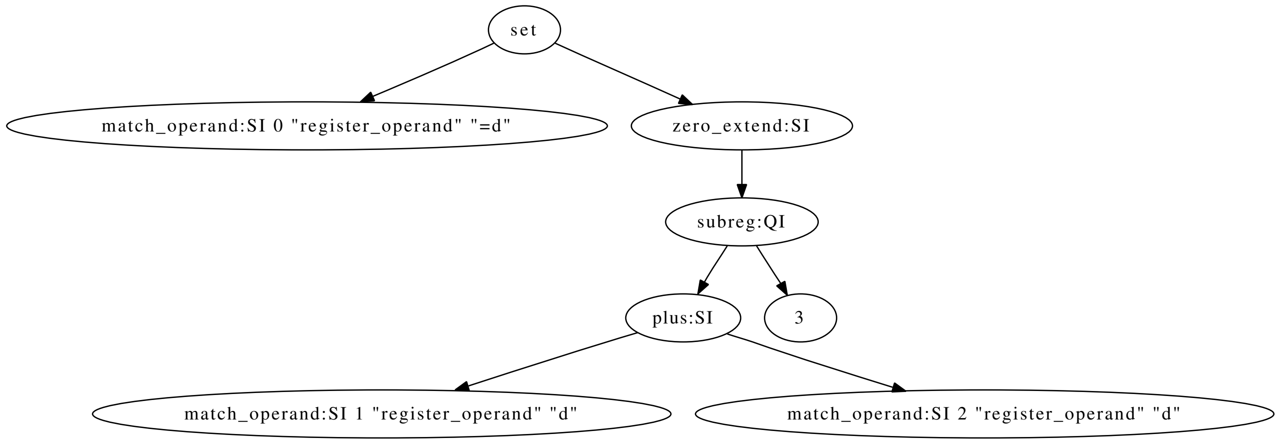

set
match_operand:SI 0 "register_operand" "=d"
zero_extend:SI
subreg:QI
plus:SI
3
match_operand:SI 1 "register_operand" "d"
match_operand:SI 2 "register_operand" "d"

Figure 3.5: Merged Tree

A naive approach for getting the minimal set is suggested below. Say, there are pattern-list files f1, f2, f3,..,fn for machines m1, m2, m3,...,mn. We can modify the count associated with each pattern as 1. Then we can perform an union of all the pattern files and then output only those patterns which occurs in at least in a specific number(threshold), say half the number of machines. This threshold can be made configurable by the user. With this in place, we can get the basis set of patterns needed to build new MD files.

3.6 Separating Machine-specific parameters from Templates

We developed a method to split an RTL template into patterns and parameters i.e., given an RTL template in its input, it splits the RTL template into the index value of its corresponding pattern and the machine-specific values as operands. The RTL templates in this format is written to an output file.

Developing this method served two purposes. The first one was to test our strategy to instantiate RTL patterns with parameters as discussed in the previous section, using real MD files. The MD files of existing machines were first split into pattern-index and machine-specific operands using this method and written to an output file. Then we tested creating RTL templates back from the output file created in the previous step.

The second purpose was in removing redundancies in MD files. Later in the next chapter, we'll show through experimental results that, patterns are repeated across RTL templates even within MD files of a single machine. Since, patterns are extracted and listed only once, this method helps removing redundancies.

3.7 Summary

In summary, we looked at what is a pattern of an RTL expression and how this could be separated from an RTL template. Then we saw how MD files were tested for similarities and how the patterns can be instantiated with machine-specific parameters to construct new RTL templates.

Chapter 4

Experimental Results

This section describes the results that we obtained using the tool MDParser, we developed to extract RTL patterns from MD files. Recall that patterns are obtained by extracting the form of the RTL templates.

The Machine description files of five machines

1. ARM
2. i386
3. MIPS
4. SPARC
5. VAX

were considered for the results enumerated below. These files are taken from the back-end of GCC version 4.6.1.

