

specRTL: A Language for GCC Machine Descriptions

Uday P. Khedker, Ankita Mathur

GCC Resource Center*,
Dept. of Computer Science and Engg., IIT Bombay, India
{uday,ankitam}@cse.iitb.ac.in

Abstract. The mechanism of GCC machine descriptions has been quite successful as demonstrated by a wide variety of targets for which GCC ports exist. However, this mechanism is quite ad hoc and the machine descriptions are difficult to read, construct, maintain, and enhance because of the verbosity, repetitiveness, and the amount of details. We propose a language called specRTL which provides a compositional specification mechanism for defining patterns that describe RTL templates in machine descriptions. These patterns can be refined by extending them and by associating concrete details with them in a need based manner. Machine descriptions written using specRTL are smaller and simpler and hence easy to read, construct, and maintain. specRTL integrates with conventional machine descriptions seamlessly. Since specRTL compiler generates the conventional machine descriptions, GCC source need not change. This enables external, incremental and non-disruptive migration of the existing machine descriptions to specRTL and construction of new machine descriptions thereby paving way for a smooth transition to better code generation in GCC.

1 Introduction

The Gnu Compiler Collection uses a retargetable compilation model which is adapted to a given target by reading a description of the target and instantiating the machine dependent parts of the generated compiler. This mechanism has been quite successful as demonstrated by a wide variety of targets for which a GCC port exists. However, this mechanism is quite ad hoc and the machine descriptions are difficult to read, construct, maintain, and enhance. They require specifying instruction patterns using Register Transfer Language (RTL) templates using a mechanism which is verbose, repetitive, non-composable, and requires a lot of details.

A typical RTL template looks as follows. Here the name of the instruction is `addsi3`.

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (match_operand:SI 1 "register_operand" "r")
                 (match_operand:SI 2 "register_operand" "r")))]
  "" /* C boolean expression, if required */
  "add \t%0 %1 %2"
  )
```

* <http://www.cse.iitb.ac.in/grc>. Funded by the Dept. of Information Technology, Gov. of India, as a part of the National Resource Center for Free and Open Source Software (NRCFOSS).

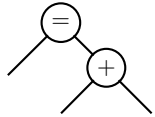
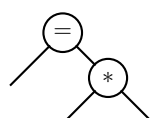
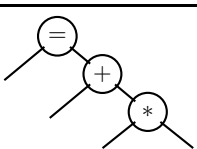
Addition (Most templates reproduced below appear in define_insn)						Structure
<pre>[(set (match_operand:m 0 "register_operand" "c0") (plus:m (match_operand:m 1 "register_operand" "c1") (match_operand:m 2 "p" "c2")))]</pre>						
Pattern name	<i>m</i>	<i>p</i>	<i>c0</i>	<i>c1</i>	<i>c2</i>	
add<mode>3	ANYF	register_operand	=f	f	f	
define_expand add<mode>3	GPR	arith_operand				
*add<mode>3	GPR	arith_operand	=d,d	d,d	d,Q	
*add<mode>3_mips16	GPR	arith_operand	=ks,d,d,d,d	ks,ks,0,d,d	Q,Q,Q,0,d	
Multiplication (Most templates reproduced below appear in define_insn)						
<pre>[(set (match_operand:m 0 "register_operand" "c0") (mult:m (match_operand:m 1 "register_operand" "c1") (match_operand:m 2 "register_operand" "c2")))]</pre>						
Structure	Pattern name	<i>m</i>	<i>c0</i>	<i>c1</i>	<i>c2</i>	
	*mul<mode>3	SCALARF	=f	f	f	
	*mul<mode>3_r4300	SCALARF	=f	f	f	
	mulv2sf3	V2SF	=f	f	f	
	define_expand mul<mode>3	GPR				
	mul<mode>3_mul3_loongson	GPR	=d	d	d	
	mul<mode>3_mul3	GPR	d,1	d,d	d,d	
Multiply and accumulate (All templates reproduced below appear in define_insn)						
<pre>[(set (match_operand:m 0 "register_operand" "c0") (plus:m (mult:m (match_operand:m 1 "register_operand" "c1") (match_operand:m 2 "register_operand" "c2")) (match_operand:m 3 "register_operand" "c3")))]</pre>						
Structure	Pattern name	<i>m</i>	<i>c0</i>	<i>c1</i>	<i>c2</i>	<i>c3</i>
	*mul_acc_si	SI	=1*?*?,d?	d,d	d,d	0,d
	mul_acc_si_r3900	SI	=1?*?,d*?,d?	d,d,d	d,d,d	0,1,d
	*macc	SI	=1,d	d,d	d,d	0,1
	*madd4<mode>	ANYF	=f	f	f	f
	*madd3<mode>	ANYF	=f	f	f	0

Fig. 1. RTL templates in `mips.md` parameterized by modes (*m*), predicates (*p*) and constraints (*c0*, *c1*, *c2*, *c3*) to highlight the verbosity, repetitiveness, and non-composability.

