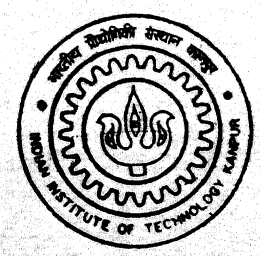
Generation of GCC Backend from Sim-nML Processor Description

By

SOUBHIK BHATTACHARYA



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Indian Institute of Technology Kanpur

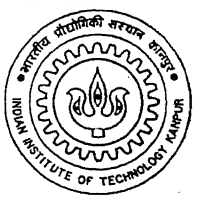
JULY, 2001

Generation of GCC Backend from Sim-nML Processor Description

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Soubhik Bhattacharya



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

July, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*Generation of GCC Backend from Sim-nML Processor Description*", by *Soubhik Bhattacharya*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

July, 2001

(Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

(Dr. Sanjeev Kumar Aggarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Increasing importance of software in *embedded systems* led to the paradigm of *hardware-software codesign*, which advocates for early integration of hardware and software, even before the hardware design is complete. To support this paradigm a set of tools are needed that can simulate the build and execution environment of hardware. The approach is developed in our group where a high-level specification of hardware is written and from which the tools assembler, linker, compiler, simulator, high-level synthesizer etc. are generated automatically.

In this thesis techniques have been developed for analyzing a high-level description of a processor, written in *Sim-nML* [17] processor specification language, and extracting the semantic information needed for automatic generation of GCC machine description. Using Sim-nML one can describe instructions of a processor in a compact hierarchical form. The hierarchy is initially flattened to obtain a sequence of C-like statements for each instruction. A sequence describes the semantic action of an instruction. The action sequences are simplified using the techniques of temporary removal and branch elimination and matched against some standard patterns so that they can be identified with one of the standard names used in GCC machine descriptions. Finally, this information is used to generate a partial GCC machine description for the processor.

Acknowledgements

It is my privilege to mention the names of Dr. Rajat Moona and Dr. Sanjeev Kumar in this page. Dr. Moona has driven this research with his enthusiasm and agility while Dr. Sanjeev Kumar's wise and experienced words helped me to avoid any possibility of diversion. Together they played the roles of mentors and teachers. I owe them a lot for what I have learnt through this research experience. I am thankful to Dr. Deepak Gupta for his constructive participation in our group discussions. I should also express my gratitude to the Department of CSE and its faculty and stuff for the beatiful academic environment that they have created.

This work is a part of an ongoing research at Cadence Research Center, IIT Kanpur. I am thankful to Cadence India Ltd. for their financial support.

I am grateful to my fellow members of CARES, Rajiv, Souvik, Prithvi, Arvind, Anand, and Mayank. They have made this group a hub of intellectual activities, stood by me at difficult times, shared frustrations, and at the same time, boosted my spirits. I should remember the past members of the group, Prashant and Sarika, who helped me during my early days. This, also, is an opportunity to bow my head before the great comradeship of MTech99. I hope this spirit will live long.

Finally, let me concede my huge debt to my parents and beloved ones. Without their support and patience this work would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

We are witnessing a time when electronic systems are being deployed in new and innovative ways across various aspects of our life and civilization, e.g., industrial automation, telecommunication, media, automobile, consumer electronics, to name a few. Use of programmable processors are no longer confined to general-purpose Personal Computers, servers, or multiprocessors. These processors are finding their ways to application specific electronic systems, better known as *embedded systems*. Use of Application Specific Instruction-set Processor (ASIP), Application Specific Integrated Circuit (ASIC), and general-purpose ISA-based processors, is also gaining popularity in the embedded systems. All these facts contribute in increasing the importance of software in embedded systems. At the same time an increasing number of vendors are trying to push embedded systems in various application areas. To quickly meet the demands of an expanding market and to obtain an edge over competitors, designer of embedded systems needs low turn-around time and cost effectiveness in the design. Electronic Design Automation (EDA) tools are used to meet these objectives. Existing EDA tools and methodologies, which facilitate design of hardware to a great extent, however do not provide significant aid in software development and hardware-software integration.

Normally hardware design and software development of an embedded system

begin nearly at the same time. However, they cannot be integrated until a prototype of the hardware can be built. *Hardware/software codesign* is a paradigm for designing embedded systems which advocates early integration of hardware and software in the design cycle, even before the hardware design is completed. This prevents errors from propagating through the design and reduces the effort spent in tracking and fixing them. This also allows the designer to evaluate performance of the system early and explore various design alternatives. To enable hardware/software codesign one needs a set of tools that can simulate the build and execution environment of the hardware. A common approach is to start with a high-level specification of the hardware, which contains enough information needed to develop software and execute it on that hardware. Tools are used to automatically generate compiler, assembler, linker from the high level specification to enable software development. Simulators are built around this specification to create an execution environment for the software. High-level synthesis tools are used to enable hardware design from this specification.

Sim-nML [17] is a high-level processor specification language, which is powerful enough to describe any ISA based processor. Tools have been developed to generate assembler [8], disassembler [7], function simulator [1], cache simulator [19] etc from Sim-nML specifications of processors. A preliminary work for generation of compiler from Sim-nML specifications has also been carried out [16].

In this work techniques have been developed for performing extensive semantic analysis of Sim-nML specifications and extracting information needed for generation of compiler. A tool has been developed that reads a Sim-nML specification in its intermediate form, and generates a partially complete GCC (GNU Compiler Collection) machine description. GCC has been retargeted to Sparc using the Sim-nML description. We have chosen GCC because it is a production quality optimizing compiler, which can be retargeted by writing a description of the target. However, GCC machine description is large and complex. Our tool reduces the effort needed to retarget GCC. The advantage is magnified by the fact that a Sim-nML specification can also be used to generate many other tools for the processor.

1.2 A Survey of Techniques for Compiler Backend Generation

A compiler translates a high-level language program to an equivalent assembly or machine language program [23] [24]. Broadly, it consists of two components. The *frontend* is responsible for lexical analysis, parsing and converting the program to an intermediate form. The *backend* or the *code generator* translates the intermediate form of the program to assembly or machine language. Ideally, compiler front end is specific to the source language and backend, to the target processor. This kind of design reduces the work needed to port an existing compiler to a new source language or target architecture.

Approaches for automatic generation of parts of the frontend from the specifications of the source language are well known [23]. Several attempts have been made to automate the generation of compiler backend from the specification of the target machine. We shall discuss some of them. At the end of this section, an overview of our approach will be given.

1.2.1 Grammar Based Approaches

Grammar based approaches attempt to extend the technique of parser generation to backend generation. A grammar for the intermediate form is specified. For each grammar rule an action is specified which constructs and/or emits assembly instructions as the rule is applied. A parser is generated from the grammar, which parses the intermediate form and generates assembly output.

Graham-Glanville [6] used a context free grammar to parse a Polish-Prefix intermediate form. A register allocator was meshed with the parser. Ganapathi-Fischer [5] used the more powerful notations of attribute grammars and disambiguating predicates. The code generators generated by them were capable of doing some simple optimizations also.

1.2.2 Approaches of Tree Pattern Matching

The approaches of tree pattern matching work on an intermediate form that is a sequence of trees. A set of tree-rewriting rules are specified. A rule has a tree pattern, which is matched within the intermediate form, a replacement node, which replaces the matched pattern, and an action to be performed on successful matching. A cost function is used to impose additional conditions for matching. Actions are responsible for emitting assembly code. Dynamic programming is used to determine an optimal cover for the intermediate form using the patterns.

Aho, Ganapathi and Tjiang developed a system called *twig* based on this approach [22]. LCC [14] also uses this approach. A program called *lburg* reads a machine specification file containing definitions of the tree rules and generates a code generator. In another work, a Reduced Instruction Set Machine (RISM), consisting of a set of simple instructions capable of simulating all other instructions, is automatically extracted from a tree-based machine description [2]. An RISM code generator is generated, which converts the intermediate form of the program, an abstract syntax tree, to a sequence of RISM instructions. An automatically generated optimizer then merges simple RISM instructions to more complex machine instructions and produces good quality assembly code.

1.2.3 Automatic Code Generation from High Level Processor Specification

A high level specification for a processor describes its ISA and additionally, provides some structural information. Unlike the grammar or tree rewriting rules, these specifications are tool independent. The source program is translated by a processor independent fronted to a suitable intermediate form, normally a CDFG (Control Data Flow Graph). The processor specification is converted to an internal data structure so that instructions can be represented by patterns which can be matched within the intermediate form of the program. Then attempts are made to cover the program optimally using instruction patterns. Basic steps performed by the retargetable code generator are *instruction selection, resource allocation,* and *instruction*

scheduling [25], [26], [27], [23].

CHESS [11] is a commercially available retargetable compiler based on nML [3] machine description formalism. CHESS has been designed for embedded fixed point DSPs (Digital Signal Processors). nML machine description is internally converted to an Instruction Set Graph, which stores information about instruction set and resources of the processor. Source program written in DFL or C is translated to a CDFG. Then the compiler backend performs code selection, register allocation and scheduling in sequence.

CodeSyn [15] compiler is a part of FlexWare [15] development environment for embedded systems. High-level program, written in C or C++, is translated to a CDFG. The code generator follows a rule-based approach. The machine description contains resource information (register sets, addressing modes etc) and a set of code selection rules, one for each high level operation. When the operation matches within the CDFG, the rule is triggered. The compiler performs global scheduling, register assignment, and code compaction in sequence.

AVIV [20] retargetable compiler focuses on processors exhibiting significant ILP (Instruction Level Parallelism) and VLIW architectures. It uses SUIF (Stanford University Intermediate Format) [28] and SPAM (Synopsys, Princeton, Aachen, MIT) [29] compilers as its frontend. The code generator reads ISDL [4] machine description and output of the frontend, which is a set of basic block DAGs (Directed Acyclic Graphs) connected through control flow information, and generates a Split-node DAG. A Split-node DAG represents a set of all possible ways the program can be executed on the processor. A heuristic branch and bound algorithm is used to produce near optimal assembly code from the Split-node DAG. Unlike most other approaches AVIV performs instruction selection, resource allocation and scheduling concurrently.

EXPRESSION [30] machine specification language describes ISA, some structural information and also the memory subsystem. Tools are used to automatically generate tree patterns describing instructions, a reservation table containing scheduling information etc. EXPRESS [30] retargetable compiler makes use of these information to generate code.

LISA [31] processor design platform includes a compiler generator. Along with LISA machine description, some additional semantic information and an ABI (Application Binary Interface) specification is provided to the compiler generator. The compiler generator then generates a machine description for LCC [14] which is built, along with LCC frontend, to obtain an LCC port for the processor.