4.1 Extracting Patterns From MD Files

Table 4.1 lists the number of RTL templates considered in each of the machine's machine description file and the number of patterns that form the basis of it.

From table4.1 it is observed that in general, the number of patterns are around one fourth of the number of templates within a given machine description. This tells

Machine	No. of Templates	No. of Patterns
ARM	1581	362
i386	2238	547
MIPS	736	209
SPARC	701	187
VAX	125	64

Table 4.1: Summary of Patterns and RTL Templates

us about the level of redundancy that is present within machine description files. For example, ARM has 1581 RTL templates, but only 362 patterns.

4.2 Common Patterns Between Machines

Table 4.2 lists the number of patterns that are common between the machines. This was obtained by performing an intersection of the pattern files extracted from each machines after parsing.

	Arm	I386	Mips	Sparc	Vax
Arm		101	75	79	35
I386	101		73	63	34
Mips	75	73		48	29
Sparc	79	63	48		30
Vax	35	34	29	30	

Table 4.2: Common Patterns

Suppose we have a pattern p from machine m1 which matches with pattern p from machine m2. Let c_i be the count associated with p of m1 and c_j be the count associated with p of m2. There are a total $c_i + c_j$ templates in the machines m1 and m2 sharing the same pattern p_i(or p_j). This is listed in table 4.3.

From the table 4.3, it is observed that on an average 50% of the templates share common patterns. We can note that the results of VAX are not as good as others. We believe the reason for that is, VAX is the smallest machine considered in terms of the number of RTL templates parsed. VAX has just 125 RTL templates and 64 patterns. In comparison, i386 has 2238 templates and 547 patterns. When comparing such a large machine with a small machine, the number of common patterns, there by, the number of templates that match them will be reduced significantly. We can see

(n1+n2)	Arm	I386	Mips	Sparc	Vax
Arm		(2281/3819) =59.72%	(1486/2317) =64.13%	(1475/2282) =64.63%	(799/1706) =46.83%
I386	(2281/3819) =59.72%		(1584/2974) =53.26%	(1441/2939) =49.03%	(719/2363) =30.42%
Mips	(1486/2317) =64.13%	(1584/2974) =53.26%		(838/1437) =58.31%	(355/861) =41.23%
Sparc	(1475/2282) =64.63%	(1441/2939) =49.03%	(838/1437) =58.31%		(410/826) =49.63%
Vax	(799/1706) =46.83%	(719/2363) =30.42%	(355/861) =41.23%	(410/826) =49.63%	

Table 4.3: Actual Templates Matched Based on Common Patterns

that i386 and VAX share a mere 34 patterns in common, which is the reason for only 30% of the templates to share common patterns. But if we look at machines of relatively similar size(in terms of RTL templates count), the pairs MIPS-SPARC and i386-ARM have about 60% of the total templates that can be instantiated from their common patterns.

4.2.1 Code_Iterator Equivalence

If two code iterators are found to be equivalent between two pattern files being compared, their names were made the same. For example

i386 has

(define_code_iterator any_extend [zero_extend sign_extend]) and

ARM has

(define_code_iterator SE [zero_extend sign_extend])

both these are made equal. While comparing, a find and replace all is done for SE to any_extend. Similarly, other equivalent code_iterators are handled.

The above tables, list results based on strict code-iterator equivalence. For example,

(define_code_iterator vqhs_ops [smax smin plus]) in arm.mi and

(define_code_iterator smaxmin [smin smax]) in i386.mi

are considered different, i.e., not equivalent, so rtl-patterns that have these code-iterators are considered as patterns that are not common even though the difference in very minor(a plus)in this case. So that the figures above gives the minimum number of patterns that are same, i.e., a safe figure.

4.3 Implications

From the above tables, we observe that machines of similar size have common RTL patterns which can instantiate at the least (due to strict code-iterator equivalence) around 60% of RTL templates. So based on this observation, we can collect a basis set of common RTL patterns from known machines and manage them in logical groups, like the set of patterns for arithmetic operations, function calls, conditional branches ,etc. This basis set can act as a starting point for writing new machine description for machines of similar class and functionality. The tool developed, supports printing the patterns in 'dot' format(graphviz), which will help in easily visualizing the patterns. Hence, classifying the patterns and filling them with machine-specific operand will be easier.

