Writing GCC Machine Descriptions

GCC Version 4.0.2

Abhijat Vichare (amvichare@iitb.ac.in)

Indian Institute of Technology, Bombay

(http://www.iitb.ac.in)

This is edition 1.0 of "Writing GCC Machine Descriptions", last updated on January 7, 2008., and is based on GCC version 4.0.2.

Short Contents

1 Introduction 1
2 Influences on the GCC MD System 3
3 MD System and GCC Architecture 8
4 Definition of Fictitious Target System 11
5 Implementation of MD for TOY 13
6 The New MD and the Build System 14
7 The New MD and the gcc architecture 16
References 17
List of Figures 18
A The CPP macros Implementation 19
B The MD-RTL code Implementation 32
C The auxiliary C code Implementation 38
D Copyright 46

Table of Contents

1 Introduction **1**

2 Influences on the GCC MD System **3**

2.1 Influence of the Source Language 3
2.2 Influence of the Target System 5
2.2.1 Target Systems and Cross Compilation 6
2.3 Influence of GCC Architecture 7

3 MD System and GCC Architecture **8**

3.1 The Organisation of the GCC MD System 8
3.2 Classification of MD Macros 9
3.3 The RTL Based MD Specifications 9

4 Definition of Fictitious Target System **11**

5 Implementation of MD for TOY **13**

5.1 C Preprocessor Macros 13
5.2 Implementing CPP macros 13
5.3 Implementing MD-RTL code 13
5.3.1 A few t_{run} time situations 13
5.3.2 Corresponding MD RTL constructs 13
5.3.3 Examples of MD-RTL code and CPP macros 13
5.4 Implementing auxiliary C code 13

6 The New MD and the Build System **14**

6.1 Making GCC Aware of the New MD 14
6.2 The RTL Object 14
6.3 Conversion of RTL Specifications to C Code 15

7 The New MD and the gcc architecture **16**

7.1 Operation of the “Generated” C Code 16

References **17**

List of Figures **18**

Appendix A The CPP macros Implementation
. . . . **19**

Appendix B The MD-RTL code Implementation **32**

Appendix C The auxiliary C code Implementation **38**

Appendix D Copyright **46**

D.1 GNU Free Documentation License 46

1 Introduction

In [The Conceptual Structure of GCC], page 17 three important time periods were identified as a consequence of the retargetability requirement:

1. the *development* of the CGF, denoted by $t_{develop}$,
2. the *building* of the target specific compiler, denoted by t_{build} , and
3. the *use* of the built compiler to compile an input program, denoted by t_{run} .

The target specific machine description must be created at time instant $t_{develop}$, incorporated into the compiler at time instant t_{build} and used at time instant t_{run} . To understand the issues that go into creating a machine description at $t_{develop}$, we need to know the details of incorporating it into the compiler at t_{build} and it's use at t_{run} . Corresponding to these time durations we have architecture descriptions that help understanding the GCC code base. At $t_{develop}$ we have the GCC Compiler Generation Architecture[1] that describes the view of GCC that a GCC developer works with. At t_{run} we have the GCC Compilation Architecture[2] that describes the concepts behind the implementation (and the running at t_{run}) of the sequence of operations needed to compile a program to target assembly. This is the view of GCC that a user of the compiler sees. The RTL system is the connecting link that converts the MD-RTL based specifications of target properties at $t_{develop}$ to IR-RTL based forms useful at t_{run} . This conversion occurs at t_{build} and is described by the GCC Build System Architecture[3].

The RTL system is used in GCC for both: (a) specifying target properties at $t_{develop}$ using the MD-RTL language, and (b) representing the input being compiled at t_{run} using the IR-RTL language. In both the cases the RTL operators are used to construct expressions, called RTXs, that capture the semantics of the target instruction. At specification time, the RTXs are in an external Lisp like syntax and at representation time they are in an internal form of chained objects. The internal representation is dumped out in external Lisp like syntax. At specification time, the RTXs are a part of MD-RTXs. They use MD constructs of the RTL system for various specification purposes. At representation time, the RTXs are a part of IR-RTXs. They use IR constructs of the RTL system for various representation purposes.

The concepts, mechanisms and operation of the RTL based machine description system in GCC are illustrated with a running example in [The Conceptual Structure of GCC], page 17.[4] We start by examining the target issues that actually influence a retargetable compiler like GCC in Chapter 2 [Influences on the GCC MD System], page 3. Chapter 3 [MD System and GCC Architecture], page 8 examines how these issues are incorporated into the GCC architecture of [The Conceptual Structure of GCC], page 17. We then translate these conceptual views of impact of retargetability into an implementation over a fictitious target in Chapter 5 [Implementation of MD for Toy], page 13. This chapter is the central core of this document. A short summary of the fictitious target properties precedes the implementation description in Chapter 4 [Definition of Fictitious Target System], page 11. At this point the specification of the Toy machine has been developed. In Chapter 6 [The

[1] See section "The GCC Compiler Generation Architecture" in *GCC 4.0.2 – The Conceptual Structure*.
[2] See section "The GCC Compiler Architecture" in *GCC 4.0.2 – The Conceptual Structure*.
[3] See section "The GCC Build System Architecture" in *GCC 4.0.2 – The Conceptual Structure*.
[4] See section "How the RTL Works" in *GCC 4.0.2 – The Conceptual Structure*.

New MD and the Build System], page 14, we use the build system to "transform" our specifications into code that is to be used in a compiler run. Finally, in Chapter 7 [The New MD and the gcc architecture], page 16, we use the compiler architecture to examine a compilation run that produces code for the fictitious target that we have ported GCC to.

The user level details of the GCC build process are provided in [GCC – An Introduction], page 17. The details of the how GCC converts the conceptual structure in [The Conceptual Structure of GCC], page 17 into an implementation are in [The Implementation of GCC], page 17. In this work, we use only the necessary implementation details. The file naming conventions in [The Implementation of GCC], page 17 are used.

2 Influences on the GCC MD System

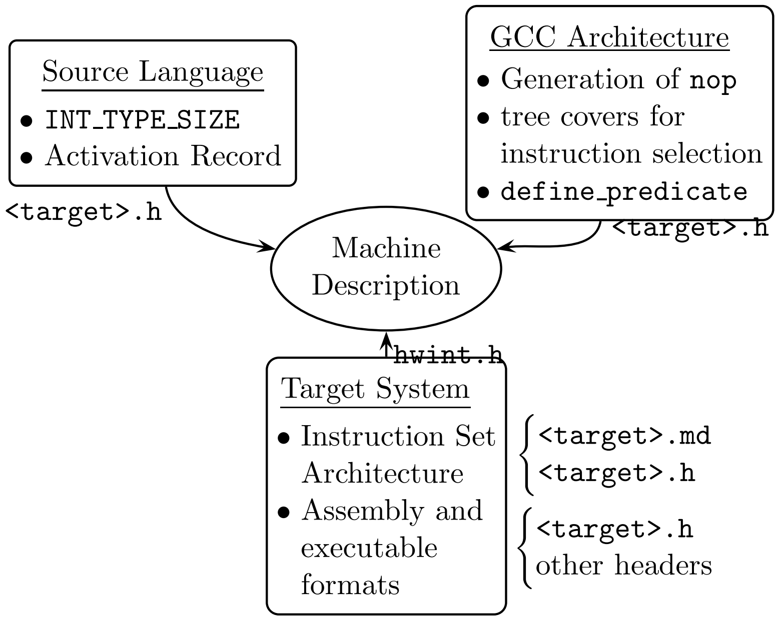

Figure 2.1: Influences on GCC MD

Figure 2.1 summarizes the various influences on the MD system. Various factors of each influence need to be captured by the MD system. We have divided the various influences into the following broad categories:

1. Influence of the source language features.

2. Influence of the target system.

3. Influence of the GCC system itself.

2.1 Influence of the Source Language

Figure 2.2 describes that various aspects of the source language features that impact the MD system. This is natural since the goal of the compiler is to eventually map the source objects to target representation.

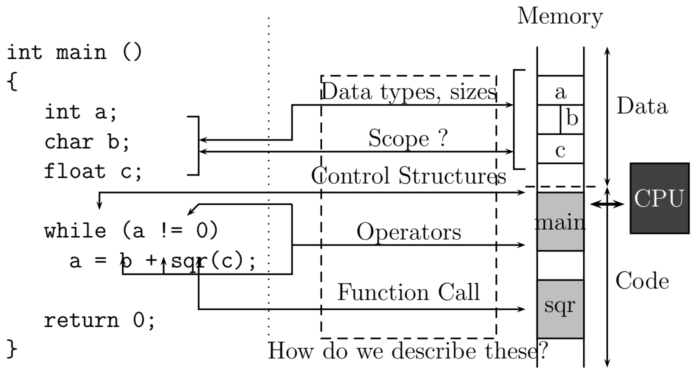

Figure 2.2: Source language influences

The dotted line in the figure divides it into two halves. On the left is some source program whose objects are to be mapped to the target on the right. The target memory is divided into code regions and data regions. Data objects in the source are specified in terms of their data types and names. The data types define the interpretation of the region of memory that will eventually be associated with the named objects of that type. The name of the data object will correspond to some memory location (not shown in the figure). The scope rules of the HLL will define some life time and access constraints on the data object and have to be implemented on the target system. The size of the region of memory that corresponds to a given data type is easy to define in terms of the number of bits. Such parametrization is implemented in terms of simple C preprocessor macros in GCC. In general, these define the sizes in bits of the data types supported by the compiler.