The inner box contains the RTL template of the computation required for addition and the last line in the outer box contains its assembly format. The RTL template uses RTL operators `set` and `plus`; the former represents an assignment. Operator `match_operand` matches an operand using a mode (SI for single integer), a predicate (`register_operand`), and constraint strings ("`=r`", "`r`" and "`r`"). Given a GIMPLE statement `a = b + c`, first an RTL statement is generated and then the assembly statement is generated eventually.

Figure 1 lists some RTL templates that appear in the file `mips.md` which is a part of the MIPS machine descriptions. We have parameterized the templates with mode, predicates, and constraints to highlight the fact that several RTL templates share the

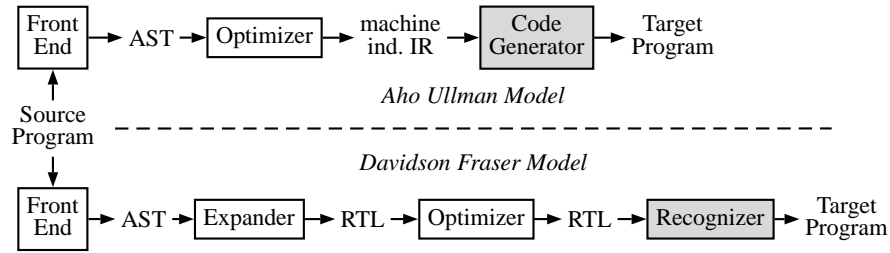


Fig. 2. Classical Compilation Models

same structure and differences in them correspond to different values of target specific attributes. GCC machine descriptions do not have any construct to create such structures and instantiate them by specifying attribute values as needed. Besides, the RTL template for multiply and accumulate instruction can be viewed as a composite structure that combines the structures for add and multiply instructions. However, the specification mechanism does not allow creation and composition of such structures.

We propose a language called specRTL which provides a compositional specification mechanism for defining patterns that describe RTL templates by providing a clean separation between the shapes of the templates and the target specific details. This is achieved by supporting creation of *abstract patterns*. They can be composed and instantiated with concrete details to create *concrete patterns* as required. For this purpose, specRTL provides well-defined simple refinement operators called *extends*, *instantiates*, and *overrides*. Machine descriptions written using specRTL are smaller and simpler and hence easy to read, construct, and maintain. Further, specRTL integrates with conventional machine descriptions seamlessly. Since specRTL compiler generates the conventional machine descriptions, there is no need to change the GCC source. This enables external, incremental and non-disruptive migration of the existing machine descriptions to specRTL and easier construction of new machine descriptions thereby enabling a smooth transition to a better code generation strategy in GCC.

The rest of the paper is organized as follows: Section 2 reviews the retargetability model of GCC. Section 3 presents empirical measurements of redundancy in machine descriptions. specRTL is presented in Section 4. Section 5 discusses the advantages of using specRTL for GCC. A brief description of the related work is presented in Section 6. Section 7 concludes the paper.

2 The Retargetability Model of GCC

GCC uses a modified version of the Davidson Fraser model of compilation [5]. Figure 2 contrasts this model with the traditional Aho Ullman model [2] which performs instruction selection over optimized machine independent intermediate representation (IR). In order to ensure the quality of generated code, instruction selection in Aho Ullman model is performed using cost based tree tiling [1,15] that tries to cover a *Subject Tree* in the IR with instructions that minimize the cost using a set of *Transformer Trees*.