From table 4.1, we observed that there is a significant amount of redundancy within MD files of the same machine, with the number of unique RTL patterns being usually in the range of one fourth of the RTL templates. So, even while writing new MD files, these redundant patterns can be factored out and listed once, and these patterns can be instantiated with parameters. Since a converter is available that converts this format to existing MD file format, this is non-disruptive and new MD files can be written in a concise way.

Chapter 5

Related Work

Research has been carried out for many years to improve the process of retargeting GCC. Some of the topics of interest are, coming up with a methodology that can systematically build MD files, improving the way machine descriptions are written, and automating the process of writing machine description files.

Khedker et al., proposed a new language specRTL[9] to eliminate the redundancies found in GCC's machine description files. Our work is largely inspired by this work.

- specRTL is a simplified language to allow users to write MD files easily with fewer redundancies.
- specRTL is not suitable to discover redundancies in existing MD files.

Our tool on the other hand is intended to be used by users to discover and minimize redundancies in MD files which in turn can be helpful in retargeting.

Sameera et al.[7] proposed a systematic way to build GCC machine descriptions. They have split the process of building machine descriptions into five steps. As the first step, handling simple assignment statements. Further steps successively handle arithmetic operations, function calls, conditional control transfers and other data type handling respectively in that order. The motivation for this work is to develop a working compiler at every step that can handle a restricted subset of the input language. This subset grows with successive steps.

Kai-Wei Lin et al.[10] work discusses about a systematic methodology, to port GCC to a new architecture. This work is very similar in nature to the work by Sameera et al. [7]. They have split the process of writing Machine Description files into seven steps. The first step involves modifying the configuration files of the target machine. Further steps involve adding functionalities to handle integer assignment, function call support, conditional branches and handling other data types. The final step involves adding optional optimization routines. Based on this methodology, they have built a tool that assists in porting. They have implemented a web based tool, that acts as a wizard and guides the user in each of the steps by providing templates relevant for the current step.

Both the above cited works suggested a systematic way to build MD files. But no attempt was made to simplify it by using existing MD files. Our work, unlike the above works, gives the flexibility to choose the architectures that we feel that the new architecture is similar to. Then we can use the tool developed to extract the patterns from them. So, the minimal set provided is not fixed and it can be changed by choosing the machines of interest and restrict it only to them, thereby the minimal set allows the user to focus only on those patterns that matters and not a universal set.This helps in making the decision process simpler.

The final goal of our work is to generate MD files using MD files of existing MD files. This thesis is the first step in our long term goal. Here we have studied existing MD files and developed tools to understand similarities between architectures. In particular, our work allows reuse of patterns from existing MD files in new MD files. The user is benefited, as patterns are shared between machines, by modifying few patterns, new MD files can be constructed.

In summary, our work is a starting point to convert existing MD files to a specRTL like specification. This specification can be modified by the user to generate MD files for new architecture. We expect the size of new MD files written using this specification to be much smaller and therefore easier to modify these files than the current MD files with full specification.

C.Colberg's [5] work makes understanding the architectural features of a new architecture easier by using the native C compiler. A prototype tool, Architecture Discovery Tool(ADT) was created, which generates a formal machine description, which can be used by a back end generator to get a native code generator. The ADT, runs simple C or FORTRAN programs on the target machine, and discovers instruction set, addressing modes, calling conventions, etc. The method used by ADT to do this is called Self-Retargeting Code Generation. One important constraint this work places is that the target architecture should have a native compiler already available, which is not always possible.

Chapter 6

Conclusion and Future Work

Retargeting compilers to a new architecture is a challenging job. But we can make this process simpler by using information from existing architectures. Through empirical studies we were able to show the similarities between architectures which is in line with Davidson and Fraser's observation[6].