In contrast, for code objects, two main kinds of HLL constructs need to be mapped on to the target code memory: control flow specifications like "`while`" or "`sqr`" and operators like "`!=`", "`+`" etc. For code objects the parametrization is not in terms of simple values. Rather the semantics have to be captured. This is possible using RTL operators, in general. This kind of parametrization cannot use preprocessor macros.

Source language defined control flow constructs like `while` can be expressed as target code if the target program counter can be sequenced and branched. All targets will always have these abilities. In fact, GCC fails to build by default if branch semantics have not been specified in the target MD. User defined control flow constructs like `sqr` in the figure need a more elaborate scheme for expression as target code. This scheme depends on the particulars of the HLL. All the HLLs that GCC supports use a stack of function call activations as the scheme[1] to manage the data and control states needed to prepare for control transfer and return back.

Operators defined by the HLL, like "`!=`", "`+`" etc., are mapped to target instructions if they exist, or implemented using the target instructions that exist. In case some operators may not be mapped to target instructions, GCC provides two alternate paths: a library –

[1] Languages, e.g. Scheme, that need structures other than the stack are not supported by GCC.

libgcc – that attempts to implement the semantics, or have constructs in the MD system that allow expressing the semantics in terms of available ones. The former path is generic and the latter is target specific and is the responsibility of the author of that MD. The latter path finds favor when both the alternates are available.

A compiler has to determine if all the HLL constructs can indeed be mapped to the instructions available on the target. Additionally, if it hopes to be retargetable then this determination can occur at t_{build}[2]. However, the development of such a retargetable compiler may prepare for this determination. GCC (i.e. the development system, at $t_{develop}$) does this by imposing certain conventions on the *names* of the instruction patterns that are specified at $t_{develop}$. Patterns that may be so tested are required to have non null names that do not begin with the "⋆" character. Further, some of such names are standard and assure the compiler of the existence of some required semantics.

Unfortunately, in GCC there is no explicit testing of the "completeness" of the target specification so that every HLL construct can be guaranteed to be mapped to equivalent target expression. It seems that the concept of Standard Pattern Names (SPNs) was evolved with such a goal. Typically, an implementation of a target specification is field tested for "completeness" and over the years the supported targets have matured.

2.2 Influence of the Target System

An "inverse" of the influences of the source on the GCC MD system is the influence of the target system. In this section we use the phrase "target system" in a broad sense to include the target system software and some target hardware properties other than the instruction set of the target processor. Figure 2.3 summarizes these.

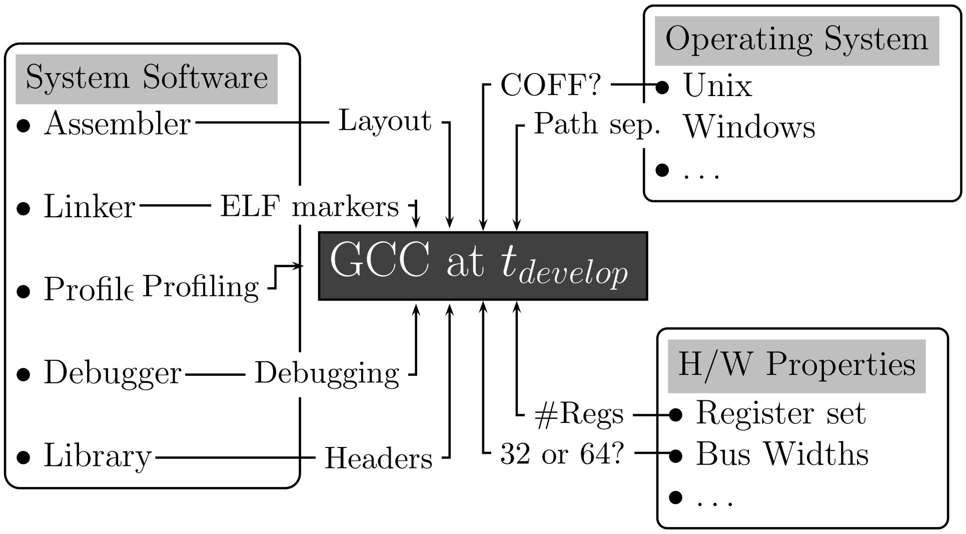

Figure 2.3: Target system influences.

2 In general, it can even occur later but in GCC it occurs at t_{build} .

The processor forms one component of the total hardware that forms a computer. Even the processor is characterized by other parameters, like the registers set, apart from it's instruction set. The values of all these characteristics are simple objects like numbers or strings. Hence these are also expressed using C preprocessor macros. [The Implementation of GCC], page 17 classifies these and [GCC Internals (by Richard Stallman)], page 17 describes these in detail. The registers set offers an interesting example of how a compiler is influenced by the target system. A processor architecture may define additional attributes of registers in it's registers set. Some registers may be dedicated to a certain operation, some may be read only, some may be write only, some may be special purpose while some may be general purpose. The compiler may need to, or even choose to, additionally block certain registers for it's own use. A parametrized compiler, thus, needs to be able to specify the attributes and the subsets of registers on a per target basis. For example, our toy processor has a dedicated, read only register that always has a constant value 0. Since in the typical hardware that surrounds our processor memory operations are slower than register operations, GCC needs to be told to use the "zero" register in all computations that have a numerical 0 as an argument. The register has to be isolated into a subset of it's own and instruction patterns need to be "instructed" to use it. In the example implementation, we will see the specification of the "zero" register into a subset of it's own, and then see the use of this subset in the specification of some instruction patterns of the toy processor.

The OS that mainly drives the hardware also exerts an influence, albeit a less direct one. The "parameters" required are from simple user level differences like the path separator character to use in expressing pathnames[3] to complex differences like the details of the system call interface. The latter are known to the system compiler[4] of the target system and can be borrowed from it's header files. This information is required for GCC to emit the correct target assembly code, especially when it is used as a cross compiler (see [Cross Compilation and GCC], page 17).

Finally, the compiler is but a part of a complete tool chain that converts a source program into a process that runs under the OS. Often we ignore the conversion to process part as that is handled by the running OS and speak of the executable program that is generated by the tool chain[5]. Since the compiler is a part of the tool chain it must emit code that can be consumed by the other tools in the tool chain. Thus, it has to know the syntactic structure of the assembly language to emit so that the assembly program it generates can be assembled by the next tool – the assembler. It may have to help the assembler to mark "regions" (like the data and code) so that the linker may link together a number of assembled object files into a single executable. Such marking also helps other program development tools like the debugger and the profiler.

2.2.1 Target Systems and Cross Compilation

The influence of the target systems is particularly sharp when a retargetable compiler like GCC is deployed as a cross compiler. Cross compilation is a complex topic and is separately dealt with in [Cross Compilation and GCC], page 17. In this section we look at a general cross compilation case to grasp the impact of the influence of the target system. In the

[3] Unix© like OSes use the forward slash "/" and the Windows© family of OSes use the backward slash "\" to separate names in a pathname.

[4] Assumed to be a C compiler

[5] In other words, the OS is usually excluded from the tool chain.

general cross compilation case (called the Canadian cross) the compiler sources are built on the "build system". The system compiler on the build system is used to build the Canadian cross compiler. The generated compiler binaries are hosted (i.e. installed and run) on the "host system". The build is therefore a cross build that generates binaries that run on the host system. The hosted compiler binary the compiles programs to generate binaries that run on the "run system"[6]. The (cross) compiler sources must be retargetable for the host as well as the run system. Retargetability information of the host system is used on the build system by the system compiler on the build system. Retargetability information of the run system is used by the hosted compiler when it compiles it's own sources, but targeted to the run system. Since the host and run systems may be different from the build system, the box labeled "Target System" in Figure 2.1 has to be replicated for each of the build, host and run systems.

2.3 Influence of GCC Architecture

GCC is a software. Depending on the requirements it's architecture changes. These changes can influence the MD system too. There are two major sources of influence within the GCC architecture that can impact it's MD system.

The compilation algorithms that GCC uses may influence the MD system. For instance, instruction selection in GCC is essentially a first-hit-table-look-up type rather than a computation of the best tree cover. As a consequence the MD author is required to order the instruction patterns such that the "best" options are hit first. Also, for some targets, the algorithms may not yield good quality output. In such cases the shortcomings may have to be overcome through the MD system. In earlier versions of GCC, the instruction scheduler did not emit `nop` instructions for some targets. To yield better code, the MD authors had to manually emit the `nop` instruction when the MD was used to emit the final assembly code.

The GCC architecture may evolve to introduce better constructs for specifying target properties. In the same vein, some constructs may be deprecated, and eventually removed, in favor of the better ones. The MD specifications need to be re-written to accommodate such changes. In Figure 2.1 we show an example of the `define_predicate` construct that was recently introduced.

[6] The cross compilation literature refers to the "run system" as the "target system". We have however used the phrase "target system" to mean something else. So to avoid confusion we will use the phrase "run system" to refer to the system for which the final compiler generates binaries.

3 MD System and GCC Architecture

Being a retargetable compiler that fixes the compilation target machine at build time, the build time, t_{build} , divides the GCC system into three different conceptual views[The Conceptual Structure of GCC], page 17. At $t_{develop}$ we have the developer view where the target of compilation has not been fixed and the GCC system has a set of specifications of target properties for each target. At t_{run} the target has been fixed. The conversion of the specifications at $t_{develop}$ to usable code at t_{run} is done at the build time t_{build} as shown in Figure 3.1.