The Davidson Fraser model advocates simple instruction selection and optimizes the selected instructions. An *expander* generates a naive machine dependent code using

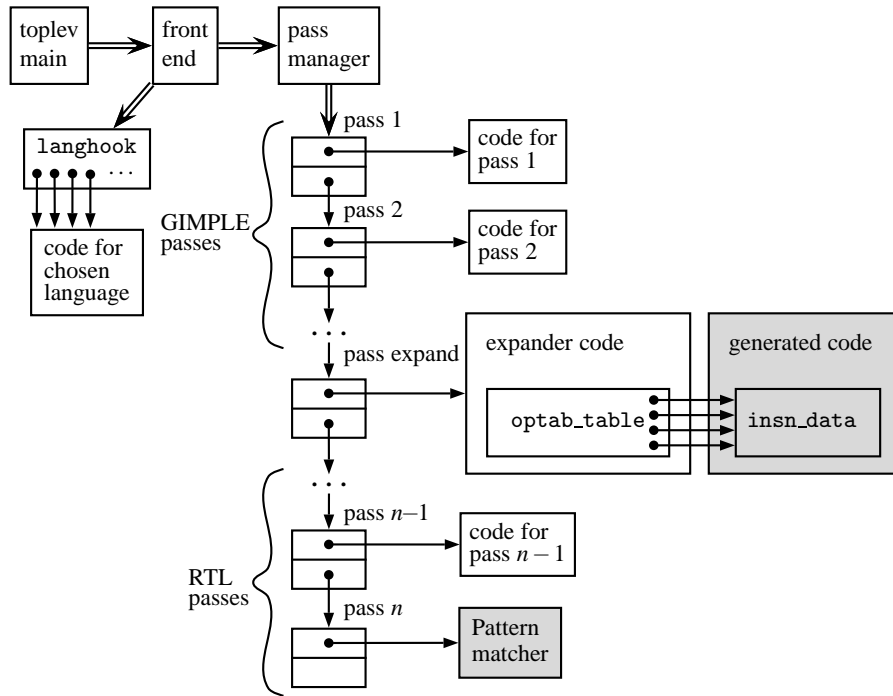


Fig. 3. GCC's adaptation of the Davidson Fraser Model. Gray boxes represent target dependent code. Double arrows represent control flow; single arrows represent pointers.

transformer trees (most often RTL trees) by employing simpler structure based tiling [4]. The final code is produced by a *recognizer* that identifies the instructions (*Inst*) corresponding to the register transfers representing the intermediate code. Retargeting a compiler in Davidson Fraser model requires rewriting the expander and the recognizer which employ simple algorithms. A generic optimizer for machine dependent code is possible because of the following key idea: *When computations are expressed in the form of allowable register transfers, although the actual register transfer statements are machine dependent, their form is machine independent.*

Figure 3 illustrates GCC's adaptation of Davidson Fraser model. Contemporary versions of GCC employ many optimizations at the machine independent level on GIMPLE representation which is a three address code and the expander generates RTL code from GIMPLE and not abstract syntax trees. More importantly, since the form of register transfers is machine independent, GCC isolates the machine specific information in carefully defined data structures. Thus a compiler for a new target can be constructed using generator programs that read machine descriptions and instantiate these data structures; the expander need not be rewritten manually. The recognizer (called a *pattern matcher* in GCC) uses a finite automaton and is generated from machine descriptions. Figure 4 compares GCC with two frameworks that use the Davidson Fraser model: Zephyr which uses VPO [3] for code generation and Quick C-- [6] which is a prototype that generates code comparable with production quality compilers.

		GCC	Zephyr/VPO	Quick C--
Expander	Transformation Trees (TT)	RTL templates	RTL templates	Expansion tiles
	Nature of TT	Target dependent	Target dependent	Target independent
	Fixing shapes of TT	MD writing	MD writing	Framework design
Recognizer	$TT \rightarrow Inst$ method	Pattern matching using finite automaton	LR parsing (Yacc based)	Pattern matching
	$TT \rightarrow Inst$ mapping	Fixed manually	Discovered automatically	Discovered automatically
	Time of devising $TT \rightarrow Inst$ mapping	MD writing	Compilation	Compiler construction

Fig. 4. Comparing some code generators in Davidson Fraser model. $TT \rightarrow Inst$ denotes the translations performed by recognizer.