The Mescal group is also working on a project to develop a retargetable compiler from MAD specification language [32]. However, this work is not complete and they are yet to report any result.

An earlier work [13] has been carried out to generate LCC [14] machine description from nML [3]. A tool has been developed that flattens an nML machine specification to obtain a set of instruction patterns. Additional transformations are applied to the instructions to synthesize an LCC machine description. The program lburg [14] then generates a backend for LCC from the synthesized machine description.

1.2.4 Automatic Code Generation from HDL

Descriptions written in HDL give a lower level view of the hardware than those written in high-level specification languages. HDL descriptions can easily accommodate architectural changes and they can be directly linked with hardware design tools. However, from the point of view of code generation, they contain unnecessary details about the hardware. The ISA, which acts as an interface between hardware and software, is not apparent in these descriptions. However, like the high-level specification languages, they are also tool independent.

A work has been carried out to extract ISA from an HDL description and generating compiler backend from these information. RECORD [27] retargetable compiler constructs a graph model, consisting of primitive processor entities and their interconnection, from an HDL description. From the graph a set of instruction templates is determined. With additional semantic knowledge of hardware operators, a tree grammar and a parser are generated. This parser works as a code selector in the compiler backend. RECORD compiler has reported to outperform (with respect to size of the generated code) native TI compiler on TMS320C25 DSP chip, when

tested with DSPstone [34] benchmark suite.

1.2.5 GCC Portable Compiler

GCC (GNU Compiler Collection) is a highly optimizing production quality compiler which has been ported to a number of processors. GCC has its own machine description format consisting of an md file, a number of C header files, and a C program file. The GCC frontend translates a source program into an intermediate form called RTL [21], which has a LISP-like recursive structure. The md file specifies a set of RTL templates and the ways to generate assembly instructions from them. Additionally, some of the templates in the md file are given standard names, which convey the semantics of the templates to GCC frontend. Frontend uses named templates to generate initial RTL intermediate form. The initial RTL form then undergoes a series of transformations for optimization, register allocation and scheduling, and then they are matched against templates defined in the md file and assembly code is generated.

GCC produces good quality code for processors with homogeneous structures. However, it is not very successful in the domain of ASIPs and DSPs, which often have heterogeneous register sets. Also, porting GCC to a new target often necessitates changes in the so-called 'machine independent' sources of GCC. So it is not retargetable, in the strictest sense of the term [25].

1.2.6 Our Approach

We propose to generate a GCC machine description from a Sim-nML specification of a processor, so that GCC can be ported to the processor with minimal effort. Sim-nML [17], which is an extension of nML [3] machine description formalism, is a tool independent high-level processor specification language. It captures information about the ISA, registers, addressing modes, functional units of a processor in a compact and easily maintainable form. Several tools have been developed to support software development and execution around the Sim-nML model of a processor [8] [7] [1] [9] [19]. In this work we have attempted to complement Sim-nML technology

by adding the compiler-generation capability to it.

A preliminary work for GCC machine description generation from Sim-nML has been carried out earlier [16]. A tool *genmd* has been developed which generated a GCC machine description for Intel8085 processor, sans control transfer instructions. However, the techniques used in this tool avoided many practical complexities and so the tool failed to work with more complex processor descriptions. Also it did not have the appropriate framework for dealing with control transfer instructions. Nevertheless, this work gave us some insight into the problem.

Figure 1.1 outlines our approach. The tool *irg* parses the Sim-nML description and stores it into a file called IR (Intermediate Representation). This IR is input of our tool. Since the description is written in a compact hierarchical form, it is initially flattened to obtain a sequence of C-like statements for each instruction. This sequence of statements describes the semantics of the instruction. Some simplifying transformations are made to remove temporaries, fold constants, eliminate branches etc. from the sequences. Then sequences are matched against some predefined patterns and identified with standard GCC names. At this stage we have gathered enough information for generation of a GCC machine description. Now it is possible to write the GCC machine description of a processor in a number of ways. We have implemented a simple and generic machine description generation strategy and generated a partial machine description. Finally, additional information are added to complete the machine description and a GCC is built for the target processor.

The advantage of this approach is that we are using a well-trusted frontend and high quality optimization and code generation techniques of GCC. But the difficulty arises because of a lack of simple formal structure in GCC machine description. Also, semantics of instruction patterns are to be conveyed to GCC explicitly by using standard GCC names. This necessitates a rigorous semantic analysis of the Sim-nML description.

1.3 Outline of the Thesis

In **chapter 2** we present an overview of the Sim-nML language.

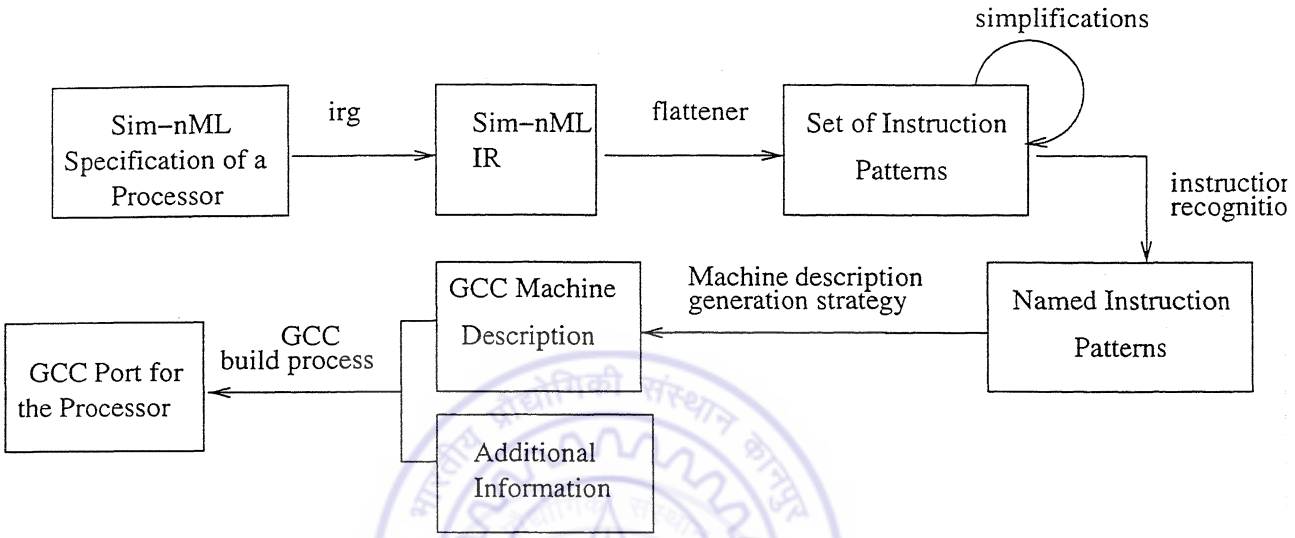


Figure 1.1: Outline of Our Approach

In **chapter 3** we discuss GCC and the mechanism to port GCC to a target.

In **chapter 4** we discuss the design and implementation of the tool that is developed in this thesis to generate GCC machine description from Sim-nML. Finally, we present the results of our work and some future directions in **chapter 5**.

Chapter 2

Sim-nML Processor Specification Language

In this chapter the formal structure of Sim-nML language, along with an overview of its syntax and semantics will be presented. To exemplify the expressibility of the language, Sim-nML description of UltraSparcIIi processor will be discussed. Detailed description of the language can be found in [17] [9] [10] [18].

2.1 Sim-nML Language

2.1.1 General Characteristics

Sim-nML processor specification language is an extension of nML [3]. It has the following general characteristics:

- **High Level of Abstraction:** A Sim-nML programmer views a processor as a machine that executes a set of instructions. For each instruction in the instruction set of the processor, the binary image, assembly syntax, functionality, resource usage and timing are specified. Additionally, the ISA-specified registers, memory and functional units are described. The description contains enough information about the processor to support software development and execution around it.

- **Tool Independence:** A Sim-nML description is not specific to a tool. A range of tools, including assembler [8], disassembler [7], function simulator [7], cache simulator [19] have been generated from it.

- **Generality:** Sim-nML language is powerful enough to capture any kind of ISA-based processor. Sim-nML descriptions have been written for different classes of processors including RISC (Sparc, Mips, PowerPC, ARM), CISC (M68HC11, 8085), and DSP (ADSP) [33].

- **Compactness:** Sim-nML allows the programmer to write a compact and hierarchical description by exploiting the commonality between instructions.

2.1.2 Basic Data Types

Sim-nML provides a set of built-in abstract types viz. *card, int, float, range, bool* etc. A near orthogonal set of operators is also provided. All the types used in the description are defined by the programmer and derived from the built-in types. For example, `card(32)` is a 32-bit unsigned integer type derived from the built-in type *card.*

Basic data types are used in the following two different contexts

- To describe data types supported by the processor, e.g. to describe types of registers, memories, or parameters of instructions.

- To describe data types needed for programming, e.g. to describe types of temporary variables used within the description.

In addition string literals are allowed in the definitions of assembly syntax and binary images of instructions.

2.1.3 Storage and Functional Units

The keyword *resource* is used to define a function unit or a pipeline stage. A storage unit may be a processor resource or a temporary variable needed for programming. The keyword *reg* declares an ISA-specified register. The keyword *mem* may declare

a memory unit or a temporary variable. The keyword *var* is specifically used to declare the temporary variable.

2.1.4 Instruction Set

In Sim-nML, instruction set of the processor is described as an S-attributed grammar. Each sentence derivable from the grammar corresponds to a single machine instruction. Each nonterminal symbol of the grammar is associated with a set of attributes. Each production rule of the grammar is associated with a set of attribute definitions. Each attribute definition computes the value of an attribute of the left-hand side of the production as a function of the values of the attributes of the symbols appearing on the right-hand side.

A set of productions of the form:

```
T : X
T : Y
.....
T : Z
```

where T, X, Y, ..., Z are nonterminal symbols, is represented in Sim-nML with an *or* rule of the form:

```
op T = X | Y | .....  | Z
```

The attribute definitions are implicit in an *or* rule. All the attributes of the right-hand symbol are assigned to the corresponding attributes of the left-hand symbol when a production of this form is applied.

A single production of the form:

```
T : X Y ....  Z
```

where T is a nonterminal and X, Y, ..., Z are terminal or nonterminal grammar symbols, is represented in Sim-nML with an *and* rule of the form:

```
op T (X, Y, ..., Z)
```

X, Y, ..., Z are called parameters of the *and* rule. If a parameter is of a basic type then it is treated as a terminal symbol of the grammar, which is a parameter of a machine instruction. Otherwise, the parameter is treated as a nonterminal symbol, which is a partial definition of a machine instruction. Each nonterminal symbol

should appear on the left-hand side of *exactly* one Sim-nML rule. In an *and* rule attributes are explicitly defined. If an attribute is not defined then it is assumed to have a *null* value.