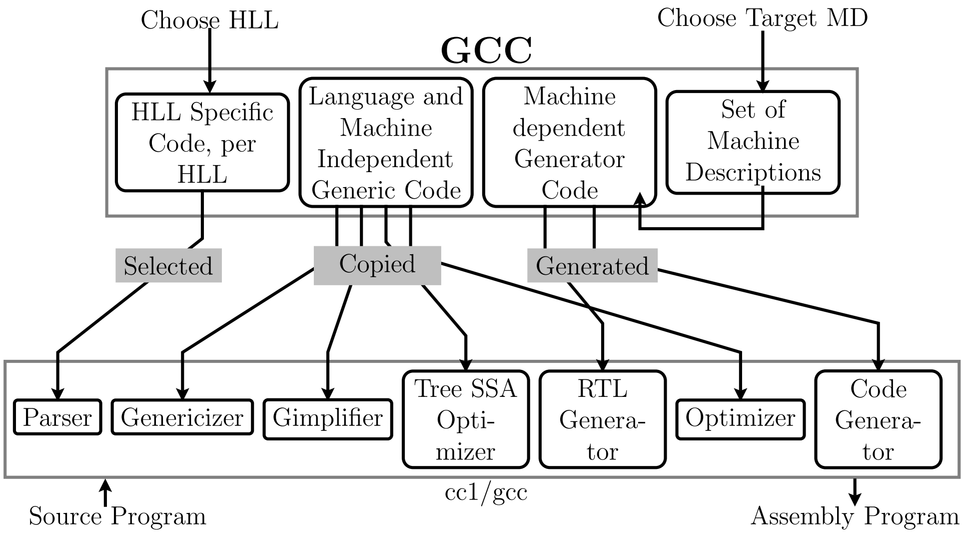

Figure 3.1: The GCC Compiler Generation Framework (CGF).

Figure 3.1 also shows the time durations that describe the distinction between the CGF, GCC, and the target specific compiler, gcc, that is generated at build time. Given the gcc phase sequence and the possible issues that might arise during the compilation of an arbitrary input at t_{run} , the developer view has to create specifications of the target properties at $t_{develop}$. In what follows we describe the process of specification of target properties given the known gcc phase sequence and anticipating the issues that might arise during t_{run} . These motivate the various specification constructs of the MD RTLs. To expose the use of these specifications at t_{run} we examine the generation of the target specific parts of GCC when our toy machine is given as the target at t_{build} .

3.1 The Organisation of the GCC MD System

Section Chapter 2 [Influences on the GCC MD System], page 3 described the various influences on the MD system. The implementation of a MD in GCC captures these influences in two forms: an RTL based specification of the target instruction semantics, and a C preprocessor based specification of all other properties. Some target properties, particularly the ones giving some specific details about the target system software (the OS, for instance) may be separated into respective header files for programming convenience. If the target requires some additional functionality, then a file of C functions is also implemented. The MD system is thus organized into two parts: the mandatory and the optional as follows:

- Mandatory: Must be implemented.
  - `$GCCHOME/gcc/config/<target>/<target>.h`: The C preprocessor based specification of target properties.
  - `$GCCHOME/gcc/config/<target>/<target>.md`: The RTL based specification of target instruction set semantics.
- Optional: May be implemented, either for reasons of need of a particular target or programming convenience.
  - `$GCCHOME/gcc/config/<target>/<target>.c`: Additional target specific functionality, if required.
    - `$GCCHOME/gcc/config/<files>.[ch]`: Properties independent of the target CPU (e.g. system software influences), but common to many targets.
    - `$GCCHOME/gcc/config/<target>/<files>.[ch]`: Target specific properties or functionalities; usually a convenience issue.

3.2 Classification of MD Macros

The C preprocessor macros for machine description have been partially classified in the GCC sources as well as in [GCC Internals (by Richard Stallman)], page 17. We introduce a higher level of abstraction to correspond to the influences in Chapter 2 [Influences on the GCC MD System], page 3. The number of macros being large, the actual listing is in [The Phase wise File Groups of GCC], page 17. The macros relevant for our implementation are in Appendix A [The CPP macros Implementation], page 19.

3.3 The RTL Based MD Specifications

RTL operators (see [The Implementation of GCC], page 17) are used to capture the semantics of target instruction set. They can capture the semantics of the operation. However target instructions also vary in their operands. The number of operands, their nature – register, memory or constants, any restrictions like size etc. are issues that need to be addressed to obtain the complete semantics. These characteristics of the operands form the matching specifications. Finally, the implementation of the specifications may be made convenient by introducing some “specification convenience” RTL objects. It is trivial to see that the RTL system made up of the RTL operators and the RTL memories – pseudoregisters and memory – are powerful enough to capture the semantics of any target. In fact, since the `jump` operation is required to ensure such capability, the GCC code *requires* the jump operation[1] to exist in the specification. For all other patterns, the GCC system calls the RTL expander only if the corresponding pattern exists in the specification.

The RTL based MD specification is composed of two parts:

- RTL operators used to *capture* the target operation semantics.
- MD RTLs are used to *describe* various aspects of the target instruction being specified. These are roughly of the following three kinds:

[1] And the indirect jump operation is also compulsory.

- Pattern definition MD RTLs that are used to *introduce* a new pattern into the specification.
- Pattern operand specification MD RTLs that are used to *capture the operand characteristics* of the target instruction.
- *Specification convenience* MD RTL objects that ease the programming burden.

The illustrative example in [The Implementation of GCC], page 17 brought out most of the issues is specifying the semantics of one instruction and [The Phase wise File Groups of GCC], page 17 describes the purpose of all MD RTLs. In the rest of this section we motivate the essential constructs of the MD RTLs.

The RTL patterns describing target instructions are required at two main points in the GCC system:

1. Gimple → RTL conversion, and
2. RTL → RTL conversion.

The first maps Gimple node semantics to equivalent target instructions. The target either may have a single instruction that can map to the given node, or may require a series of instructions. The second occurs during the various RTL phases. Typically for optimization reasons it is possible that:

- An RTL pattern can be replaced by another RTL pattern, or
- A set of RTLs may be replaced by another RTL pattern, or
- A RTL pattern may be replaced by a set of RTL patterns.

The RTL patterns that capture target instruction semantics and are used during the Gimple → RTL conversion phase are introduced by the `define_insn` and `define_expand` MD RTLs. The RTL patterns that may be used during RTL → RTL phases are introduced by the `define_split`, `define_combine`, and `define_peephole[2]` MD RTLs. Most other MD RTLs prefixed by `define_` are programming conveniences that ease the specification task.

The next level of matching is at the details of the operand level issues.

Before describing the implementation of full target instruction semantics, we give an overview of the semantics of various instructions of our fictitious processor in Chapter 4 [Definition of Fictitious Target System], page 11. Chapter 5 [Implementation of MD for Toy], page 13 then describes the full implementation.

4 Definition of Fictitious Target System

We use a fictional system that runs the same system software that runs on the development system. The processor of the fictional system is the fictitious Toy CPU surrounded by a typical 32 bit hardware environment. For concreteness, it is useful to imagine that the development system is a GNU/Linux system. In particular, the kernel is a Linux©[1] kernel and the tool chain is the GNU©[2] tool chain giving us the known parameters like an ELF object format, AT&T style assembler syntax etc. (See Section 2.2 [Influence of the Target System], page 5.)

The Toy CPU is a simple fictitious processor. It has a 32 bit data and address bus. It has 16 registers, each 32 bit and numbered from 0 to 15. The register number 0 is a read only register that always contains the number 0. Register number 1 is the stack pointer register. The program counter is distinct from the registers and is not directly manipulated. It autoincrements unless the jump instruction has been executed. The primary memory is random access and byte addressable with a four byte word. Data movement from the core memory to the registers occurs through the "load" instruction. Moving data from the registers to the core memory is through the "store" instruction. The instruction set is simple and RISC like:

- Arithmetic:
  - add <reg1> <reg2>

    Source registers: reg1 and reg2, destination register: reg1
  - mult <sreg1> <sreg2> <dreg3> <dreg4>

    Source registers: sreg1 and sreg2, destination registers: dreg3 and dreg4.
- Logical:
  - and <dreg> <sreg1> <sreg2>
  - or <dreg> <sreg1> <sreg2>
  - not <dreg> <sreg1> <sreg2>

  Source registers: sreg1 and sreg2, destination register: dreg
- Bit wise logical:
  - band <dreg> <sreg1> <sreg2>
  - bor <dreg> <sreg1> <sreg2>
  - bnot <dreg> <sreg1> <sreg2>

  Source registers: sreg1 and sreg2, destination register: dreg
- Control flow:
  - jump <reg>

  Register "reg" contains the destination of the jump
- Data movement:
  - load <mem> <reg>
  - wload <mem> <reg>

---

[1] Copyright: Linus Torvalds.

[2] Copyright: FSF, the Free Software Foundation.

- `store <reg> <mem>`
- `wstore <reg> <mem>`

Memory addresses are 32 bit. Bytes are loaded into or read from the least significant byte of the destination register for `load` and `store` instructions. Words are loaded into or read from the least significant byte of the destination register for `wload` and `wstore` instructions. For `wload` and `wstore` instructions the memory addresses are word aligned. The behaviour is *undefined* if the addresses are not word aligned.