3 Measuring Redundancy in RTL Templates

The motivating example in Figure 1 illustrates the existence of two kinds of redundancies in machine descriptions:

1. If we treat the modes, predicates, and constraints as attributes of the nodes, then a large number of instructions have identical structures with differences restricted to the attributes of the nodes.
2. Many structures have common sub-structures of the following two kinds:
 - (a) There may be an overlap in two structures. For example, the structure of `macc` instruction has an overlap with the structures for `add` and `mul`.
 - (b) A structure may appear as a substructure in some other structure.

Redundancies of the kind (1) and (2b) were measured as follows [16]. An `.md` file parser sorts the RTL templates in an `.md` file by the height of their trees where the height of a tree is the length of the longest path from the root to a leaf node. Then the trees with height i are compared with all trees of height i to discover the instances of redundancy (1) and all trees of height $i - j$, $1 \leq j \leq i$ to discover the instances of redundancy (2b). We call a tree as a primitive trees if it cannot be expressed as a composition of other trees appearing in the `.md` file. The table below summarizes our measurements.

MD File	Total number of patterns	Number of primitive trees	Number of times primitive trees are used to create composite trees
<code>i386.md</code>	1303	349	4308
<code>arm.md</code>	534	232	1369
<code>mips.md</code>	337	147	921

It is clear that RTL templates have a high amount of redundancy. The MIPS and ARM machine descriptions have less redundancy compared to `i386`. This is because they are RISC architectures and `i386` is a CISC architecture. Discovering redundancy (2a) automatically is a much harder problem and was not attempted. However, it is easy to visualize the presence of such redundancy as seen in the case of `macc` instruction.

Clearly there is a need of a better specification mechanism for machine descriptions to make them simpler and more understandable.

4 specRTL: A Language for Specifying RTL Templates

We introduce specRTL by creating the patterns for describing some of the RTL templates in our motivating example. A complete grammar of specRTL and other resources are available at <http://www.cse.iitb.ac.in/grc/index.php?page=specRTL>.

4.1 An Overview of specRTL

Based on the observations in our motivating example (Figure 1), we view the following key ideas as specRTL requirements.

1. It should be possible to create abstract structures of RTL templates that can be refined later to create new RTL templates.
2. Following three kinds of refinements should be possible:
 - (a) Composing abstract structures to create new abstract structures.
 - (b) Instantiating abstract structures with concrete details.
 - (c) Changing concrete details without changing the structure.

specRTL meets the above requirements by providing a mechanism to create *abstract patterns* which can be refined into *concrete patterns*. Each pattern represents a particular computation and can be viewed as an operation with a fixed number of operands (except for patterns involving the RTL operator `parallel` or a sequence of RTL templates). Recall the key idea from the Davidson Fraser model: Although register transfers are machine dependent, their form is not. Abstract patterns represent the form and the concrete patterns represent their machine dependent instances.

The structure of a pattern is represented by a tree in which each internal node is labeled with an RTL operator. We allow the leaf nodes to remain unspecified; such leaf nodes are called abstract nodes. A concrete leaf node could be a GIMPLE operand to be matched, a fixed register, a constant value, replication of another leaf node, or a number. What separates an internal node from a leaf node is that an internal node has an RTL expression as its operand and can have only a *mode* as its attribute. An abstract pattern must have at least one abstract node and its arity is defined by the number of abstract nodes. A concrete pattern cannot have any abstract node. By definition, each RTL operator is an abstract pattern with known fixed arity.

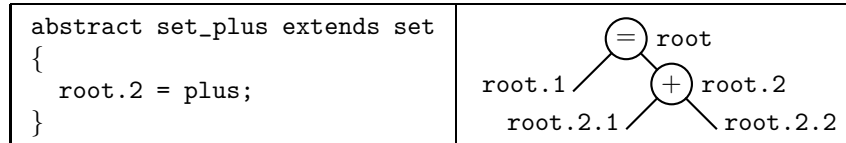
The pattern which is refined is called a *base pattern*. The resulting pattern is called a *derived pattern*. Refinement of patterns is supported by the following operations: *extends*, *instantiates*, and *overrides*. These operations directly correspond to the requirements (2a), (2b), and (2c) above. The pattern used by *extends* to replace a leaf node in the base pattern is called an *extender pattern*; this must be an abstract pattern.¹ The properties of these operations are described below.