Sim-nML provides a set of attributes with a predefined semantics. When an instruction is derived from the grammar, complete definitions of all the attributes are obtained. The attribute *syntax* stores the syntax of the instruction. Likewise, the attributes *image, action,* and *uses* store, respectively, the binary image, functionality, and resource-usage of the instruction.

A Sim-nML rule whose left-hand side is an addressing mode or a partial definition of an addressing mode, is called a *mode* rule. All other rules are called *op* rules. A *mode* rule differs from an *op* rule because it can have a value. The value of a *mode* rule is stored in a hidden attribute.

It is intuitively obvious that the S-attributed grammar supported by Sim-nML can be used to describe any context free grammar and hence, any instruction set.

2.1.5 Attribute Types

The attributes for assembly syntax and binary image are strings. The attribute for resource-usage follow a usage grammar. Attributes defining functionality of instructions are sequences of C-like statements, often called *action sequences.*

Sim-nML provides a restricted programming model to define functionality of instructions. It supports built-in and user defined types, built-in operators, sequence of statements, control transfer, function call etc. It also allows programmer to define variables, which are called temporaries, and have a global scope and infinite lifetime.

To support control transfer Sim-nML has an *if-then-else-endif* statement. However, there is no construct for loops and goto-like jumps.

Level of abstraction of an action sequence is lower than that of a machine instruction because an action sequence is used to express functionality of an instruction. It gives programmer a lot of freedom through bit-selection and concatenation operators. Bit-selection allows a programmer to view an arbitrary chunk of bits of a storage unit as a single object. Concatenation allows programmer to form an object

by combining a number of objects. In Sim-nML language, the smallest unit of storage that can be viewed as an object is a single bit. However, these low-level features make application of traditional algorithms for data flow analysis, copy propagation etc. difficult, as will be observed in chapter 4.

2.2 An Example: Sim-nML Description of UltraSparcIIi Processor

UltraSparcIIi is a 64-bit superscalar RISC processor [36] that implements Sparc V9 [35] ISA. In this section we will discuss the ways in which some of its interesting features have been expressed using Sim-nML.

2.2.1 Windowed Register Set

Sparc V9 supports the notion of a windowed register set. The mapping between a register number generated by software and actual hard register number depends upon the state of a special register, called *window pointer register*. Software generated register numbers are partitioned into four classes viz., *global, out, local,* and *in*, each containing eight registers.

Following formulas show the relationship between software register numbers and hardware register numbers:

```
hard_reg_no = global_reg_no + pstate.ag * 8
hard_reg_no = out_reg_no + cwp * 16
hard_reg_no = local_reg_no + cwp * 16 + 8
hard_reg_no = in_reg_no + cwp * 16 + 16
```

Here pstate.ag is a single bit in a state register *pstate*. *cwp* is 5-bit current window pointer register. Following is a Sim-nML mode rule defining an addressing mode for local registers:

```
mode loc(x:card(3))=winreg[16*cwp + 8 + x]
syntax=format("%%l%d",x)
image=format("%5b",x+16)
```

'winreg' has earlier been declared to be a register file of 128 registers. Note that the index of the register file is a function of a state register.

2.2.2 Delayed Transfer of Control

In Sparc V9 all the control transfer instructions (call, jump, branches) are delayed. The delayed semantics has been expressed by introducing a next-PC register, along with the normal PC. In the description *pc* refers to the normal PC and *npc*, to the next PC register. All the non control transfer instructions execute the following pair of statements

```
pc = npc;
npc = npc + 4;
```

On the other hand a call instruction, which unconditionally transfer control to a PC-relative target, executes the following pair of statements

```
tmpc = pc;
pc = npc;
npc = tmpc + 4*coerce(sxword, label);
```

Here *tmpc* is a temporary, *label* is a parameter, which specifies the target. *coerce* operator sign-extends *label* to a signed 64-bit integer. The target is multiplied by 4 to maintain alignment.

2.2.3 Branches

In Sparc V9 there are 5 classes of branch instructions viz., *bpr, fbfcc, fbpfcc, bicc,* and *bpcc.* A class contains 24, 32 or 64 branch instructions. For example there are six variations of *bpr* (branch on integer register condition with predictions)

- Branch if zero
- Branch if nonzero
- Branch if less than zero
- Branch if less than equal to zero

- Branch if greater than zero
- Branch if greater than equal to zero

Each of these branches can be annulling or non annulling, and predict-taken or predict-not-taken. So there are total 24 branches in class *bpr*. Such a large number of branches have been described in Sim-nML in a very compact manner by introducing dummy *mode* rules. A *mode* rule normally corresponds to an operand of the instruction. A dummy *mode* rule, however, represents a constituent of the opcode. In this description dummy *mode* rules have been used to specify the register condition to be evaluated, the annul bit and the prediction bit of a branch instruction. Here is an example from the UltraSparcIIi description:

```
//'rcond' field in 'bpr'
//if equal to zero
mode rz()=1
    syntax="z"
    image="001"
//if less than equal to zero
mode rlez()=2
    syntax="lez"
    image="010"
//if less than zero
mode rlz()=3
    syntax="lz"
    image="011"
//if not zero
mode rnz()=5
    syntax="nz"
    image="101"
//if greater than zero
mode rgz()=6
    syntax="gz"
```

```
    image="110"
//if greater than equal to zero
mode rgez()=7
    syntax="gez"
    image="111"
//one of the above conditions
mode rcond= rz | rlez | rlz | rnz | rgz | rgez
```

Now instead of using 24 different *op* rules, all the branches of class *bpr* have been described using a single *op* rule, which has a parameter of type *rcond*. Within the *action*, value of this parameter is checked and register condition is evaluated accordingly. The *action* of this *op* rule sets a temporary, *taken*, which is examined by another higher-level *op* rule and *pc* and *npc* are adjusted accordingly. Following is the example of the *op* rule *bpr* (*a* and *p* are two dummy *mode* rules specifying annul and prediction bits, *disp16* is a *mode* rule for a 16-bit displacement, *gpr* is a *mode* rule for a general purpose register):

```
op bpr(x:a, y:rcond, z:disp16, w:p, u:gpr)
syntax=format("br%s%s%s %s, %s", y.syntax, x.syntax, w.syntax, u.syntax,
z.syntax)
image=format("00%s0%s011%s%s%s%s", x.image, y.image, z.image<14..15>,
w.image, u.image, z.image<0..13>)
action={
    annul=x; //save annul bit in a temporary
    ea=pc + 4*coerce(sxword,z); //save target address in a temporary
    //evaluate register condition
    if y==1 //equal to zero
    then
      if u==0
      then
        taken=1;
      else
        taken=0;
```

```
  endif;
else if y==2 //less than equal to zero
then
  if coerce(sxword, u) <= 0
  then
    taken=1;
  else
    taken=0;
  endif;
else if y==3 //less than zero
then
  if coerce(sxword, u) < 0
  then
    taken=1;
  else
    taken=0;
  endif;
else if y==5 //not zero
then
  if u!=0
  then
    taken=1;
  else
    taken=0;
  endif;
else if y==6 //greater than zero
then
  if coerce(sxword, u) > 0
  then
    taken=1;
  else
```

```
        taken=0;
      endif;
    else if y==7 //greater than equal to zero
    then
      if coerce(sxword, u) >= 0
      then
        taken=1;
      else
        taken=0;
      endif;
    endif;
    endif;
    endif;
    endif;
    endif;
    endif;
}
```

Chapter 3

GCC and its Porting Mechanism

GCC is an open-source compiler, developed by the GNU community [37]. It is available for a number of frontends, including C, C++, Fortran, Java, Objective-C, and a number of processors including Intel x86, Sparc, Mips, Arm, Motorola 68HC11. It is known to be a production quality compiler with high quality optimization and code generation techniques. GCC can be ported to a new target by providing a target description in the form of an md file, a number of C header files, and a C program file. GCC frontend produces code in an intermediate form, known as an RTL representation. RTL patterns also appear in the md file of GCC machine description.

In this chapter an introduction to GCC RTL representation will be given. Also, GCC's internal representation of a program, and machine description will be discussed in brief. Finally, we shall present an overall picture of the translation procedure. All the four topics, among others, have been discussed at length in [21].

3.1 RTL Representation Basics

RTL has a LISP like recursive structure. An RTL object is the most fundamental abstraction of an RTL representation. An RTL object can represent an operator, an operand, side-effect (functionality) of an instruction, an instruction, a definition of an instruction etc. An RTL object is one of the following

- integer: C type `int`
- wide integer: C type `HOST_WIDE_INT`, as defined in GCC's source files
- string: C type `char *`
- expression of RTL objects: a pointer to a structure
- vector of RTL expressions: an arbitrary number of RTL expressions

3.1.1 RTL Expressions

Internally (within GCC sources) an RTL expression, also called an RTX, is a pointer to a structure. Operands of the expression are members of the structure, which in turn, are RTL objects. The structure also has a member called code of the RTX. The code gives the expression a name and a semantic meaning, and defines the number and types of its operands. A list of RTX codes can be found in Appendix A. Another member of the structure is machine mode of the RTX. Machine mode defines the type and width of the value produced by the RTX.

An RTX has an external representation, which appears in debugging dumps and md files. In this form, an RTX is enclosed within a pair of parentheses. Name of the RTX appears first, followed by the machine mode and operands. Absence of machine mode implies VOIDmode. Some of the examples of RTXes are as follows:

An RTX representing register number 10 is written as `(reg:SI 10)` where *reg* is the name of the RTX. *SI* stands for single integer machine mode. A *reg* RTX has only one operand, which is an integer RTL object. The operand signifies the register number.

An RTX representing a constant integer 5 is written as `(const_int 5)`. A *const_ int* RTX does not have a machine mode (or is equivalent to a machine mode VOIDmode). The only operand of this RTX signifies the value of the integer.

An RTX representing result of the addition of register number 10 and a constant integer 5 is written as `(plus:SI (reg:SI 10) (const_int 5) )`. *plus* RTX has two operands, both of which are RTXes. They signify the operands of an addition. The machine mode of *plus* specifies the type and width of the result of addition.