The instruction set may be augmented, or the given instructions may be refined to illustrate various issues of writing the GCC MD. Additionally, there is a status register that sets the following flags to report the "success" status of the instructions mentioned beside.

- Comparison flag (CF): "set" (i.e. "1") for logical and bitwise logical operations.
- Zero flag (ZF): "set" (i.e. "1") for arithmetic operations.
- Overflow flag (OF): "set" (i.e. "1") for arithmetic operations that result in overflow.

5 Implementation of MD for Toy

As described in section Chapter 2 [Influences on the GCC MD System], page 3 the MD is influenced by three major factors. Although three different factors can be identified influencing the MD system, the implementation uses two main files to organise the information and a support file as follows:

- `$GCCHOME/gcc/config/toy/toy.h`: The C preprocessor macros that define the values of parameters that can be specified using simple data types like numbers or strings.
- `$GCCHOME/gcc/config/toy/toy.md`: The RTL based specification of target instruction set semantics.
- `$GCCHOME/gcc/config/toy/toy.c`: The additional target specific functionality.

All the macros are listed in the GCC Internals documentation (see [GCC Internals (by Richard Stallman)], page 17) and have also been classified in [The Implementation of GCC], page 17. The MD RTL and RTL operators that are used in the RTL based specification are listed and described together in the GCC Internals documentation (see [GCC Internals (by Richard Stallman)], page 17) and classified in [The Conceptual Structure of GCC], page 17. Our description of the GCC implementation details, [The Implementation of GCC], page 17, also details the transformations that are carried out on the MD system at build time, t_{build} . We avoid reproducing the information already available in the GCC Internals documentation.

The final step is to introduce the new MD into the GCC system. The steps are described (with reference to GCC 4.0.2) in Section 6.1 [Making GCC Aware of the New MD], page 14. The complete implementation of the MD for toy machine is given in the appendices Appendix A [The CPP macros Implementation], page 19, Appendix B [The MD-RTL code Implementation], page 32 and Appendix C [The auxiliary C code Implementation], page 38.

5.1 C Preprocessor Macros

5.2 Implementing CPP macros

5.3 Implementing MD-RTL code

5.3.1 A few t_{run} time situations

5.3.2 Corresponding MD RTL constructs

5.3.3 Examples of MD-RTL code and CPP macros

5.4 Implementing auxiliary C code

6 The New MD and the Build System

6.1 Making GCC Aware of the New MD

6.2 The RTL Object

The main data structure that is used for internal representation of RTL objects, called RTL expressions, or RTX for short, is a union whose main component is the RTX structure. These data structures are found in `$GCCHOME/gcc/rtl.h`.

```
/* Common union for an element of an rtx.  */

union rtunion_def
{
  int rt_int;
  unsigned int rt_uint;
  const char *rt_str;
  rtx rt_rtx;
  rtvec rt_rtvec;
  enum machine_mode rt_type;
  addr_diff_vec_flags rt_addr_diff_vec_flags;
  struct cselib_val_struct *rt_cselib;
  struct bitmap_head_def *rt_bit;
  tree rt_tree;
  struct basic_block_def *rt_bb;
  mem_attrs *rt_mem;
  reg_attrs *rt_reg;
};
typedef union rtunion_def rtunion;

/* RTL expression ("rtx").  */

struct rtx_def GTY((chain_next ("RTX_NEXT (&%h)"),
                    chain_prev ("RTX_PREV (&%h)")))
{
  ENUM_BITFIELD(rtx_code) code: 16;

  ENUM_BITFIELD(machine_mode) mode : 8;

  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned frame_related : 1;
  unsigned return_val : 1;
```

```
  union u {
    rtunion fld[1];
    HOST_WIDE_INT hwint[1];
  } GTY ((special ("rtx_def"), desc ("GET_CODE (&%0)"))) u;
};
```

6.3 Conversion of RTL Specifications to C Code

7 The New MD and the gcc architecture

7.1 Operation of the “Generated” C Code

References

(**Note:** In the URLs below: `$GCCINTDOCSHOME` is `http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs`)

1. Richard. M. Stallman.
   GCC Internals.
   (`http://gcc.gnu.org/onlinedocs/gccint`)
   2007.
2. Abhijat Vichare.
   GCC – An Introduction.
   (`$GCCINTDOCSHOME/html/gcc-basic-info.html`)
   2007.
3. Abhijat Vichare.
   Cross Compilation and GCC.
   (`$GCCINTDOCSHOME/html/gcc-cross-compilation.html`)
   2007.
4. Abhijat Vichare.
   Writing GCC Machine Descriptions.
   (`$GCCINTDOCSHOME/html/gcc-writing-md.html`)
   2007.
5. Abhijat Vichare.
   The Conceptual Structure of GCC.
   (`$GCCINTDOCSHOME/html/gcc-conceptual-structure.html`)
   2007.
6. Abhijat Vichare.
   The Implementation of GCC.
   (`$GCCINTDOCSHOME/html/gcc-implementation-details.html`)
   2007.
7. Abhijat Vichare.
   The Phasewise File Groups of GCC.
   (`$GCCINTDOCSHOME/html/gcc-source-blocks.html`)
   2007.
8. Uday Khedker and Sameera Deshpande.
   Systematic Development of GCC Machine Descriptions.
   (`http://www.cse.iitb.ac.in/~uday/soft-copies/incrementalMD.pdf`)
   2007.

List of Figures

Figure 2.1: Influences on GCC MD 3
Figure 2.2: Source language influences 4
Figure 2.3: Target system influences. 5
Figure 3.1: The GCC Compiler Generation Framework (CGF). 8