Operation	Base pattern	Derived pattern	Nodes influenced	Can change
<i>extends</i>	Abstract	Abstract	Leaf nodes	Structure
<i>instantiates</i>	Abstract	Concrete	All nodes	Attributes
<i>overrides</i>	Abstract	Abstract	Internal nodes	Attributes
	Concrete	Concrete	All nodes	Attributes

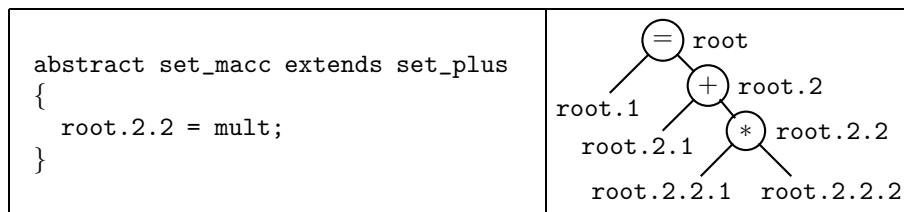
¹ It is tempting to view our refinement as inheritance in object oriented languages. However, specRTL does not have class-object or function-data dichotomy.

4.2 Creating Abstract Patterns in specRTL

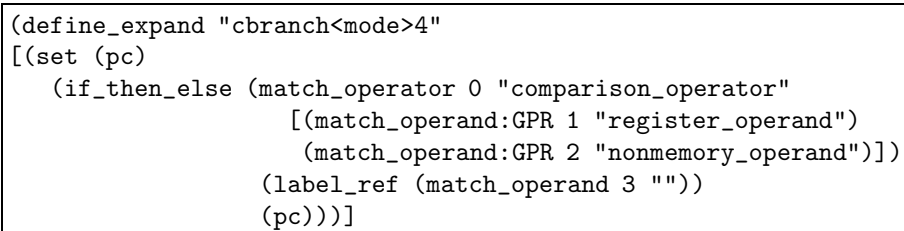
First we define an abstract pattern called `set_plus` by extending the abstract pattern `set` using another abstract pattern `plus`. In this case `set` is the base pattern, `set_plus` is the derived pattern, and `plus` is the extender pattern.



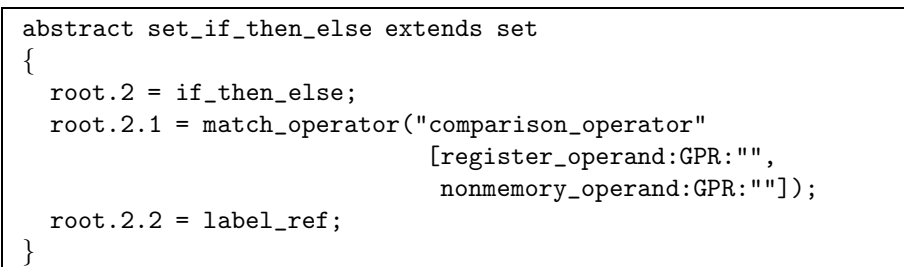
Effectively, this specification makes a copy of `set` and replaces its second operand (`root.2`) by `plus`. Hence the root node of `set_plus` is the RTL operator `set`. The arity of `set_plus` is 3 because it has three unspecified (and hence abstract) leaf nodes. Now we create an abstract pattern called `set_macc` to represent multiply and accumulate operations by plugging in the RTL operator `mult` as the third operand of `set_plus`. Note that the arity of `set_macc` is 4.



We use the specification of `cbranch<mode>4` in `mips.md` in order to show the rich set of possibilities in specRTL:



We create the structure of the instruction using the abstract pattern `set_if_then_else`. The internal nodes of the pattern are RTL operators `if_then_else`, `match_operator`, and `label_ref`.