3.2 Internal Representation of a Program

An *insn* is an RTX, which is GCC's abstraction of an instruction. GCC frontend translates a compilation unit into a doubly linked chain of *insns*. Translation is performed on a statement-by-statement basis during parsing. At the time of assembly output generation, an *insn* is typically converted into a sequence of one or more assembly instructions. Some *insns*, however, are not real instructions, and represent labels or some declarative information.

Following are the RTX codes that an *insn* can have

insn, jump_insn, call_insn, note, barrier, and *code_label.*

An *insn* has an operand of type RTX which defines its functionality or 'side-effect'. A side effect typically performs an arithmetic/logic operation and stores the result to a register, or moves between registers, or moves between a register and a memory location, or sets PC conditionally to a target etc.

Following is an example of an *insn,* that adds a register and an immediate constant, and stores the result into another register:

```
(insn
  10 7 11
  (set (reg:SI 9) (plus:SI (reg:SI 10) (const_int 5)) )
  -1 (nil) (nil)
)
```

The side-effect of the *insn* is a *set* RTX, whose first operand signifies the destination of the assignment and the second operand, source. *set* does not produce a value and so, does not have a machine mode.

Three numbers preceding the side-effect expression represent, in order, uid (unique identity) of the *insn,* uid of the previous *insn,* and uid of the next *insn.* Others operands of an *insn* are not important in this context.

3.3 Machine Description

GCC machine description contains the following information:

- Processor architecture– functional behavior, and optionally, resource-usage of instructions, endianness, memory addressability etc.
- ABI– register usage, function-calling conventions etc.
- Layout of source language data types– sizes of `int`, `float`, `char` etc.
- Format of binary files– format of object and executable files, format of debugging information.
- Compiler environment– conventions for assembler, linker, libraries, location of system's headers and libraries etc.

The machine description consists of an md file, a C program file and a number of C header files.

3.3.1 md File

An md file can contain the following information:

- Definitions of RTL patterns which can appear as side-effects of *insns*. The RTXes *define_expand* and *define_insn* are used to provide these information.
- Ways to generate assembly instructions from *insns*. The RTXes *define_insn* and *define_peephole* are used for this purpose.
- Ways to split a single *insn* into a sequence of *insns*. The RTX *define_split* is used for this purpose.
- Information about function units and latencies of instructions. The RTXes *define_delay* and *define_function_unit* are used for this purpose.

Names of Patterns

Names are given to RTL patterns defined using *define_expand* and optionally, to those defined using *define_insn*. Two different definitions cannot use same name. GCC provides a set of standard names which convey the semantics of the patterns to

GCC frontend. Standard names are used while translating the high-level language program into an RTL intermediate form. GCC generates a *gen_name* function to generate a pattern whose name is *name* and *name* does not begin with the character '*'. A *gen_name* function accepts the operands of the pattern as arguments. These functions are often used in a machine description to explicitly generate a pattern. A list of standard names can be found in Appendix A.

Templates Used in a Pattern

RTL templates are used to specify a set of operands or operators that can appear in a particular position in the pattern. The following RTXes are used as RTL templates

match_operand, match_dup, match_operator etc.

match_operand specifies a set of operands. It has three operands. First operand is an integer RTL object, specifying operand number. Second and third operands are string RTL objects, which specify a *predicate* and a set of *constraints*, respectively. A predicate specifies a broad class for the operand. e.g. whether it is a register or an immediate operand. Constraint imposes stricter conditions e.g. the exact class of registers or range of immediate operands. Two patterns which differ only in the constraints of their templates cannot be defined separately. A single definition is used for them with a set of alternative constraints.

Here is an example of a *match_operand* template

```
(match_operand:SI 0 "register_operand" "a")
```

In an actual *insn* this template will be replaced by the operand number 0, which must be a register of class 'a', with machine mode SImode. There are a number of built-in predicates provided by GCC. Additional predicates can be defined in the C program file. A list of built-in predicates has been provided in Appendix A. Meanings of constraint letters are specified in a C header file within the machine description.

define_expand Patterns

*define_expand*s are only used during RTL generation i.e. during translation of the

high-level program into RTL. They must have a name. *define_expand* allows generating a sequence of RTL patterns, each to appear as a side-effect of an *insn* in a sequence of *insns*. Every pattern that may be generated by a *define_expand* should also be defined using a *define_insn*. Using *define_expand* one can also specify a fragment of a C code to be executed before the generation of the patterns. The constraints which appear in an RTL template of a pattern are ignored by *define_expand*.

Here is an example of a *define_expand*

```
(define_expand "addsi3"
(set
  (match_operand:SI 0 "general_operand" "")
  (plus:SI
    (match_operand:SI 1 "general_operand" "")
    (match_operand:SI 2 "general_operand" "")
  )
)
""
""
)
```

First operand of *define_expand*, "addsi3", is the name of the pattern. "addsi3" is a standard GCC name, meaning addition in single integer mode. Second operand is the pattern. Note that *match_operand* templates have occupied the places of real operands. The predicate "general_operand" allows any general register, memory or immediate constant as an operand. Third operand is a string, which is supposed to specify a condition to be tested before this *define_expand* is used. Last operand is also a string where one can put a fragment of C code.

define_insn Patterns

define_insn may or may not have a name. Named *define_insn*s may be used during RTL generation. *define_insn*s are also used at later stages of compilation. Using *define_insn* one can specify a single RTL pattern which can appear as a side-effect of an *insn*. Using *define_insn* one also specifies the assembly code to be generated

from the pattern or a fragment of a C program to be executed to generate the assembly code.

Here is an example of an unnamed *define_insn*, which defines a pattern that may be resulted from the *define_expand* shown in the last example:

```
(define_insn ""
(set
  (match_operand:SI 0 "register_operand" "a")
  (plus:SI
    (match_operand:SI 1 "register_operand" "a")
    (match_operand:SI 2 "register_operand" "a")
  )
)
""
"add %1, %2, %0"
)
```

The first operand, which specifies the name, is an empty string. Second operand is the pattern. Third operand specifies a condition which must be true when this pattern is used. Last operand specifies an assembly instruction that will be generated from this pattern.

- *define_peephole*

define_peephole is used to define machine specific peephole optimizations. GCC uses a *define_peephole* optionally, only if optimizations are enabled. It allows one to specify a sequence of patterns and an assembly code to be emitted for the sequence.

3.3.2 C Header and Program Files

C header files and the C program file contain all the information needed by GCC which cannot be represented properly within an md file. The header files define a number of macros and enum types, and declare some global variables and routines. The program file defines some global variables and routines. These files also contain

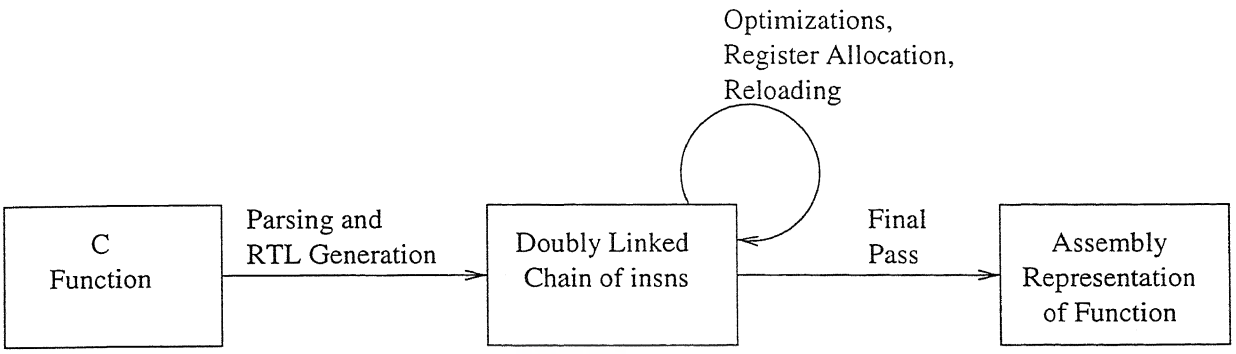


Figure 3.1: Translation Process of GCC

support information, viz. macros, variables, and routines, which are used elsewhere in the machine description. A list of some useful macros can be found in Appendix A.

3.4 The Translation Process

Figure 3.1 outlines the way GCC translates a C function into an assembly code. Steps are discussed below.

3.4.1 Parsing and RTL Generation

The C function is parsed and *insns* are generated on a statement-by-statement basis. At this stage GCC looks the md file for a *define_ expand* or a *define_ insn* with some standard name. If found, it checks the condition (third operand of *define_ expand* or *define_ insn*) and the predicates of the templates used within the pattern. If all of them are satisfied, then an *insn* or a sequence of *insns* is generated to express an operation of the high-level language.

3.4.2 Optimization, Register Allocation, Reloading

In this phase GCC performs several optimizations e.g. jump optimizations, loop optimizations, scheduling. It also performs register allocation. Following things can

happen in this phase:

- **Deletion:** An *insn* may be deleted.
- **Matching:** An *insn* may be matched against a *define_insn* pattern. During matching predicates of the templates are checked, but constraints are not. Matching helps in assembly code generation.
- **Scheduling:** An *insn* may be matched against a *define_delay* or *define_function_unit*. This matching helps in delay slot scheduling and instruction scheduling.
- **Combination:** A sequence of *insns* may be combined to form a single, more complex *insn*. Resulting complex *insn* should, of course, match a *define_insn* pattern. This helps in machine independent peephole optimization.
- **Splitting:** An *insn* may be matched against a *define_split* and split into a sequence of simpler *insns*. Each simpler *insn* should match a *define_insn* pattern. Splitting is needed if a complex *insn* formed by *insn* combination does not match any *define_insn*. Splitting also helps in delay slot scheduling and instruction scheduling.
- **Construction:** A new *insn* may be constructed and added to the doubly linked list.
- **Reload:** An *insn*, that does match a *define_insn*, may be invalidated because constraints may not be satisfied. So GCC generates extra move *insns* to ensure that constraints are satisfied.

3.4.3 Final Pass

At this stage GCC performs machine specific peephole optimizations, generates assembly code for a function, generates function entry and exit code.

If a sequence of *insns* matches a *define_peephole* then the sequence is replaced by the corresponding assembly code. Otherwise, assembly instruction for an *insn* is generated from the matching *define_insn* pattern.