Appendix A The CPP macros Implementation

```
/*----------------------------------------------------------*/
/* STORAGE LAYOUT OF TARGET MACHINE                         */
/*----------------------------------------------------------*/

/* If  this  is ‘1’  then  most  significant  bit is  lowest
   numbered  in   instructions  that  operate   on  numbered
   bit-fields. */
#define BITS_BIG_ENDIAN 0

/* If this is  ‘1’ then most significant byte  in a word has
   the lowest number. */
#define BYTES_BIG_ENDIAN 0

/* Define this  if most significant  word of a  multiword is
   lowest numbered.  For toy we can decide arbitrarily since
   there are no machine instructions for them.  */
#define WORDS_BIG_ENDIAN 1

/* Number of bits per addressable unit of memory */
#define BITS_PER_UNIT 8

/* Number of bits per word */
#define BITS_PER_WORD 32

/* Number of units(byte) per word */
#define UNITS_PER_WORD 4

/* Width of a pointer in bits. */
#define POINTER_SIZE 32

/* Define number  of bits in  most basic integer  type.  (If
   undefined, default is BITS_PER_WORD). */
#define INT_TYPE_SIZE 32

/* Boundary  (in  bits) on  which  stack  pointer should  be
   aligned. */
#define STACK_BOUNDARY 32

/* Define this if ‘move’  instructions will actually fail to
   work when given unaligned data. */
#define STRICT_ALIGNMENT 1

/* Biggest alignment that any  data type can require on this
   machine , in bits. */
#define BIGGEST_ALIGNMENT 32
```

```
/* Normal alignment required  for function parameters on the
   stack, in  bits.  All  stack parameters receive  at least
   this much alignment regardless of data type.*/
#define PARM_BOUNDARY 32

/* The maximum number of bytes that a single instruction can
   move quickly between memory  and registers or between two
   memory locations.*/
#define MOVE_MAX 4

/* Recognize any constant value that is a valid address.  */
#define CONSTANT_ADDRESS_P(X)                                   \
 ( GET_CODE (X) == LABEL_REF || GET_CODE (X) == SYMBOL_REF    \
   || GET_CODE (X) == CONST_INT || GET_CODE (X) == CONST      \
   || GET_CODE (X) == HIGH )

/* Value of following macro is 1 if truncating an integer of
   `INPREC'  bits   to  `OUTPREC'  bits  is   done  just  by
   pretending that it is already truncated.  */
#define TRULY_NOOP_TRUNCATION(OUTPREC, INPREC) 1

/* Following macro  is nonzero if access to  memory by bytes
   is slow and undesirable. */
#define SLOW_BYTE_ACCESS 0

/* Maximum number  of registers that  can appear in  a valid
   memory address.  */
#define MAX_REGS_PER_ADDRESS 1


/*---------------------------------------------------------*/
/* INFORMATION REGARDING REGISTERS ON THE TARGET MACHINE*/
/*---------------------------------------------------------*/

/* Number of  Hard-Register available on  target m/c.  These
   Hard-Registers   are   assigned    number   from   0   to
   (FIRST_PSEUDO_REGISTER-1). */
#define FIRST_PSEUDO_REGISTER 12

/* Some  Hard-Registers   are  used  for   specific  purpose
   throughout   the   compilation   process  like   `Program
   counter',  `Stack  pointer'  etc.   The  following  macro
   assign `1' to those registers. */
#define FIXED_REGISTERS {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1}

/* Following macro assign `1' to those registers whose value
```

```
   can be  changed during a function  call.  These registers
   include Fixed  registers as well as  other registers like
   the  register in which  returned value  is stored.   If a
   register  has given '0'  value then  the content  of that
   register  will be saved  automatically on  function entry
   and restored on the exit.*/
#define CALL_USED_REGISTERS {1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1}

/* Returns number of  consecutive Hard-Registers starting at
   reg  REGNO to  hold a  value of  mode MODE.   Following C
   expression   assume  that   each  Register's   length  is
   1-word. */
#define HARD_REGNO_NREGS(REGNO, MODE)   \
  ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)

/* Register used for the program counter. */
#define PC_REGNUM 11

/* Register used for pushing function arguments. */
#define STACK_POINTER_REGNUM 10

/* Base register  used for access to local  variables of the
   function.  */
#define FRAME_POINTER_REGNUM 9

/* Base  register  used  for  access  to  arguments  of  the
   function.  */
#define ARG_POINTER_REGNUM 8

/* Register in  which address to store a  structure value is
   passed to a function.  */
#define STRUCT_VALUE_REGNUM 7

/* Following macro is 1 if  hard register 'REGNO' can hold a
   value of machine-mode 'MODE'. */
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1

/* Value  of  the  following  macro  should  be  nonzero  if
   functions must have frame pointers. */
#define FRAME_POINTER_REQUIRED 1

/* A C expression that is nonzero if Register with reg. name
   'X' is valid for use as an index register. */
#define REG_OK_FOR_INDEX_P(X) 0

/* A C expression that is nonzero if Register with reg. name
   'X' is valid for use as an base register. */
```

```
#define REG_OK_FOR_BASE_P(X) 0

/* A  C   expression  that  is  nonzero   if  Register  with
   reg.  no.   ‘REGNO’  is  valid   for  use  as   an  index
   register. */
#define REGNO_OK_FOR_INDEX_P(REGNO) 0

/* A  C   expression  that  is  nonzero   if  Register  with
   reg. no. ‘REGNO’ is valid for use as an base register. */
#define REGNO_OK_FOR_BASE_P(REGNO) 0

/* Following macro define  the maximum number of consecutive
   registers of class ‘CLASS’ needed to hold a value of mode
   ‘MODE’.  The value  of the macro ‘CLASS_MAX_NREGS’ should
   be the  maximum value of  macro ‘HARD_REGNO_NREGS (regno,
   mode)’ for all ‘regno’ values in the class ‘CLASS’. */
#define CLASS_MAX_NREGS(CLASS, MODE)    \
   ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1) / UNITS_PER_WORD)



#define INITIALIZE_TRAMPOLINE(TRAMP, FNADDR, CXT)              \
{                                                              \
  emit_move_insn                                               \
     (gen_rtx_MEM                                              \
         (SImode, plus_constant (TRAMP, 2)), CXT);             \
  emit_move_insn                                               \
     (gen_rtx_MEM                                              \
          (SImode, plus_constant (TRAMP, 9)), FNADDR);         \
}
#define FIRST_PARM_OFFSET(FNDECL) 0
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 0)
#define CPP_PREDEFINES ""
#define LIBCALL_VALUE(MODE)  gen_rtx_REG (MODE, 0)

/* Output assembler code to FILE to increment profiler label
*    # LABELNO for profiling a function entry.  */

#define FUNCTION_PROFILER(FILE, LABELNO)          \
  fprintf (FILE,                                  \
           "\tmovw &.LP%d,%%r0\n\tjsb _mcount\n", \
           (LABELNO))

/* This  is how  to output  an assembler  line that  says to
*    advance the location counter by SIZE bytes.  */

/* The ‘space’ pseudo in  the text segment outputs nop insns
```

```
*    rather than 0s, so we  must output 0s explicitly in the
*    text segment.  */

#define ASM_OUTPUT_SKIP(FILE,SIZE) {}
#define FUNCTION_ARG_REGNO_P(N) 0
#define PREFERRED_RELOAD_CLASS(X,CLASS)  (CLASS)
#define INITIAL_FRAME_POINTER_OFFSET(DEPTH) (DEPTH) = 0;


/* Following  macro is  an  C expression  which defines  the
   machine-dependent operand constraint letters for register
   classes. If ‘C’ is such a letter, the value should be the
   register class corresponding to it.  Otherwise, the value
   should be NO_REGS. */
#define REG_CLASS_FROM_LETTER(C)                \
        ((C) == ’r’ ? GENERAL_REGS : NO_REGS)

/* Following   defines   the   machine-dependent   operand
   constraint letters (‘I’, ‘J’,  ‘K’, ... ‘P’) that specify
   particular ranges  of integer values.   If ‘C’ is  one of
   those letters, the  expression should check that ‘VALUE’,
   an integer, is  in the appropriate range and  return 1 if
   so, 0 otherwise.  If ‘C’ is not one of those letters, the
   value should be 0 regardless of value.

   In the .md file of toy  m/c, we are not using any special
   constraint  letter. So  this  macro returns  ‘0’ for  all
   letters. */
#define CONST_OK_FOR_LETTER_P(VALUE, C)  0


/* Following define the machine-dependent operand constraint
   letters  that specify  particular ranges  of const_double
   values (‘G’ or ‘H’).  Rest are the same as above. */
#define CONST_DOUBLE_OK_FOR_LETTER_P(VALUE, C)  0

/* Following enumeration define the classes of registers for
   register constraints in the .md file.  Classes ‘ALL_REGS’
   include  all  hard   regs.  ‘NO_REGS’  class  contain  no
   registers.

   ‘GENERAL_REGS’ is the class  of registers that is allowed
   by ‘g’ or ‘r’ in a register constraint.

   A   larger-numbered  class   must   never   be   contained
   completely in a smaller-numbered class.
```

```
   ‘LIM_REG_CLASSES’,  is not  a register  class  but rather
   tells how many classes there are. */
enum reg_class{
   NO_REGS,
   GENERAL_REGS,
   ALL_REGS,
   LIM_REG_CLASSES
};

/* Following macro tells the number of Register classes. */
#define N_REG_CLASSES (int)LIM_REG_CLASSES

/* Following is  an initializer containing the  names of the
   register classes  as C string constants.  These names are
   used in writing some of the debugging dumps. */
#define REG_CLASS_NAMES {"NO_REGS", "GENERAL_REGS", "ALL_REGS"}

/* Following macro define which registers fit in which classes.
   Register r is in the class if mask & (1 << r) is 1. */

#define REG_CLASS_CONTENTS {      \
   {0},       /* NO_REGS */       \
   {0xff},    /* GENERAL_REGS */  \
   {0xfff},    /* ALL_REGS */     \
}

/* A macro which define the name of class to which a valid index
   register must belong. */
#define INDEX_REG_CLASS NO_REGS

/* Return the class number of the smallest class containing
   reg number ‘REGNO’. */
#define REGNO_REG_CLASS(REGNO)            \
        (((REGNO) < 7) ? GENERAL_REGS : ALL_REGS)

/* The Register class for base registers. */
#define BASE_REG_CLASS GENERAL_REGS

/* Following is an  C initializer containing the assembler’s
   names for the  machine registers, each one as  a C string
   constant.   This is what  translates register  numbers in
   the compiler into assembler language. */
#define REGISTER_NAMES \
    {"ret", "r1", "r2", "r3", "r4", "r5", \
     "r6",  "r7", "r8", "fp", "sp", "pc"}

/*-----------------------------------------------------------*/
```

```
/* INFORMATION REGARDING FUNCTION CALL. */
/*-----------------------------------------------------------*/

/* Define  how  to  find  the value  returned  by  a
   function.  'VALTYPE'  is  the  data  type  of  the
   value. 'TYPE_MODE' is used to find out the mode of a data
   type.  If the  precise  function being  called is  known,
   'FUNC'  is a  tree node  (FUNCTION_DECL;  otherwise, FUNC
   will be a null pointer.

   On the toy, the return value is in r0. */
#define FUNCTION_VALUE(VALTYPE, FUNC)             \
    gen_rtx_REG (TYPE_MODE (VALTYPE), 0)

/* Following macro define how  to find the value returned by
   a library  function assuming the value has  mode MODE. On
   the  toy m/c,  the  return value  will  be in  register-0
   regardless.  */
#define LIBCALL_VALUE(MODE) gen_rtx_REG (MODE, 0)

/* This is a Data type for declaring a variable that is used
   as the first argument of 'FUNCTION_ARG' and other related