Since we have specified the concrete details of the operands of `match_operator`, this pattern has two concrete leaf nodes: `root.2.1.1` and `root.2.1.2`. It has three abstract leaf nodes: `root.1` (the LHS of `set`), `root.2.2.1` (the operand of `label_ref`), and finally `root.2.3` (the third operand of `if_then_else`). Needless to say, a different choice of concrete and leaf nodes is also possible in this case.

4.3 Creating Concrete Patterns in specRTL

Concrete patterns are created by two operations: `instantiates` (which defines attributes in terms of concrete details) and `overrides` (which changes the attributes). An RTL operator cannot be changed directly; it requires extending a base pattern suitably.

Specification of concrete patterns has the following syntax:

```
concrete spec_header { in_specs } { : other_stuff : } { out_specs }
```

in_specs and *out_specs* describe the *input* and *output* RTL templates. The *other_stuff* could be boolean conditions, assembly output formats, C code etc. required in the conventional machine descriptions. Delimiters `{:` and `:}` are used to simplify parsing. When *other_stuff* or *out_specs* are not required, we also omit the delimiters. For simplicity we ignore *out_specs* and *other_stuff* in this paper.

We now instantiate the base pattern `set_plus` to specify the concrete patterns for `define_insn "add<mode>3"` and `define_expand "add<mode>3"`.

```
concrete add<mode>3.insn instantiates set_plus
{ set_plus(register_operand:ANYF:"=f", register_operand:ANYF:"f",
  register_operand:ANYF:"f");
  root.2.mode = ANYF;
}
concrete add<mode>3.expand instantiates set_plus
{ set_plus(register_operand:GPR:"", register_operand:GPR:"",
  arith_operand:GPR:"");
  root.2.mode = GPR;
}
```

The suffixes `insn` and `expand` attached to the name `add<mode>3` specify whether the RTL template is to be generated for `define_insn` or `define_expand`. Additional suffixes such as `peephole2`, `attr` etc. can be used for each kind of “define_” supported in the conventional machine descriptions. The mode iterator `<mode>` is carried over unchanged from the conventional machine descriptions. Attributes of the operands of the base pattern `set_plus` are specified by supplying three arguments with the syntax *predicate:mode:constraints*. There are no constraints in our template for `define_expand "add<mode>3"` and hence empty strings `"` are specified.

Concrete details of leaf nodes are specified using the following syntax. The names that appear for *register_specifier* and *constant_specifier* and *register_name* are reserved words in specRTL.

Type of leaf	Specification Syntax
GIMPLE operand	<i>predicate : mode : constraints</i>
Scratch operand	<i>mode : constraints</i>
Fixed register	<i>register_specifier (mode : register_name) or register_name</i>
Constant Value	<i>constant_specifier : value : mode (mode is optional)</i>
Copy of another leaf	<i>duplicate leaf_number</i>
Number	<i>number</i>

The mode of an internal node can be assigned by describing the path of the node from the root. The mode of a leaf node can also be assigned in a similar manner. However, assigning it as an argument of the base pattern name is concise and simpler.

The RTL templates of the remaining two patterns for addition differ from that in `add<mode>3.expand` only in terms of constraints. Hence it is more convenient to override it and merely change the constraints as shown below.

```
concrete *add<mode>3.insn overrides add<mode>3.expand
{ allconstraints = ("=d,d", "d,d", "d,Q"); }
```

We create the first multiplication pattern in order to show two additional useful features.

```
concrete *mul<mode>3.insn instantiates set_mult
{ set_mult(register_operand:SCALARF:"f",
  register_operand:SCALARF:"f", register_operand:SCALARF:"f");
  root.2.mode = SCALARF;
}
```

The RTL template of `*mul<mode>3_r4300` is identical to that of `*mul<mode>3` whereas `mulv2sf3` differs from `*mul<mode>3` only in that the mode SCALARF is replaced by V2SF for each node. These can be specified as shown below.

```
concrete *mul<mode>3_r4300.insn overrides *mul<mode>3.insn
{}
concrete mulv2sf3 overrides *mul<mode>3.insn
{ SCALARF -> V2SF; }
```

Now we instantiate the abstract pattern `set_if_then_else` to create the concrete pattern `cbranch<mode>4.expand`. By our design, this requires supplying the concrete details of the LHS of `set`, the GIMPLE operand of `label_ref`, and the third operand of `if_then_else` (which is the destination if the condition is false).