Chapter 4

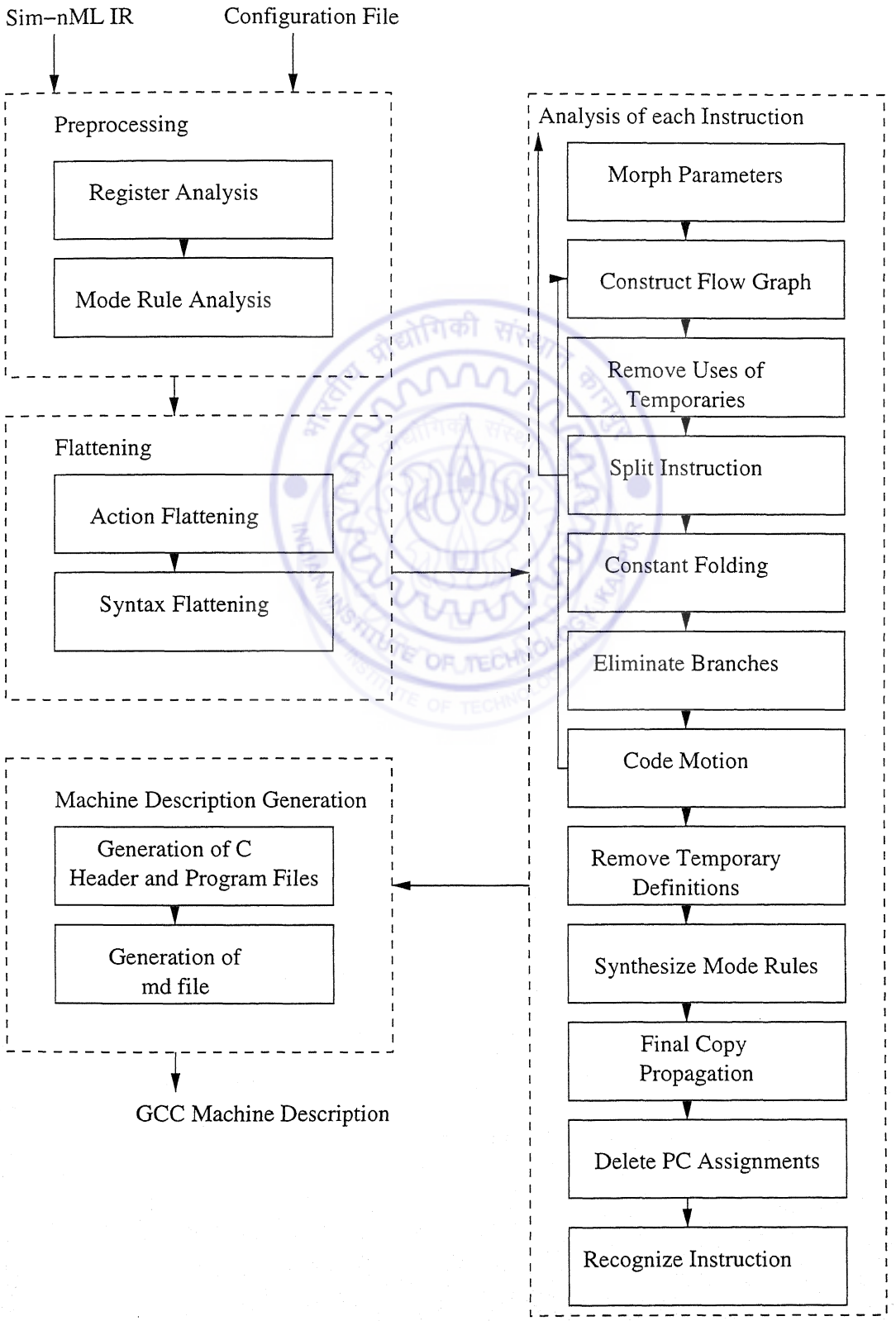
Generation of GCC Machine Description from Sim-nML Specification

In this chapter we shall discuss the techniques for automatic generation of a partial GCC machine description from the Sim-nML specification. We have implemented these techniques in the form of a tool.

In chapter 1 an outline of our approach was presented. A more detailed block diagram can be found in figure 4.1. GCC machine description generator reads a Sim-nML IR and a configuration file and generates the files *target.md, target.h,* and *target.c.* In the rest of this chapter individual components of the tool will be discussed.

4.1 Preprocessing

At preprocessing stage the machine description generator parses its arguments, reads the configuration file and the Sim-nML IR. Sim-nML IR is an intermediate representation of a Sim-nML specification, generated by the tool *irg*[18]. Configuration file contains additional information about the processor, as needed by the tool. This


Sim–nML IR
Configuration File
Preprocessing
Register Analysis
Mode Rule Analysis
Flattening
Action Flattening
Syntax Flattening
Machine Description Generation
Generation of C Header and Program Files
Generation of md file
GCC Machine Description
Analysis of each Instruction
Morph Parameters
Construct Flow Graph
Remove Uses of Temporaries
Split Instruction
Constant Folding
Eliminate Branches
Code Motion
Remove Temporary Definitions
Synthesize Mode Rules
Final Copy Propagation
Delete PC Assignments
Recognize Instruction

information includes the name of the PC and other PC-class registers (e.g. next-PC), name of the stack pointer, return address pointer, condition code registers etc. The structure of a configuration file has been described in Appendix C.

After reading the inputs the tool performs some analyses to gather information about the registers and addressing modes of the processor.

4.1.1 Register Analysis

At register analysis phase a map of all the register of the processor, as described in the Sim-nML specification, is created. Also special registers, as named in the configuration file, are identified.

4.1.2 Mode Rule Analysis

A Sim-nML mode rule typically describes an addressing mode of the processor. Mode rules are used to define parameter types of instruction actions. Mode rule analysis is performed to gather the following information about each mode rule

- If the mode rule is an *or* rule then it is viewed as a set of alternative addressing modes. Total number of alternatives and each alternative are determined. A child of a mode *or* rule may be a mode *and* rule or a mode *or* rule. In the former case, the child represents a single alternative. In the later, the child itself is a set of alternatives and so, alternatives represented by it are similarly determined.

- If the mode rule is an *and* rule and it's value is an *if-then-else-endif* expression then also it is viewed as a set of alternative addressing modes and total number of alternatives and each alternative are similarly determined.

- If a mode *and* rule represents a single addressing mode then it's value expression is analyzed and the predicate code, machine mode and constraint are determined.

  Value expression of a mode *and* rule is a prefix expression with arithmetic-logic operators, index, concatenation, bit-selection etc., and whose operands may be

parameters of the mode *and* rule (basic type or another mode rule), immediate constants, registers or memories. A grammar for the value expressions that is recognized by our tool is given in Appendix B. The recursive algorithm for evaluation of prefix expressions has been used to analyze value expressions.

A predicate code specifies the broad class in which an operand of an instruction belongs to and is one of the *register code, immediate code, memory code,* and *operator code. Operator code* is used to deal with complex mode rules where the value expression contain arithmetic-logic operators. Analyses of this kind of mode rules yield additional *secondary* mode rules which are operands of the original mode rule.

A machine mode specifies the type (integer or floating point) and width of the operand.

For a register operand the constraint refers to a class of registers, which is a subset of the register map created during register analysis. For an immediate operand the constraint specifies a range of immediate values while for a memory operand, the constraint specifies the addressing mode.

Additionally, mode rule analysis determines registers which can be used as base registers, index registers, window pointer registers etc. and a range of numbers which can be used as displacements.

To some extent, mode rule analysis can check for semantic validity of the Sim-nML specification as well. For example, it can be checked by this analysis whether an index used with a register is within the range or not.

4.2 Flattening of Action Sequence

A Sim-nML specification of a processor contains a compact hierarchical description of the instructions of the processor. The hierarchy of *op* rules is assumed to be a tree. A path from the root of the tree to a leaf is viewed as a single instruction, which can have parameters of basic and mode rule types. By flattening we mean traversing all such paths to obtain definitions of a particular attribute of all the

instructions. In the context of the current work only *syntax* and *action* attributes are relevant. A recursive algorithm has been used to traverse the tree and obtain complete syntax string, action sequence and parameter list of each instruction. The algorithm has been presented by [16].

4.3 Instruction Analysis

The flattened action sequence of an instruction may be very complex with the presence of temporaries and spurious branches. Therefore, it is difficult to recognize the instruction from this description. Simplifying transformations, which constitute the heart of GCC machine description generator, convert the action sequence to a set of simple parallel statements. This simpler set can be matched against a small set of simple patterns and the instruction can be recognized. Following subsections describe the simplifying transformations and instruction recognition.

4.3.1 Morphing Parameters

A parameter reference that appears in a statement within the flattened action sequence is the parameter number as assigned in the *op and* rule from which the statement is resulted. However, for validity of the following transformations a uniform numbering scheme for parameters is needed. So all parameter references are replaced by the parameter numbers as assigned in the parameter list of the instruction.

4.3.2 Construction of Control Flow Graph

The basic blocks[1] and control flow arcs connecting the basic blocks are determined. Since the action sequence does not have any loop or unconditional goto, the resulting control flow graph is a Directed Acyclic Graph (DAG). This graph contains information necessary for removal of uses of temporary variables and copy propagation.

[1] A sequence of statements with a single entry and a single exit point

4.3.3 Removal of Uses of Temporary variables

A temporary variable represents a state that is not specified in the ISA but is used in the Sim-nML specification to simplify the description. Temporary variables are usually defined using the *var* keyword in the Sim-nML description. All temporary variables should be removed from the action sequence before the instruction can be recognized.

To remove a use of a temporary variable, it is replaced by its most recent definition. However, if a state appearing on the right hand side of the definition changes before the use of the temporary variable then this transformation cannot be applied. In such a case we view the resulting statements as parallel statements i.e. all the states are read before any one of them is written [16].

Consider the following example of sequential statements

```
tmpc = pc;
pc = npc;
npc = tmpc+4;
```

Above sequence of statements updates PC and NPC registers within the action sequence of a non control transfer instruction of UltraSparc processor with delayed transfer of control. The first statement defines the temporary variable *tmpc*, the second statement writes to the *pc* register, which appears on the right hand side of the definition of temporary variable, and third statement uses *tmpc* to define *npc* register. After transformation these statements can be written as the following parallel statements:

$\mathtt{pc}_{\mathtt{out}} = \mathtt{npc}_{\mathtt{in}};$

$\mathtt{npc}_{\mathtt{out}} = \mathtt{pc}_{\mathtt{in}} + 4;$

There are some additional complexities present in this phase because of the possible uses of bit-selection and concatenation operators in the action sequence. This forces one to keep track of virtually every single bit of every temporary variable.

Following is a pair of definitions of temporary variable *temp* in presence of bit-selection, where the later nullifies the effect of the former:

```
temp<2..8> = ...;
temp<4..12> = ...;
```

The following is another example where it is not possible to say whether the first statement is nullified or whether the effect of the first statement is visible after the second statement.

```
temp1::temp2 = ...;
temp2 = ...;
```

In such a case we have chosen to ignore the instruction.