   values and used  to hold the number of  bytes of argument
   so far.

   There is no need  to record in 'CUMULATIVE_ARGS' anything
   about  the  arguments  that  have  been  passed  on  the
   stack. The compiler has  other variables to keep track of
   that.  For target  machines  on which  all arguments  are
   passed on the  stack, there is no need  to store anything
   in  'CUMULATIVE_ARGS'; however,  the data  structure must
   exist and should not be empty, so use int. */
#define CUMULATIVE_ARGS int

/* Alignment required for a function entry point, in bits. */
#define FUNCTION_BOUNDARY 32

/* This macro is used to indicate the number of bytes that a
   function  pops on returning  or '0'  if function  pops no
   arguments  and the  caller  must therefore  pop them  all
   after the function returns.

   'FUNDECL' is  the declaration node of the  function (as a
   tree).  'FUNTYPE' is the data  type of the function (as a
   tree).  'SIZE' is the number of bytes of arguments passed
   on the  stack.  Here we are assuming  that every function
   will always pop their arguments. */
```

```
#define RETURN_POPS_ARGS(FUNDECL,FUNTYPE,SIZE) (SIZE)

/* Following is a macro  for initializing the variable 'CUM'
   for scanning the argument  list. The value of 'FNTYPE' is
   the tree  node for  the data type  of the  function which
   will  receive the  args,  or '0'  if  the args  are to  a
   compiler   support  library   function.   The   value  of
   'INDIRECT' is  nonzero when processing  an indirect call,
   for example a call  through a function pointer. The value
   of 'INDIRECT'  is '0' for  a call to an  explicitly named
   function, a library function call.

   When  processing a  call  to a  compiler support  library
   function,  'LIBNAME' identifies  which one.  'LIBNAME' is
   '0' when an ordinary C function call is being processed.

   Thus, each time this macro is called, either 'LIBNAME' or
   'FNTYPE' is nonzero, but never both of them at once.  */
#define INIT_CUMULATIVE_ARGS(CUM,FNTYPE,LIBNAME,INDIRECT) \
     ((CUM) = 0)

/* Following is  an C statement  that is used to  update the
   summarizer variable 'CUM'.  The values 'MODE', 'TYPE' and
   'NAMED'  describe the properties  of the  argument.  This
   macro need  not do anything  if the argument  in question
   was passed on the stack. */
#define FUNCTION_ARG_ADVANCE(CUM, MODE, TYPE, NAMED)          \
   ((CUM) = CUM + (((MODE) != BLKmode)                        \
                  ? (GET_MODE_SIZE (MODE) + 3) & ~3           \
                  : (int_size_in_bytes (TYPE) + 3) & ~3))

/* This macro controls whether a function argument is passed
   in a  register or not.   If yes then return  the register
   number.  Here we are  assuming that all the arguments are
   passed on the stack. */
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED) 0

/*-----------------------------------------------------*/
/* INFORMATION REGARDING STACK LAYOUT, MACHINE MODE ETC. */
/*-----------------------------------------------------*/

/* stack growing downwards */
#define STACK_GROWS_DOWNWARDS

/* frame growing downwards */
#define FRAME_GROWS_DOWNWARDS
```

```
/* Offset from the frame pointer to the first local variable
   slot to be allocated. */

#define STARTING_FRAME_OFFSET 0

/* Define  this as 1 if  `char' should by default  be signed
   else 0. */
#define DEFAULT_SIGNED_CHAR 0

/* Specify the machine mode for pointers. */
#define Pmode SImode

/* Specify the  machine mode that this machine  uses for the
   index in the tablejump instruction. */
#define CASE_VECTOR_MODE SImode

/* Following  is an  alias  for the  machine  mode used  for
   memory  references to functions  being called,  in `call'
   RTL expressions. */
#define FUNCTION_MODE QImode

/* Following is an C expression  that is nonzero if `X' is a
   legitimate  constant  for  an  immediate operand  on  the
   target  machine.   We   can  assume  that  `X'  satisfies
   `CONSTANT_P', so we need not check this.  In fact, `1' is
   a suitable  definition for  this macro on  machines where
   anything `CONSTANT_P' is valid. */
#define LEGITIMATE_CONSTANT_P(X) 1

/* Try machine-dependent  ways of modifying  an illegitimate
   address `X' to be legitimate.  If we find one, return the
   new,  valid address.   This  macro is  used  in only  one
   place: `memory_address' in explow.c.

   `X'   will  always   be   the  result   of   a  call   to
   `break_out_memory_refs', and  `OLDX' will be  the operand
   that was given to that function to produce `X'.

   It is always safe for this macro to do nothing. */
#define LEGITIMIZE_ADDRESS(X,OLDX,MODE,WIN) {}

#define GO_IF_LEGITIMATE_ADDRESS(MODE, X, LABEL) {}

/* Following is an  C expression that is nonzero  if a value
   of  mode `MODE1'  is accessible  in mode  `MODE2' without
   copying.
```

```
   If      ‘HARD_REGNO_MODE_OK      (R,      MODE1)’      and
   ‘HARD_REGNO_MODE_OK (R,  MODE2)’ are always  the same for
   any  R, then ‘MODES_TIEABLE_P  (MODE1, MODE2)’  should be
   nonzero. If they differ for any R, you should define this
   macro to return zero  unless some other mechanism ensures
   the accessibility of the value in a narrower mode.

   You should define this macro to return nonzero in as many
   cases  as  possible since  doing  so  will  allow GCC  to
   perform better register allocation. */
#define MODES_TIEABLE_P(MODE1, MODE2) 0

/* Following macro  is an C statement  or compound statement
   with  a  conditional  goto  ‘LABEL’; executed  if  memory
   address  ‘ADDR’  (an  RTX)  can have  different  meanings
   depending on the machine  mode of the memory reference it
   is used for or if the address is valid for some modes but
   not others.

   Autoincrement and  autodecrement addresses typically have
   mode-dependent effects. */
#define GO_IF_MODE_DEPENDENT_ADDRESS(ADDR,LABEL) {}

/*----------------------------------------------------------*/
/* INFORMATION REGARDING ASSEMBLY OUTPUT CODE */
/*----------------------------------------------------------*/

/* First line of every assembly file should be ‘.file "input
   file name" ’.  Following macro output the input file name
   on that line. For ex.  .file "test3.c". */
#define ASM_FILE_START(FILE) output_file_directive      \
        ((FILE), main_input_filename)

/* A C string constant for text to be output before each asm
   statement  or  group  of  consecutive  ones  in  assembly
   file. */
#define ASM_APP_ON "#APP\n"

/* A C string constant for  text to be output after each asm
   statement  or  group  of  consecutive  ones  in  assembly
   file. */
#define ASM_APP_OFF "#NO_APP\n"

/* Following  is an  C expression  whose value  is  a string
   containing  the assembler  operation that  should precede
   instructions and read-only data. */
#define TEXT_SECTION_ASM_OP "\t.text"
```

```
/* Following  is an  C expression  whose value  is  a string
   containing  the  assembler  operation  to  identify  the
   following data as writable initialized data. */
#define DATA_SECTION_ASM_OP "\t.data"

/* Following is an C statement that is used to output to the
   'FILE',  the  assembler  definition  of  a  label  named
   'NAME'. */
#define ASM_OUTPUT_LABEL(FILE,NAME)     \
do{                                     \
   assemble_name (FILE, NAME);          \
   fputs (":\n", FILE);                 \
}while (0)


/* Following is an C statement that is used to output to the
   'FILE'  some commands  that  will make  the label  'NAME'
   global.   That  is, available  for  reference from  other
   files. */

#define GLOBAL_ASM_OP            "\t.globl\t"

/* It is  used to output  an assembler code to  'FILE' which
   will push hard register number 'REGNO' onto the stack. */
#define ASM_OUTPUT_REG_PUSH(FILE,REGNO) \
   fprintf (FILE, "\tpushw %s\n", reg_names[REGNO])

/* It is  used to output  an assembler code to  'FILE' which
   will  pop  hard  register  number  'REGNO'  off  of  the
   stack. */
#define ASM_OUTPUT_REG_POP(FILE,REGNO)  \
   fprintf (FILE, "\tpopw %s\n", reg_names[REGNO])

/* Following macro output to the 'FILE' an assembler command
   to advance the location counter to a multiple of 2 to the
   'POWER' bytes.   'POWER' will be  a C expression  of type
   int. */
#define ASM_OUTPUT_ALIGN(FILE,POWER)            \
   if ((POWER) != 0)                            \
        fprintf (FILE, "\t.align %d\n", 1 << (POWER))

/* Following is  an C statement  which is used to  output to
   'FILE',  a  label whose  name  is  made  from the  string
   'PREFIX' and the number 'NUM'. */
#define ASM_OUTPUT_INTERNAL_LABEL(FILE, PREFIX, NUM)      \
   fprintf (FILE, ".TOY%s%d:\n", PREFIX, NUM)
```

```
/* Following is an  C statement which is used  to store into
   the string ‘LABEL’,  a label whose name is  made from the
   string ‘PREFIX’ and the number ‘NUM’.

   This string, when output subsequently by ‘assemble_name’,
   should        produce        the        output        that
   ‘ASM_OUTPUT_INTERNAL_LABEL’  would produce with  the same
   prefix and num. */
#define ASM_GENERATE_INTERNAL_LABEL(LABEL,PREFIX,NUM)   \
   sprintf (LABEL, ".TOY%s%d", PREFIX, NUM)

/* This macro says  that how to output an  assembler line to
   define a local common symbol. */
#define ASM_OUTPUT_LOCAL(FILE, NAME, SIZE, ROUNDED) {}

/* This macro says  that how to output an  assembler line to
   define a global common symbol. */
#define ASM_OUTPUT_COMMON(FILE, NAME, SIZE, ROUNDED) {}

/* Following is an C  compound statement to output to ‘FILE’
   the assembler syntax for  an instruction operand ‘X’. ‘X’
   should be an RTL expression.   ‘CODE’ is a value that can
   be used  to specify one  of several ways of  printing the
   operand.*/
#define PRINT_OPERAND(FILE, X, CODE)            \
        print_operand((FILE), (X), (CODE))

/* Print Register Name */
#define PRINT_REG(X, CODE, FILE) print_reg((X), (CODE), (FILE))
/* */
#define PRINT_OPERAND_ADDRESS(FILE, ADDR)
/*-------------------------------------------------------*/
/* Run-time Target Specification */
/*-------------------------------------------------------*/

/* This macro  defines names of  command options to  set and
   clear bits in ‘target_flags’.

   Each subgrouping contains a string constant, that defines
   the option name, a number, which contains the bits to set
   in  target_flags,  and  a  second  string  which  is  the
   description  displayed  by ‘--help’.   If  the number  is
   negative  then  the  bits  specified by  the  number  are
   cleared instead of being set.

   The actual option  name is made by appending  ‘-m’ to the
```

```
   specified name.  Non-empty  description strings should be
   marked with N_(...) for ‘xgettext’.

   One of  the subgroupings should  have a null  string. The
   number  in  this  grouping   is  the  default  value  for
   target_flags. */

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

/*----------------------------------------------------------*/
/* Trampolines */
/*----------------------------------------------------------*/

/* A C expression that define  the size of the trampoline in
   number of bytes. */
#define TRAMPOLINE_SIZE 0

/* Here  we  define machine-dependent  flags  and fields  in
*    cc_status (see ‘conditions.h’).  */

#define NOTICE_UPDATE_CC(EXP, INSN)   \
{                                     \
   { CC_STATUS_INIT; }                \
}
```