```
concrete cbranch<mode>4.expand instantiates set_if_then_else
{ set_if_then_else(pc, null:NULL:"", pc); }
```

Here `pc` is a fixed register. GIMPLE operand to be matched for `label_ref` has no mode, predicate, or constraints.

If there is no need to override a concrete pattern name, then uniqueness of names among the set of one particular kind of `define_` is not essential. For creating unnamed patterns the name can be dropped and the header could begin with “`concrete .insn`”.

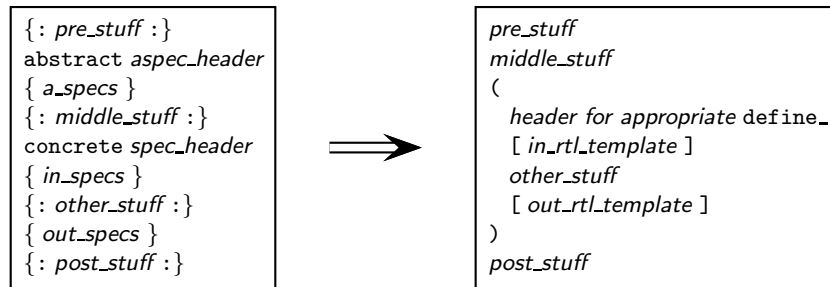


Fig. 5. Translation performed by a specRTL compiler. *pre_stuff*, *middle_stuff*, *other_stuff*, and *post_stuff* refer to the other text that appears in the conventional machine descriptions. All of them (and *out_specs*) are optional along with their delimiters.

5 Advantages of Using specRTL in GCC

Since abstract patterns capture the structure or form of computation by hiding the details that vary, a large number of abstract patterns are common to most machine descriptions. Thus it is possible to create a basis set of abstract patterns that can be shared by most machine descriptions. This is expected to reduce the size of machine descriptions significantly and would simplify the task of writing machine descriptions.

Further, specRTL integrates seamlessly with conventional machine descriptions. Figure 5 illustrates the translation performed by a specRTL compiler. The abstract specifications are read and relevant information is stored but the output corresponds to the concrete specifications. Everything else is treated as a comment but instead of ignoring, it is copied to the output. This has two pleasant consequences: In order to start using specRTL, neither the GCC source needs to be changed, nor the machine descriptions need to be re-written completely; they can be updated incrementally and each increment can be validated by building GCC.

Thus, not only are specRTL based machine descriptions smaller and simpler, migration to them is an external, incremental, and non-disruptive change in GCC.

6 Related Work

Languages to describe instruction set architectures (ISA) [9,11,12,13,14] have been found useful in hardware software co-design. However, generating a production quality code generator for a processor like x86 requires much more information than just the ISA. GCC machine descriptions are large because they are used to generate the expander, recognizer, as also machine dependent optimizers for optimizations such as instruction scheduling, peephole optimizations etc. Quick C-- [6] uses a combination of SLED [13] and λ -RTL [12] and processes them to generate only the recognizers—other phases are fixed in the framework. Hence its machine descriptions are much smaller.

specRTL is a follow up of an attempt to design a mechanism of factoring out common information in .md files [10]. This was achieved by supporting a new construct called `define_rtltemplate` which creates a named RTL template that is parameterized for variable parts. A named RTL template is then instantiated using the construct

`define_pattern` by supplying parameters. Similarly, the variations in C code are abstracted out using `define_code` that names the code fragments and allows them to be parameterized. These names can then be used with appropriate parameters where required. A parser reads the machine descriptions containing these new constructs (along with the conventional constructs) and generates conventional `.md` files. Machine descriptions for `i386` and `rs6000` were rewritten using these constructs. They were validated by doing a native build for `i386` GCC and running `make check`. The numbers below show the redundancy encountered and reductions in patterns.