There may be situations where it may be very difficult to determine the most recent definition of a temporary variable. Here is an example:

```
temp<4..12> = ...;
x = temp<3..7>;
```

Here is another similar example:

```
temp1::temp2 = ...;
x = temp1;
```

In such scenarios we have chosen to ignore the instruction.

4.3.4 Instruction Splitting

If an instruction's behavior depends upon the value of a parameter that is not a part of the processor state, then we view it as a compact representation of a number of alternative instructions. Such an instruction is split into a number of alternative instructions and the original instruction is ignored.

In particular, if an action sequence has an *if-then-else-endif* statement which tests the value of a parameter of immediate type, then only one of the two paths will be executed and the exact path of execution can be known at the time when compiler generates this instruction. Such an instruction is split into two instructions, one for each of the two paths of execution.

Here is an example taken from the *action* of Store Byte Update instruction of PowerPC 603

```
if ra == 0 then
  EA = d;
else
  EA = GPR[ra]+d;
```

```
endif;
```

The above statements compute an effective address, which is a sum of a register and a displacement, *d*. *GPR* is a general-purpose register file and *ra* is an immediate type parameter of this instruction which indexes this register file. However, *GPR[0]* is hardwired to 0 and should always read as 0. So the effective address, *EA*, has two definitions, depending upon whether *ra* is 0 or nonzero. Splitting yields two instructions, in one of which *ra* is always 0 and effective address is computed by the following single statement

EA = d;

In the other instruction *ra* is always nonzero and effective address is computed by the statement

EA = GPR[ra]+d;

4.3.5 Constant Folding

Removal of the uses of temporary variables often creates constant expressions. We evaluate constant expressions and replace them by their values.

4.3.6 Branch Elimination

The *if-then-else-endif* statements whose outcomes are known (as a result of constant folding) are eliminated. All the statements in the 'false' path are deleted.

4.3.7 Code Motion

The statements which follow an *if-then-else-endif* statement are moved to the end of the *then* and *else* branches.

After this step it is checked whether there are any uses of temporary variables left in the action sequence. In such a case the control flow graph is constructed once again and the above steps are iterated.

4.3.8 Removal of Definitions of Temporary Variables

Once all the uses of temporary variables are removed from the action sequence the statements which define temporary variables are also deleted. From this point onwards the action sequence is free of all the temporary variables.

4.3.9 Mode Rule Synthesis

In general, an operand of an instruction is a function of the parameters of the instruction. For example, an instruction may access a register operand indexed by an immediate type parameter, as shown in the following code

```
GPR[i] = GPR[j] + GPR[k];
```

Here *GPR* is a general-purpose register file, indexed using parameters i, j and k, of immediate type.

We synthesize new mode rules for such operand functions. This way a uniform representation of operands is used that helps in final copy propagation, as well as, in instruction recognition.

4.3.10 Final Copy Propagation

In this phase all the uses of register and memory operands are replaced by their most recent definitions. A copy propagation involving temporary variables was performed during removal of uses of temporary variables, which necessitated us to view the action sequence as a set of parallel statements. Final copy propagation is needed to truly convert the action sequence to a set of parallel statements.

It is important to note that a single register or memory operand is actually a mode rule and can represent a set of states. Therefore following are the possible scenarios

- Definition of a variable $\mathbf{x}$ is live at a statement that uses $\mathbf{x}$. The solution is to replace the use of $\mathbf{x}$ by the most recent definition of it.
- Definition of a variable $\mathbf{x}$ is live at a statement that uses a variable $\mathbf{y}$, and $\mathbf{x}$ and $\mathbf{y}$ are different variables and the sets of states represented by them do not

```
TMP_WORD = para_no_0 <31..0>;
TMP_WORD = TMP_WORD << para_no_2;
GPR[para_no_1] = TMP_WORD;
```

Figure 4.2: Flattened Action for Mips SLL Instruction

intersect. In such a case no replacement can be performed.

- Definition of a variable **x** is live at a statement that uses a variable **y**, **x** and **y** are different variables and the sets of states represented by them intersect. In such a case **x** and **y** are forced to be same variable and use of **y** is replaced by the most recent definition of **x**.

4.3.11 Deletion of PC Assignments

An action sequence contains a set of statements for updating the PC and other PC-class registers (e.g. next-PC). In a non control transfer instruction such statements do not carry any useful information and so, are deleted.

4.3.12 Instruction Recognition

After simplifications, attempts are made to identify the simplified parallel action of an instruction with a standard GCC name. There is no exhaustive strategy for instruction recognition. We have followed a heuristic approach by matching the action against a set of known patterns and checking for some additional conditions (For example, whether the lvalue of an assignment is a PC-class register). If the matching succeeds and conditions are satisfied then the instruction is identified with a standard GCC name.

As an example, figure 4.2 shows the flattened action sequence of Mips SLL (Shift Left Logical) instruction. *para_no_0* is a mode rule type parameter representing a 64-bit general-purpose register. *para_no_1* and *para_no_2* are of basic cardinal

type. TMP_WORD is a temporary variable and GPR is a general-purpose 64-bit register file. After the simplification the statement gets converted to `operand_0 = operand_1 « operand_2;`

Here *operand_0* and *operand_1* are mode rule type operands, which represent indexed 64-bit general purpose registers. *operand_2* is a basic cardinal type. The simplified action readily matches the typical pattern for left shift.

Now the condition that, the lvalue is not a PC-class register, is tested. Since *operand_0* is not a PC-class register the condition is satisfied and the instruction is identified as a shift-left instruction.

As another example, consider the simplified action sequence for UltraSparcIIi instruction Branch if Register Zero:

```
if x == 0 then
  npc = npc + d;
else
endif;
```

This sequence matches the following pattern:

```
if operand1 == 0 then
  operand2 = operand2 + operand0;
else
endif;
```

Additionally the conditions that operand0 is an immediate operand, operand1 is a register operand and operand2 is a PC-class register are tested and the instruction is recognized as a branch-if-equal instruction.

4.4 Machine Description Generation

The instruction analysis phase gathers enough information about the processor for generation of GCC machine description. In particular, it determines a set of named instructions and their operands, and a set of mode rules that are 'true operands', i.e., used in named instructions. This information, along with the information about registers gathered during register analysis and mode rule analysis, is utilized in

generation of the files *target.h, target.c* and *target.md.*

4.4.1 Generation of target.h and target.c

In this phase the macros and enumeration types that define the general properties of the processor, register classes, ranges of immediate constants and addressing modes are generated in *target.h.* Table 4.1 shows the information generated in the header file. Definitions of variables that are used in the md and header file, are generated in *target.c.*

4.4.2 Generation of target.md

Finally, the instruction patterns are generated in the file *target.md.* Now, it is possible to describe a given instruction set in a number of ways. We have adopted a machine description generation strategy keeping simplicity in mind.

A single named *define_ expand* pattern is generated for all the instructions with same opcode and machine mode. Then an unnamed *define_ insn* pattern is generated for each group of instructions whose patterns differ only in the constraints of their operands. With branch patterns additional *tst* patterns are generated which compare a register operand with constant 0 and stores the result in condition code register.

Figure 4.3 shows two patterns for addition, taken from the generated md file for PowerPC 603. The first one is a named *define_ expand* pattern, which is used at the time of RTL generation. The second pattern is an unnamed *define_ insn*, which is used later for matching and generation of assembly instruction. This pattern specifies two alternative assembly instructions for addition. It also captures the fact that addition is a commutative operation.

Figure 4.4 shows two named *define_ expand* patterns, taken from the generated md file for Sparc. The first one is for comparing a register with zero and setting the condition code accordingly. The second one reads the condition code and decides whether to branch to a target. GCC ensures that a *tst* pattern and a branch pattern are always used one after another during RTL generation. It is noteworthy that the first *define_ expand* does not actually generate the *tst* pattern. It only stores its

Category	General Properties of Processor
	FIRST_PSEUDO_REGISTER
	FIXED_REGISTERS
	BITS_PER_UNIT
	BITS_BIG_ENDIAN
	BYTES_BIG_ENDIAN
	WORDS_BIG_ENDIAN
Category	Register Classes
	enum reg_class
	GENERAL_REGS
	N_REG_CLASSES
	REG_CLASS_NAMES
	REG_CLASS_CONTENTS
	REGNO_REG_CLASS
	REG_CLASS_FROM_LETTER
Category	Ranges of Immediate Constants
	CONST_OK_FOR_LETTER_P
Category	Addressing Modes
	BASE_REG_CLASS
	INDEX_REG_CLASS
	REGNO_OK_FOR_BASE_P
	REGNO_OK_FOR_INDEX_P
	REG_OK_FOR_BASE_P
	REG_OK_FOR_INDEX_P
	GO_IF_LEGITIMATE_ADDRESS
	CONSTANT_ADDRESS_P
	EXTRA_CONSTRAINT

Table 4.1: Macros and enum Types Generated in target.h

```
(define_expand  "addsi3"
[
(set
(match_operand:SI 0 "general_operand" "")
(plus:SI
(match_operand:SI 1 "general_operand" "")
(match_operand:SI 2 "general_operand" "")
)
)
]
""
""
)

(define_insn ""
[
(set
(match_operand:SI 0 "register_operand" "=a,a")
(plus:SI
(match_operand:SI 1 "register_operand" "%a,a")
(match_operand:SI 2 "register_operand" "a,a")
)
)
]
""
"@
add %0,%1,%2
addc %0,%1,%2"
)
```

Figure 4.3: Patterns for Adding Single Integers in the md File of PowerPC 603

operand in a global variable. The branch pattern actually does the work of both test and branching and so, it *uses* this global variable.

4.5 Summary

We have developed techniques for rigorous semantic analysis of a Sim-nML processor specification and automatic generation of GCC machine description from it. The generated machine description is partially complete. Some of the reasons for incompleteness in the generated machine description are as follows.

- A Sim-nML specification of the processor describes the instruction of the processor. But as noted in chapter 3, GCC needs some additional information, which are not present in it. In particular, information about the ABI, compiler environment etc. are not present in the Sim-nML specification and so, are to be added manually to the generated machine description.