Appendix B The MD-RTL code Implementation

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    GCC Machine description for Toy architecture     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; ----------------------------------------------------
;; PATTERN FOR ARITHMETIC OPERATIONS ;;
;; ----------------------------------------------------

;; Pattern for an Add operation:
;;   No. of operand = 3
;;   No. of In-operand = 2
;;   No. of Out-operand = 1
;;   Type of operand = SI or 4-byte integer.
;; x <- y + z

(define_insn "addsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=mr")
        (plus:SI (match_operand:SI 1 "general_operand" "mri")
                 (match_operand:SI 2 "general_operand" "mri")))
  ]
  ""
  ;;;;; "%0 <- %1 + %2;\t\t// ADD(si3)\n"
  "add\t %1\t %2\t %0\t;; ADD(si3)\n"
)

(define_insn "mulsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=mr")
        (mult:SI (match_operand:SI 1 "general_operand" "mri")
                 (match_operand:SI 2 "general_operand" "mri")))
  ]
  ""
  ;;;;; "%0 <- %1 * %2;\t\t// MUL(si3)\n"
  "mul\t %1\t %2\t %0\t;; MUL(si3)\n"
)

(define_insn "divsi3"
  [(set (match_operand:SI 0 "nonimmediate_operand" "=mr")
        (div:SI (match_operand:SI 1 "general_operand" "mri")
                 (match_operand:SI 2 "general_operand" "mri")))
  ]
  ""
  ;;;;; "%0 <- %1 / %2;\t\t// DIV(si3)\n"
  "div %1\t %2\t %0\t;; DIV(si3)\n"
)
```

```
;; ----------------------------------------------------
;; PATTERN FOR MOVE OPERATIONS ;;
;; ----------------------------------------------------

;; Pattern for an Move operation:
;; No. of operand = 2
;; No. of In-operand = 1
;; No. of Out-operand = 1
;; Type of operand = SI or 4-byte integer.
;; x = y

(define_expand "movsi"
  [(set (match_operand:SI 0 "general_operand" "")
        (match_operand:SI 1 "general_operand" ""))
  ]
  ""
  "")

(define_insn "*movsi_1"
  [(set (match_operand:SI 0 "move_dest_operand" "")
        (match_operand:SI 1 "move_src_operand" ""))]
  ""
  "*
        toy_move_insn (operands[0], operands[1]);
  ")

(define_insn "*movsi_1"
  [(set (match_operand:SI 0 "" "")
        (subreg:SI (match_operand:SI 1 "" "") 0))]
  ""
  ;;;;; "%0 = %1\t\t// MOV"
  "mov\t %1\t %0\t\t;; MOV"
)

;; ----------------------------------------------------
;; PATTERN FOR CALL OPERATIONS ;;
;; ----------------------------------------------------

;; Pattern for an Call operation:
;; No. of operand = 2
;; No. of In-operand = 2
;; No. of Out-operand = 0
;; Type of operand:
;;    0 => Represents memory address of function.
;;    1 => Len of arguments to this function in byte.
;; call fun len
```

```
(define_insn "call"
  [(call (match_operand:QI 0 "" "")
         (match_operand:SI 1 "" ""))]
  ""
  ;;;;; "call %M0, %1\t\t// CALL mem_addr arg_len"
  "call\t %M0\t %1\t\t;; CALL mem_addr arg_len"
)

;; -------------------------------------------------------

;; Pattern for an Call operation which return a value:
;; No. of operand = 3
;; No. of In-operand = 2
;; No. of Out-operand = 1
;; Type of operand:
;;   0 => Register in which value is returned by function.
;;   1 => Represents memory address of function.
;;   2 => Len of arguments to this function in byte.
;; call fun len

;; (define_insn "call_value"
;;   [(set (match_operand 0 "" "=r")
;;         (call (match_operand:QI 1 "" "")
;;               (match_operand:SI 2 "" "")))]
;;   ""
;;   "callVal\t %0\t %M1\t %2\t;; CALL reg mem_addr arg_len"
;; )

;; -------------------------------------------------------
;; PATTERN FOR JUMP OPERATIONS ;;
;; -------------------------------------------------------

;; Pattern for the Indirect Jump operation.
;; No. of operand = 1
;; No. of In-operand = 1
;; No. of Out-operand = 0
;; Type of operand = SI or 4-byte integer.
;; jump x

(define_insn "indirect_jump"
  [(set (pc) (match_operand:SI 0 "" ""))]
  ""
  ;;;;; "jmp %0\t\t// INDIRECT JUMP"
  "jmp\t %0\t\t\t;; INDIRECT JUMP"
)

;; -------------------------------------------------------
```

```
;; Pattern for the Direct Jump operation.
;; No. of operand = 1
;; No. of In-operand = 1
;; No. of Out-operand = 0
;; Type of operand = Label.
;; jump x

(define_insn "jump"
  [(set (pc) (label_ref (match_operand 0 "" "")))]
  ""
  ;;;;; "jmp %0\t\t// DIRECT JUMP"
  "jmp\t %0\t\t\t;; DIRECT JUMP"
)

;; -------------------------------------------------------
;; PATTERN FOR MISCELLANEOUS OPERATIONS ;;
;; -------------------------------------------------------

;; Pattern for the No operation.
;; No. of operand = 0
;; No. of In-operand = 0
;; No. of Out-operand = 0
;; NOP

(define_insn "nop"
  [(const_int 0)]
  ""
  ;;;;; "NOP\t\t// NOP"
  "NOP\t\t\t\t;; NOP"
)

;; -------------------------------------------------------
;; PATTERN FOR COMPARISION OPERATIONS ;;
;; -------------------------------------------------------

;; Pattern for the comparision operation.
;; No. of operand = 3
;; No. of In-operand = 2
;; No. of Out-operand = 1
;; Type of operand:
;;   cc0 => Condition code register which store the result
;;          of comparision.
;;     0 => SI operand (Should not be immediate).
;;     1 => SI operand.
;; cmp x y
```

```
;(define_expand "cmpsi"
;;  [(set (cc0)
;;        (compare (match_operand:SI 0 "general_operand" "")
;;                 (match_operand:SI 1 "general_operand" "")))]
;  ""
;  "")

(define_insn "*cmpsi_1"
  [(set (cc0)
        (compare (match_operand:SI 0 "cmp_dest_op" "")
                 (match_operand:SI 1 "cmp_src_op" "i")))]
  ""
  "*
        toy_cmp_insn (operands[0], operands[1]);
  ")

;; -------------------------------------------------------
;; PATTERN FOR CONDITIONAL BRANCH OPERATIONS ;;
;; -------------------------------------------------------

;; Pattern for the conditional branch operations.
;; No. of operand = 1
;; No. of In-operand = 1
;; No. of Out-operand = 0
;; Type of operand:
;;    0 => Label.
;; jump_eq label

(define_insn "beq"
  [(set (pc)
        (if_then_else (eq (cc0)(const_int 0))
                      (label_ref (match_operand 0 "nitin7" ""))
                      (pc)))]
  ""
  ;;;;; "jmp %0\t\t// Branch if equal"
  "jmp\t %0\t\t\t;; Branch if equal"
)

;;(define_insn "beq"
;;  [(set (pc)
;;        (if_then_else (le (cc0) (const_int 0))
;;                      (label_ref (match_operand 0 "" ""))
;;                      (pc)))]
;;  ""
;;  "nitin ha ha")
```

```
;;(define_insn "tstsi"
;;  [(set (cc0)
;;        (match_operand:SI 0 "nonimmediate_operand" "mr"))]
;;  ""
;;  "TSTW %0")

;; ;; -------------------------------------------------------
;; (define_insn "sgt"
;;   [(set (match_operand:QI 0 "nitin5" "")
;;         (gt:QI (cc0) (const_int 0)))]
;;   "nitin6()"
;;   "\nSQT=%0\n")
;;
;; (define_insn "sle"
;;   [(set (strict_low_part
;;            (subreg:QI (match_operand:SI 0 "nitin3" "") 0))
;;         (le:QI (cc0) (const_int 0)))]
;;   "nitin9()"
;;   "\nSLE=%0\n")
;;
```

Appendix C The auxiliary C code Implementation

```
#include "config.h"
#include "system.h"
#include "insn-config.h"
#include "rtl.h"
#include "function.h"
#include "real.h"
#include "recog.h"
#include "output.h"
#include "regs.h"
#include "tree.h"
#include "expr.h"
#include "hard-reg-set.h"
#include "tm_p.h"
#include "target.h"
#include "target-def.h"