As part of our work, we have implemented a parser to read MD files. We chose to concentrate on RTL templates in MD files, which are used to build expander, describe instructions, instruction scheduling, peephole optimization, etc. We devised a method to extract RTL patterns from the RTL templates. With this framework in hand, we measured similarities between machine description files based on their RTL patterns and showed that similarities that exist are promising. We also proposed that the RTL patterns which are common between machines can be grouped and we can come up with a minimal set of RTL patterns which can be used in building machine description files for new machines of similar class. These RTL patterns can be filled with the new machine's parameters to generate parts of the machine description file for the new machine. As part of that, we showed how to instantiate a RTL pattern with parameters to get the RTL template. We also found that this technique can be used to remove redundancies within machine description files of a particular machine by listing RTL patterns separately and instantiating them with parameters.

One of the immediate future works is to create a Graphical User Interface(GUI) that can help the user to visualize RTL patterns and fill it. The long term goal of this work is to build a tool which can automatically generate parts of MD files from existing MD files with user assistance. While it seems impossible to generate a complete MD file automatically, even a partially generated MD file will help the user in retargeting.

Appendix A

MDParser User's Manual

MDParser is the tool that implements the techniques to extract RTL patterns from MD files as explained in our thesis. This section describes the system requirements and some examples to show the usage of the tool.

A.1 System Requirements

The tool has been successfully tested under the following conditions. The tool is expected to work in any compatible systems.

- *Processor:* Intel Core i3.
- *OS:* Ubuntu 10.04 LTS.
- *Compiler:* gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5.1), used for building the tool.
- *MD Files Used:* MD files are taken from the back-end of GCC version 4.6.1.
- *Compiler Tools:* flex 2.5.35, bison (GNU Bison) 2.4.1., GNU Make 3.81.

A.2 Installation

A make file is provided along with the source of MDParser. The tool can be built using this make file by giving the following command at the root of the source tree, make

The tool will be built and stored as a binary executable file bin/md_parser. The compiled binary can be moved to any directory.

A.3 Options

The following are the options supported by the tool.

1. parse
2. output
3. graph
4. include
5. param
6. expand
7. split
8. dir
9. intersect

A.3.1 parse

This option parses the MD files and generate patterns.

The argument is a directory name. For example, to parse mips.md, mips.md should be in directory mips. As argument, give the full path to mips without any trailing '/' character.

Example :

```
bin/md_parser -parse test/mips
```

the above command starts parsing mips.md in ./test/mips/mips.md

Use the option output along with it to specify the file to which it has to be output

bin/md_parse -parse test/mips -output mips.mi

will write output to file mips.mi. If output is not set, default option is to print it to stdout.

A.4 Patterns in dot Format

We added the functionality to print each of the patterns in the pattern list in the dot format, which can be used by the tool graphviz[8]. This tool helps in visually representing graphs. By writing patterns this way, the output file can be compiled using the dot tool to get a graphical representation of each of the patterns in the patterns list as seen in the examples above. This helps understanding the patterns easier.

Use the option graph to print the patterns in dot format.

bin/md_parser - parse test/mips -output mips.mi -graph mips.dot

The above command will list the patterns in dot notation in the file mips.dot apart from writing patterns to mips.mi. The mips.dot file can be further compiled using the dot tool to get the graphical representation of the patterns, with each pattern listed in one page.

A.4.1 intersect

This option, given two pattern files, lists the patterns that are common to both the files.

- This option expects two mi files which are list of patterns generated by parse command separated by commas(no spaces).

- Copy the code_iterators listed in first file to second file and remove code iterators from first file.

- If two code_iterators are found to be equivalent, but have different names, use "find and replace" to make both the code iterators have the same name and the same name is used in patterns.

A sample usage scenario is,

bin/md_parse -intersect mips.mi,arm.mi -output mips-arm.mi

A.4.2 split

This option splits an MD file into two files, one containing pattern index in place of RTL templates and another containing the corresponding parameters for the template. This option can also be made to output a single file containing the pattern index and its parameter together.