- Incompleteness of the machine description can, in part, be attributed to the limitations of the tool. During instruction analysis some complex action sequences had to be ignored, as noted in section 4.3. Also, instruction recognition is heuristic in nature and cannot identify all possible and complex instruction actions.

```
(define_expand "tstdi"
[
(set (cc0)
(match_operand:DI 0 "register_operand" "")
)
]
""
"
{
target_cmp_op0 = operands[0];
target_cmp_op1 = const0_rtx;
DONE;
}
"
)

(define_expand "beq"
[
(parallel [
(set (pc)
(if_then_else
(eq (cc0) (const_int 0) )
(label_ref (match_operand 0 "" "") )
(pc)
)
)
(use (match_dup 1))
])
]
""
"
{
operands[1] = target_cmp_op0;
}
"
)
```

Figure 4.4: Test and Branch-if-equal Patterns in the md File of Sparc

Chapter 5

Results and Future Work

In this work we have developed techniques for extensive semantic analysis of a Sim-nML processor specification, which led to automatic generation of a part of GCC machine description from Sim-nML. GCC machine description generator has been tested extensively with the Sim-nML specification of UltraSPARC IIi. The generated description of SPARC has been integrated with GCC frontend and a minimal port for Sparc64 has been built. We have also generated machine descriptions of MIPS R10000, and PowerPC 603, which, though, have not been integrated with GCC frontend.

5.1 GCC Port for Sparc64

We have generated the files *target.md, target.h* and *target.c* from the Sim-nML specification of UltraSPARC IIi. With some additional human effort the machine description has been completed and a minimal GCC port for Sparc64 has been built.

Generated md file has the patterns for instructions of arithmetic-logic type (e.g. *add, sub, div, udiv, and, xor*), data movement type (*mov*), control transfer type (e.g. *beq, bne, bgt, ble*), and comparison type (*tst*). The C header file contains definitions of macros and enum types for the set of allocatable registers, addressability and endianness of memory, register classes, ranges of immediate constants, addressing modes etc., as noted in chapter 4. The C program file defines variables used in the

File Name	Status	Lines of Code
target.md	Generated	1922
target.md	Generated + Hand-coded	2694
target.h	Generated	261
target.c	Generated	260
sparc.h	Hand-coded	3617
sparc.c	Hand-coded	1489
sol2.h	Reused	185
sysv4.h	Reused	221
svr4.h	Reused	980

Table 5.1: Summary of the Effort Needed to Port Sparc64

used in the generated md and header files.

All the three generated files together consist of 2434 lines of code. To complete the port we have manually added the files *sparc.h, sparc.c, sol2.h, sysv4.h* and *svr4.h* and edited *target.md*[1]. The size of the complete port is 9707 lines. However, some of the additional files are specific to target families and so, have been reused. The table 5.1 summarises the total human effort spent to obtain the port. Our experience shows that it is possible for a person, with reasonable exposure to GCC porting mechanism, to build a port in 10 days using this tool. It is worth noting here that the GCC 2.8.1 port for *sparc-sun-sunos5.5* has 16718 lines of code. However, this port describes several versions of SPARC cpu viz. V7, V8, V9, SuperSPARC etc., contains resource-usage information, supports sophisticated optimizations, position-independent code generation etc.

GCC port for Sparc64 that we have built supports a subset of C language consisting of integer arithmetic-logic, data movement, and control transfer. In unoptimized compilation the quality of the produced code is comparable with that produced by the manually ported GCC.

We show an example of the compilation process through a simple C program that have been successfully compiled with the GCC port that we have built

[1]sparc.h includes generated target.h and sparc.c includes generated target.c

```
int main(void)
{
  int i=0, j;
  j = i+2;
  if (j > 0)
    i++;
  else
    i--;
  return 0;
}
```

The SPARC V9 assembly version of the program, as produced by our GCC port is shown below. We did not use any option except that for the generation of assembly language output (-S). In our runtime system **%i6** is the frame pointer register. The variables **i** and **j** have been assigned stack slots (**%i6** - 24) and (**%i6** - 32) respectively. C type **int** has been mapped to a 64-bit word. Register **%i0** is the return value register in the callee's window.

```
gcc2_compiled.:
.section    ".text"
  .align 4
  .global main
  .type    main,#function
  .proc 03
main:
  !#PROLOGUE# 0
  save %sp,-224,%sp
  !#PROLOGUE# 1
  st %i0,[%fp+-36]
  add %i6, -24, %g1
  xor %g0, 0, %i4
  stx %i4, [%g1]
  add %i6, -24, %g1
  ldx [%g1], %i4
  add %i4, 2, %i4
  add %i6, -32, %g1
  stx %i4, [%g1]
  add %i6, -32, %g2
  ldx [%g2], %g1
  brgz,pt %g1, .LL2
  nop
  ba .LL1
  nop
```

```
.LL2:
  add %i6, -24, %g1
  ldx [%g1], %i4
  add %i4, 1, %i4
  add %i6, -24, %g1
  stx %i4, [%g1]
  ba .LL3
  nop
.LL1:
  add %i6, -24, %g1
  ldx [%g1], %i4
  add %i4, -1, %i4
  add %i6, -24, %g1
  stx %i4, [%g1]
.LL3:
  xor %g0, 0, %i0
  ret
  restore
  ret
  restore
.LLfe1:
  .size main,.LLfe1-main
  .ident "GCC: (GNU) 2.8.1"
```

5.2 Future Directions

Some of the possible directions to which the work presented in this thesis can be extended are given below.

- Work can be carried out to make generated machine description more complete so that the total effort needed to obtain a GCC port is further reduced and to improve the quality of the generated description so that the GCC port can

produce better code.

Resource usage information available in a Sim-nML specification can be analyzed to generate the definitions *define_function_unit*, *define_delay* and *define_attribute*, which will allow GCC to perform instruction scheduling and delay slot scheduling.

Instruction analysis can be made more powerful so that bit-selection and concatenation can be handled elegantly.

Heuristics of instruction recognition can be improved so that complex action sequences that normally appear in descriptions of CISC architectures can be recognized.

Simple machine description generation strategy adopted by us can be replaced by a more mature one so that more compact descriptions can be generated and sophisticated optimizations can be supported.

- Another possibility is to develop a new retargetable backend, that can be integrated with an existing frontend. Techniques are to be developed for instruction selection, resource allocation, and instruction scheduling. Information gathered during instruction analysis phase can be used in instruction selection. Further analysis of resource-usage is needed for resource allocation and instruction scheduling.

Bibliography

[1] S. Chandra and R. Moona. Retargetable functional simulator using high level processor models. In *Proceedings of the 13th International Conference on VLSI Design, Calcutta, India.*, January 2000.

[2] Sanjeev Kumar and V. M. Malhotra. Automatic Retargetable Code Generation: A New Technique. *Foundations of Software Technology and Theoritical Computer Science, Lecture Notes in Computer Science*, vol. 241, Springer-Verlag, 1986.

[3] Fauth A., Praet Vwn J. , and M. Freericks *Describing Instruction Sets Using nML (Extended Version).* Available at: ftp://ftp.imec.be/pub/vsdm/reports/retargetable_code_generation/af-edtc95.ps.gz, 1995.

[4] S. D. G. Hadjiyiannis, Silvina Hanono. ISDL: An instruction set description language for retargetability. In *Proceedings of the 34th DAC*, June 1997.

[5] M. Ganapathi and C. N. Fischer. Affix grammer driven code generation. *ACM TOPLAS*, 7(4), October 1985.

[6] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 231–240, 1978.

[7] N. C. Jain. Disassembler using high level processor models. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1997/9711113.html, January 1999.

[8] S. Kumari. An automatic assembler generator for sim-nml description language. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1998/9811119.html, March 2000.

[9] Rajesh V. A Generic Approach to Performance Modeling and its Application to Simulator Generator. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur. Available at: http://www.cse.iitk.ac.in/sim-nml/index.cgi.

[10] Subhash Chandra Y. Retargetable Functional Simulator. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur. Available at: http://www.cse.iitk.ac.in/sim-nml/index.cgi.

[11] Lanneer D., Praet J. V., Kifli A., Schoofs K., Geurts W., Thoen F. and Goossens G. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Systems.* Kluwer Academic Publishers, 1995.

[12] P. Marwedel. The MIMOLA Design System: Tools for the design of digital processors. In *Proceedings of the 21st DAC*, pages 587–593, 1984.

[13] S. Mondal. Compiler back-end generation using nml machine description. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1997/9711117.html, June 1999.

[14] Hanson D., Fraser C. W. and Proebsting T. Engineering a simple, efficient code generator generator. Available at: http://sunsite.org.uk/Mirrors/ftp.cs.princeton.edu/pub/lcc/contrib.

[15] Paulin. Flexware: A flexible firmware development environment for embedded systems. In *Code Generation for Embedded Systems.* Kluwer Academic Publishers, 1995.

[16] P. Pogde. Retargettable code generation using sim-nml machine description. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1998/9811114.html, May 2000.

[17] V. Rajesh and R. Moona. Processor modeling for hardware software co design. In *Proceedings of the 12th International Conference on VLSI Design, Goa, India.*, January 1999.

[18] R. Ravindran. Retargetable profiling tools and their application in cache simulation and code instrumentation. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1998/9811116.html, Dec 1999.

[19] R. Ravindran and R. Moona. Retargetable cache simulation using high level processor models. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference, Gold Coast, Australia*, January 2001.

[20] S. D. Silvina Hanono. Instruction selection, resource allocation and scheduling in the aviv retargetable code generator. In *Proceedings of the DAC*, June 1998.

[21] R. M. Stallman. *Using and Porting GNU CC.* http://gcc.gnu.org/onlinedocs/gcc.html.

[22] Aho Alfred V., M. Ganapathi, and S. Tjiang. Code generation using tree pattern matching and dynamic programming. *ACM TOPLAS*, 11(4), October 1989.

[23] Aho Alfred V., Sethi Ravi, and Ullman Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1999.

[24] Muchnick Steven S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[25] Marwedel P. *Compilers for Embedded Processors*. Available at: http://ls12-www.cs.uni-dortmund.de/publications/global_index.html

[26] Malik S. *Optimal Code Generation For Embedded Memory Nonhomogeneous Register Architectures*. In 8th International Symposium on System Synthesis (ISSS), 1995.

[27] Leupers R., Marwedel P. *Retargetable Generation of Code Selectors from HDL Processor Models*. Available at: http://ls12-www.cs.uni-dortmund.de/publications/global_index.html

[28] Stanford Compiler Group. *The SUIF Library*. 1.0 edition, 1994. Available at http://suif.stanford.edu.

[29] SPAM Research Group. *SPAM Compiler User's Manual*. 1.0 edition, 1997. Available at: www.ee.princeton.edu/spam/

[30] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau. *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability*. Available at: http://www.cecs.uci.edu/

[31] S. Pees, A. Hoffmann, V. Živojnovic, and H. Meyr. *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures* Available at: http://www.ert.rwth-aachen.de/Projekte/Tools/LISA/index.html

[32] Mescal (Modern Embedded Systems: Compilers, Architectures, and Languages). http://www.gigascale.org/mescal/index.html

[33] Sim-nML Processor Description Language. http://www.cse.iitk.ac.in/sim-nml/index.cgi.