/* Register Names */
static const char *register_names[] = REGISTER_NAMES;

struct gcc_target targetm = TARGET_INITIALIZER;

void print_operand(FILE*, rtx, int);
void print_reg(rtx, int, FILE*);

/*******************************************************/
void print_operand(FILE* file, rtx x, int code){
  switch(code){
  case 'M':
    if(GET_CODE(XEXP(x,0)) == REG){
      fprintf(file, "*(");
      PRINT_REG(XEXP(x, 0), 0, file);
      fprintf(file, ")");
    } else {
      if(GET_CODE(XEXP(x, 0)) == PLUS){
        fprintf(file, "*(");
        print_operand(file, XEXP(x, 0), 0);
        fprintf(file, "+");
        print_operand(file, XEXP(x, 1), 0);
        fprintf(file, ")");
      } else
        fprintf(file, "%d", GET_CODE(XEXP(x, 0)));
    }
    break;

  default:
```

```
    if(GET_CODE(x) == REG)
      PRINT_REG(x, code, file);
    else if (GET_CODE(x) == CONST_INT)
      fprintf(file, HOST_WIDE_INT_PRINT_DEC, INTVAL(x));
    else if (GET_CODE(x) == SYMBOL_REF)
      fprintf(file, "&%s", XSTR(x, 0));
    else
      fprintf(file, "%d", GET_CODE(XEXP(x, 0)));
  }
}
/**********************************************************/

/* `x'  is a REGISTER  rtx. This function takes  the alphanumeric
name of  the REGISTER from  the array `register_names'  and print
that on `file'. */

void print_reg(rtx x, int code, FILE* file){
  fputs(register_names[REGNO(x)], file);
}

/**********************************************************/

/* This  function takes care  of destination operand of  a `move'
operation. */

int move_dest_operand(rtx op, enum machine_mode mode){
  rtx x0;
  rtx x1, x2;
  int retVal = FALSE;

  /* If the  mode of operand is not as required  and not equal to
     `VOIDmode' then return FALSE. */

  if(GET_MODE(op) != mode && mode != VOIDmode)
    retVal =  FALSE;

  /* If operand is an memory operand. */
  if(GET_CODE(op) == MEM) {
    x0 = XEXP(op,0);
    switch(GET_CODE(x0)) {

      /* If the operand is  an `PLUS' operand and its first child
         is an `REG' and second one is an `INT', then return
         TRUE. */

    case PLUS:
      x1 = XEXP(x0,0);
```

```
      x2 = XEXP(x0,1);
      if((GET_CODE(x2) == CONST_INT) && (GET_CODE(x1) == REG))
        retVal = TRUE;
      break;

      /*  If the  destination  is either  'REG'  or 'SYMBOL'  and
          contain the address of some memory location, then
          return TRUE. */
      /*                case SYMBOL_REF:*/
    case REG:
      retVal = TRUE;
      break;

    default:
      retVal = FALSE;
    }
  }
  /*  If the  destination is  either 'REG'  or  'SYMBOL_REF' then
      return TRUE. */

    else if (GET_CODE(op) == REG || GET_CODE(op) == SYMBOL_REF)
      retVal =  TRUE;

  return retVal;
}

/***********************************************************/
/*  This  function takes  care  of  source  operand of  a  'move'
    operation. */

int move_src_operand(rtx op,enum machine_mode mode){
  rtx x0;
  rtx x1,x2;
  int retVal= FALSE;

  /* If the  mode of operand is not as required  and not equal to
     'VOIDmode' then return FALSE. */
  if(GET_MODE(op) != mode && mode != VOIDmode)
    retVal =  FALSE;

  /* If operand is an memory operand. */
  if(GET_CODE(op) == MEM){
    x0 = XEXP(op,0);
    switch(GET_CODE(x0)){

      /* If the operand is  an 'PLUS' operand and its first child
         is an 'REG' and second one is an 'INT', then return
```

```
         TRUE. */
    case PLUS:
      x1 = XEXP(x0,0);
      x2 = XEXP(x0,1);
      if((GET_CODE(x2) == CONST_INT) && (GET_CODE(x1) == REG))
        retVal = TRUE;
      break;

      /* If the destination is ‘REG’ and contain the the address
         of some memory lacation, then return TRUE. */
    case REG:
      retVal = TRUE;
      break;

    default:
      retVal  = FALSE;
    }
  }
  /* If the destination is ‘REG’ or ‘SYMBOL_REF’ or ‘CONST_INT’,
     then return TRUE. */
  else{
    if(GET_CODE(op) == CC0 ||
       GET_CODE(op) == REG ||
       GET_CODE(op) == SYMBOL_REF ||
       GET_CODE(op) == CONST_INT){
      retVal = TRUE;
    }
  }

  return retVal;
}

/*************************************************************/
/*  Function  to  generate  the  assembly  code  for  the  ‘move’
    instruction.*/
const char* toy_move_insn(rtx dest, rtx src){
  /* If destination is an ‘REG’ operand. */
  if (GET_CODE(dest) == REG) {
    if (GET_CODE(src) == MEM)
       return "mov\t %M1\t %0\t\t;; MOVE Dest = REG, Src = MEM";
    else
       return "mov\t %1\t %0\t\t;; MOVE Dest = REG, Src = REG";
  }

  if (GET_CODE(dest) == MEM) {
    if (GET_CODE(src) == MEM)
      return "mov\t %M1\t %M0\t\t;; MOVE Dest = MEM, Src = MEM";
```

```
    else
      return "mov\t %1\t %M0\t\t;; MOVE Dest = MEM, Src = REG";
  }
}

/***************************************************************/
const char* toy_cmp_insn(rtx left, rtx right){
  if(GET_CODE(left) == MEM) {
    if (GET_CODE(right) == MEM)
       return
          "cmp\t %M1\t %M0 \t\t;; CMP Dest = MEM, Src = MEM";
    else
       return
          "cmp\t %1\t %M0 \t\t;; CMP Dest = MEM, Src = REG";
  }

  if (GET_CODE(left) == REG) {
    if(GET_CODE(right) == MEM)
      return "cmp\t %M1\t %0 \t\t;; CMP Dest = REG, Src = MEM";
    else
      return "cmp\t %1\t %0 \t\t;; CMP Dest = REG, Src = REG";
  }
}

int nitin(rtx left, rtx right){
  printf("HIII\n");
  return 1;
}

int nitin1(rtx left, rtx right){
  printf("H-1\n");
  return 1;
}

int nitin2(rtx left, rtx right){
  printf("H-2\n");
  return 1;
}

int nitin3(rtx left, rtx right){
  printf("sle\n");
  return 1;
}

int nitin5(rtx left, rtx right){
  printf("sgt\n");
  return 1;
```

```
}

int nitin8(rtx left, rtx right){
  printf("H-8\n");
  return 1;
}

int nitin6(){
  printf("H-6\n");
  return 1;
}

void nitin12(){
  printf("H-12\n");
}

void nitin11(){
  printf("H-11\n");
}

void nitin10(){
  printf("H-10\n");
}

int nitin13(){
  printf("H-13\n");
  return 1;
}
int nitin9(){
  printf("H-9\n");
  return 1;
}

int nitin7(rtx left, rtx right){
  printf("If_then_else\n");
  return 1;
}

int nitin4(rtx op, enum machine_mode mode){
  enum rtx_code code;
  printf("Nitin Jain\n");
  code = GET_CODE(op);

  if(code == GT)
    return 1;
  else
    return 0;
```

```
}

/*********************************************************/
int cmp_dest_op(rtx op, enum machine_mode mode){
  rtx x0;
  rtx x1, x2;
  int retVal = FALSE;

  /* If the  mode of operand is not as required  and not equal to
     `VOIDmode' then return FALSE. */
  if(GET_MODE(op) != mode && mode != VOIDmode)
    retVal =  FALSE;

  /* If operand is an memory operand. */
  if(GET_CODE(op) == MEM){
    x0 = XEXP(op,0);
    switch(GET_CODE(x0)){

      /* If the  operand is an `PLUS' operand  and its first
         child is an `REG' and  second one is an `INT', then
         return TRUE. */

    case PLUS:
      printf("PLUS:DEST\n");
      x1 = XEXP(x0,0);
      x2 = XEXP(x0,1);
      if((GET_CODE(x2) == CONST_INT) && (GET_CODE(x1) == REG))
        retVal = TRUE;
      break;

      /* If  the destination is either `REG' or `SYMBOL' and
         contain the  address of some  memory location, then
         return TRUE. */
      /*               case SYMBOL_REF:*/
    case REG:
      retVal = TRUE;
      break;

    default:
      retVal = FALSE;
    }
  }
  /*  If the  destination is  either `REG'  or  `SYMBOL_REF' then
      return TRUE. */
  else{
    if(GET_CODE(op) == REG)
      /*       || GET_CODE(op) == SYMBOL_REF)*/
```

```
      retVal =  TRUE;
  }

  return retVal;
}

/**************************************************************/

/* This function takes care of source operand of a ‘move’
    operation. */

int cmp_src_op(rtx op,enum machine_mode mode){
  rtx x0;
  rtx x1,x2;
  int retVal= FALSE;

  /* If the mode of operand is not as required and not equal
     to ‘VOIDmode’ then return FALSE. */

  if(GET_MODE(op) != mode && mode != VOIDmode)
    retVal =  FALSE;

  if(GET_CODE(op) == CONST_INT){
    retVal = TRUE;
  }

  return retVal;
}
```

Appendix D Copyright

D.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this

License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.