	define_rtltemplate		define_code		define_patterns		define_insn		define_expand	
	Defs	Uses	Defs	Uses	Defs	Uses	Old	New	Old	New
<code>i386.md</code>	295	1638	42	150	170	-	622	350	236	159
<code>rs6000.md</code>	94	578	2	4	36	-	577	491	167	134

Using `define_rtltemplate` requires identifying and naming all parameters of a template which is tedious and error-prone. `specRTL` obviates this need by allowing abstract nodes which are left implicit and can be concretized when needed by naming their paths from the root. Further, the mechanism of `define_rtltemplate` is highly contextual and restrictive in that it has to be written with a particular `.md` file in mind. It seems difficult to write general `define_rtltemplate` that are common across a large number of machine descriptions. `specRTL` makes this possible.

Gimple Back End [17] is an ambitious project that plans to eliminate RTL from GCC completely and build a recognizer that accepts GIMPLE. The plan as of now is to use CGEN [8] as the specification mechanism for machine descriptions. We believe that this change is disruptive and non-incremental.

7 Conclusions and Future Work

`specRTL` facilitates smaller, simpler, and more understandable machine descriptions. Abstract patterns provide target independence and hence a common basis set of patterns that can be refined as needed for a target would be of a great help. Since `specRTL` integrates seamlessly with conventional machine descriptions, migration to `specRTL` is an external, incremental, and non-disruptive process with few regressions, if any, in each step. Further, constructing new machine descriptions becomes much easier.

We would like to create a basis set of abstract patterns that is common to a large number of targets. A medium term goal, we would like to explore the possibility of using smaller declarative machine descriptions (eg. λ -RTL or SLED [12,13]) to concretize the basis set for a given target. We believe that this is achievable because of the following reason: Although RTL is target independent, RTL templates are target dependent in GCC because they combine shapes with target details. If we separate them and construct target independent shapes, it may be possible to fill in target details by reading crisp and succinct descriptions of ISA. We would also like to explore combining actions in the spirit of modular attribute grammars [7].

Eventually, we want to improve code generation in GCC. This requires changing machine descriptions and the retargetability mechanism. Changing both of them simultaneously is a big architectural change that is radical and is likely to be disruptive. It may

be preferable to first migrate all existing machine descriptions to a clean and concise form without changing code generation, and then change the code generation strategy.

Acknowledgments

Abhijat Vichare, Sameera Deshpande, Sagar Kamble, Ketaki Tiwatne, and Ashish Mishra have contributed to related GCC explorations. We would also like to thank the referees for valuable suggestions. Ankita is funded by the DIT grant.

References

1. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 207–217, 1975.
2. A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.
3. A. Appel, J. Davidson, and N. Ramsey. The zephyr compiler infrastructure. Technical report, 1998. <http://www.cs.virginia.edu/zephyr/overview98.ps> (Last accessed on 28 Jan 2011).
4. A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
5. J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6:505–526, October 1984.
6. J. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 403–416, 2010.
7. G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Comput. J.*, 33:164–172, April 1990.
8. D. Evans, F. Ch. Eigler, B. Elliston, and D. Brolley. CGEN, the Cpu tools GENerator, 2009. <http://sourceware.org/cgen/> (Last accessed on Jan 28, 2011).
9. A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.
10. S. Kamble. Improved machine description specification in gcc. Master's thesis, Department of Computer Science and Engineering, IIT Bombay, 2010.
11. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor description. *Design Automation for Embedded Systems*, 3:75–108, 1998.
12. N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *In ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES98)*, volume 1474 of LNCS, pages 172–188. Springer Verlag, 1998.
13. N. Ramsey and M. F. Fernández. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19:492–524, May 1997.
14. O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture implementation using the machine description language lisa. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference, ASP-DAC '02*, pages 239–, Washington, DC, USA, 2002. IEEE Computer Society.
15. P. Shankar. Instruction selection using tree parsing. In *The Compiler Design Handbook*, pages 565–602. 2002.
16. K. Tiwatne and A. Mishra. Measuring redundancy in machine descriptions. Internal Document, GCC Resource Center, Dept. of Computer Science and Engg., IIT Bombay, 2010.
17. K. Zadeck. GBE: A gimple back end for GCC, 2009. <http://gcc.gnu.org/wiki/gimplebackend> (Last accessed on Jan 28, 2011).