Example : bin/md_parse -split test/mips -include mips.mi -dir mips-output

- split expects a directory as in the case of parse command as an argument. This command will start parsing from test/mips/mips.md.
- -include tells the pattern file(mips.mi) to look up. This has to be generated by the parse command.
- -dir command tells the directory in which the output files will be generated. Say, for a file mips.md, mips.com and mips.par will be generated. mips.com contains instructions in compressed form, templates will be given by index of pattern in mi file. mips.par will contain parameters.

A.4.3 expand

This option, combines the pattern index and parameters and produces the RTL templates back to get the MD files in the format supported by GCC. This option was tested by generating first splitting MD files to patterns and parameters using the split command and then regenerating the MD files back.

bin/md_parse -expand mips-output/mips.com -include mips.mi -param mips-output/mips.par -dir mips-output1

- expand option expects the base .com file which *include*s all other .com files. for mips machine it will be mips.com.
- include must be used with -expand as is the case with split option.
- param tells the base file which *include*s parameters for all .com files.
- dir tells the directory in which the output files has to be written.

All corresponding com and par files will be merged to get the .md files back. Example, mips.com and mips.par will be merged to get mips.md in mips-output1 directory.

Appendix B

Additional Results

Let us say, there are two machines m1 and m2 whose individual patterns lists are available with us. We did an intersection of these patterns and got the common patterns list. We calculated the number of templates that can be instantiated in each of these machines, using this common patterns list as the source. This is listed in table B.1. This gives the number of templates that can be instantiated in one machine using another machine as the source.

Source→ Destination↓	Arm	I386	Mips	Sparc	Vax
Arm		(1196/2238) =53.44%	(504/736) =68.48%	(483/701) =68.91%	(88/125) =70.4%
I386	(1085/1581) =68.63%		(509/736) =69.16%	(447/701) =63.76%	(87/125) =69.6%
Mips	(982/1581) =62.11%	(1075/2238) =48.03%		(391/701) =55.77%	(80/125) =64%
Sparc	(992/1581) =62.75%	(994/2238) =44.41%	(447/736) =60.73%		(74/125) =59.2%
Vax	(711/1581) =44.97%	(632/2238) =28.23%	(275/736) =37.36%	(336/701) =47.93%	

Table B.1: Templates Matched on Target Machine Based on Common Patterns

It can be seen from the above table that, when using Vax as the source, the percentage of templates that can be filled in a target is significantly reduced. We believe that the reason for this anomaly is the small size of Vax.

We can repeat the above experiment by taking patterns from two or more machines as the source machines and try generating templates for a new machine. The number of possibilities are overwhelming. So such possibilities were not tried owing to the lack of time.

Bibliography

[1] Gcc wiki. `http://gcc.gnu.org/wiki`.

[2] Gnu compiler collection (gcc) internals. `http://gcc.gnu.org/onlinedocs/gccint/`.

[3] Essential abstractions in gcc, 2011. `http://www.cse.iitb.ac.in/grc/gcc-workshop-11/`.

[4] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[5] Christian S. Collberg. Automatic derivation of machine descriptions. *ACM Transactions on Programming Languages and Systems*, 24(4):369–408, July 2002.

[6] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[7] Sameera Deshpande and Uday P. Khedker. Incremental machine descriptions for gcc. In *GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems*, 2007.

[8] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing Graphs with* dot.

[9] Uday P. Khedker and Ankita Mathur. specrtl: A language for gcc machine descriptions. `http://www.cse.iitb.ac.in/grc/index.php?page=specRTL`.

[10] K.-W. Lin and P.-S. (2012) Chen. An assistance tool employing a systematic methodology for gcc retargeting. *Journal of Software: Practice and Experience*, 42(1):19–36.

[11] Richard M. Stallman and the GCC Developer Community. Using the gnu compiler collection, (for gcc version 4.3.6).