[34] Ziovojnovic V., Tjiang S., and Meyr H. *DSPstone: A DSP-oriented Benchmarking Methodology*. In International Conference on Signal Processing Applications and Technology (ICSPAT), 1994.

[35] Weaver D. L. and Germond T. The SPARC Architecture Manual, Version 9. SPARC International, Inc., 1994.

[36] Sun Microsystems. UltraSPARC - IIi User's Manual. Available at: http://www.sun.com/microelectronics/UltraSPARC/index.html.

[37] GNU Compiler Collection. http://gcc.gnu.org

Appendix A

GCC Internals

In this appendix we present some useful information about GCC internals. Emphasis will be given on a comprehensive organization. Much of the information provided here are available at http://gcc.gnu.org/onlinedocs/gcc.html [21]. Some more information is gathered from GCC 2.8.1 sources and is presented here.

A.1 Components of GCC Compiler Suite

Following are the software components needed for the functioning of GCC. Some of these come with GCC, while others are provided by the system or one third party software.

- Preprocessor (`cpp, cccp`).
- Compiler proper (`cc1, cc1plus, cc1obj, f771`).
- Assembler (e.g. `as`, provided by the system).
- Linker and a linker frontend (`ld, collect2`).
- Headers (GCC specific headers are searched in `prefix/gcc-lib/target-name /gcc-version/include` and system headers are searched in `local-prefix/include`. 'prefix' defaults to /usr/local/lib/ and local-prefix defaults to /usr/local/).

- Library (GCC provides the following libraries: `libgcc`, `libgcc1`, `libg2c`, `libobjc`, `libstdc++`).
- Start up files (e.g. `crtbegin.o`, `crtend.o` etc).
- Compiler driver (`gcc`, `g++`, `g77`).

A.2 A Grouping of RTL Expression Codes

In this section we classify the RTX codes on the basis of their uses.

A.2.1 Operands

Following RTL expressions can appear as operands in the side-effects of *insns*.

- *Constants*

RTX codes representing constant operands are as follows.

const_int, const_double, const_string, symbol_ref, label_ref, const, high.

- *Registers and Memory*

RTX codes representing register and memory operands are as follows.

reg, subreg, scratch, cc0, pc, mem, addressof.

- *Bit Fields*

Following RTX codes represent bit-fields within a register or memory location.

sign_extract, zero_extract.

- *Type Conversions*

Following RTX codes are used for converting types of operands.

sign_extend, zero_extend, float_extend, truncate, float_truncate, float, unsigned_float, fix, unsigned_fix.

- *Declaration*

The following RTX is used to declare that only lower half of the operand will be modified.

strict_low_part.

A.2.2 Operations

In this subsection we group the RTX codes used to represent operations in a side-effect of an *insn.*

- *Arithmetic-Logic*

Following RTX codes represent arithmetic-logic operations. These RTXes produce values, which are same as the result of the operation.

plus, lo_sum, minus, compare, neg, mult, div, udiv, mod, umod, smin, umin, smax, umax, not, and, ior, xor, ashift, lshiftrt, ashiftrt, rotate, rotatert, abs, sqrt, ffs.

- *Comparison*

Following RTXes represent comparison operations. These RTXes can be used to compare two registers, or a register and a constant, or a condition code and *(const_int 0)*.

eq, ne, le, leu, lt, ltu, ge, geu, gt, gtu, if_then_else, cond.

A.2.3 Side Effects

Following RTXes represent functionality of an *insn.* They do not produce any value. But they may modify a processor state.

set, return, call, trap_if, clobber, use, parallel, sequence, asm_input, asm_output, unspec, unspec_volatile, addr_vec, addr_diff_vec.

A.2.4 Embedded Side Effects

These are special side-effects which may be associated with memory addresses.

pre_dec, pre_inc, post_dec, post_inc.

A.2.5 Insns

Following is a list of RTX codes for *insns*.

insn, call_insn, jump_insn, note, barrier, code_label.

A.2.6 RTL Templates

Following is a list of RTXes which are used as place-holders for operands or operations within a pattern in the md file.

match_operand, match_scratch, match_dup, match_operator, match_parallel, match_op_dup, match_par_dup, address.

A.2.7 Definitions

Following RTX codes are used to define various things, for example, instructions, functional units, etc., inside an md file.

define_expand, define_insn, define_peephole, define_split, define_combine[1], define_delay, define_function_unit, define_attr.

A.3 A Grouping of Standard GCC Names

In this section we present a classification of standard pattern names used in GCC machine description.

A.3.1 Data Movement

Following names are used for instruction patterns which move data between two registers, or between a register and a memory location etc.

[1]unused in GCC 2.8.1

mov*mode*, reload_in*mode*, reload_out*mode*, movstrict*mode*, load_multiple, store_multiple, move*modecc*.

A.3.2 Arithmetic-Bitwise Operations

Following names represent instructions which perform arithmetic-logic operations on their operands and store the result.

add*mode*3, sub*mode*3, mul*mode*3, div*mode*3, udiv*mode*3, mod*mode*3, umod*mode*3, smin*mode*3, umin*mode*3, smax*mode*3, umax*mode*3, mulhisi3, mulqihi3, mulsidi3, umulhisi3, umulqihi3, umulsidi3, mul3_hipart, umul3_hipart, divmod*mode*3, udivmod*mode*3, neg*mode*2, abs*mode*2, sqrt*mode*2.

and*mode*3, ior*mode*3, xor*mode*3, ashl*mode*3, ashr*mode*3, lshr*mode*3, rotl*mode*3, rotr*mode*3, one_cmpl*mode*2, ffs*mode*2, insv, extv, extzv.

A.3.3 Type Conversions

Following names are used for instructions which convert the type of data.

float*mn*2, floatuns*mn*2, fix*mn*2, fixuns*mn*2, ftrunc*mode*2, fix_trunc*mn*2, fix_unstrunc*mn*2, trunc*mn*2, extend*mn*2, zero_extend*mn*2.

A.3.4 Comparisons

These are names for instructions which compare their operands and store the result in a condition code or any ordinary register.

cmp*mode*, tst*mode*, s*cond*.

A.3.5 String Operations

Instructions with following names perform operations on string.

movstr, clrstr, cmpstr, strlen.

A.3.6 Control Transfers

Instructions with following names are responsible for constrol transfer.

b*cond*, indirect_jump, jump, call, call_value, call_pop, call_value_pop, untyped_call, return, untyped_return, casesi, tablejump, nonlocal_goto, nonlocal_goto_receiver, exception_receiver, builtin_setjmp_receiver.

A.3.7 Stack Operations

Following names are for instructions that access and manipulate stack.

save_stack_block, restore_stack_block, save_stack_function, restore_stack_function, save_stack_nonlocal, restore_stack_nonlocal, allocate_stack, probe, check_stack.

A.3.8 Others

Names which do not fit into any of the above categories are listed here.

nop, canonicalize_funcptr_for_compare.

A.4 Useful RTX Related Functions and Macros

Several functions and macros are defined in the source files of GCC, which are used to read and manipulate RTL expressions. These macros are often used in the GCC machine descriptions. In this section we present a list of these macros. The exact definitions can be found in the corresponding source files.

- *Macros Defined in rtl.h*

GET_CODE(),
PUT_CODE(),
GET_RTX(),
LENGTH(),
GET_RTX_FORMAT(),
GET_RTX_CLASS(),
XEXP(),
XINT(),
XWINT(),

XVEC(),
XVECLEN(),
XVECEXP(),
GEN_INT().

- *Functions Defined in rtl.c*

read_rtx().

- *Functions Defined in emit-rtl.c*

gen_rtx(), gen_reg_rtx(), gen_label_rtx().

- *Functions Defined in print-rtl.c*

print_rtx(), print_rtl().

A.5 Machine Mode Related Macros

A list of macros defined in differnt source files of GCC, which are used to access machine modes, are presented below.

- *Macros Defined in rtl.h*

GET_MODE(),
PUT_MODE().

- *Macros Defined in machmode.h*

GET_MODE_NAME(),
GET_MODE_CLASS(),
INTEGRAL_MODE_P(),
FLOAT_MODE_P(),
GET_MODE_SIZE(),

GET_MODE_UNIT_SIZE(),
GET_MODE_NUNITS(),
GET_MODE_BITSIZE(),
GET_MODE_MASK(),
GET_MODE_WIDER_MODE(),
GET_MODE_ALIGNMENT(),
GET_CLASS_NARROWEST_MODE().

A.6 Functions Related to Insns

In this section we list the functions which are responsible for emitting *insn*s. These functions are sometimes used in machine descriptions to explicitly control the generation of *insn*s.

- *Functions Defined in emit-rtl.h*

emit_insn(), emit_call_insn(), emit_jump_insn().

A.7 Set of Built-in Predicates

A set of basic predicates are defined in the sources of GCC. Here we present a list of them.

- *rtl.c defines a set of useful predicates*

general_operand, register_operand, immediate_operand,
const_int_operand, const_double_operand, non_immediate_operand,
memory_operand, nonmemory_operand, indirect_operand,
push_operand, address_operand, comparison_operator.

A.8 Notion of an Address

There is a notion of an address of a memory location within GCC. The memory location may contain data or may be target of a control transfer. For example, the first operand of a *mem* RTX is the address of a memory location. Similarly, the first operand of an *indirect_jump* or *jump* pattern is an address, which specifies the target of the jump.

A.8.1 RTXes used as Addresses

These are the RTXes which may be used as addresses.

const_int, const_double, symbol_ref, label_ref, high, const, RTXes for arithmetic operations and conversion (see A.2.1 and A.2.2), addressof, scratch, reg, mem (a *mem* RTX may refer to the contents of a memory location, which may in turn be an address. A *mem* RTX may also refer to the address of a memory location, for example, in the case, when constraint letter 'p' is used.).

A.8.2 Definition of a Valid Address

Following macros, predicates and constraints are used to define the notion of a valid address:

- *Macros Defined in target.h*

CONSTANT_ADDRESS_P,
GO_IF_LEGITIMATE_ADDRESS,
REG_OK_FOR_BASE_P,
REG_OK_FOR_INDEX_P,
GO_IF_MODE_DEPENDENT_ADDRESS,
REG_MODE_OK_FOR_BASE_P,
MAX_REGS_PER_ADDRESS,
HAVE_POST_INCREMENT,
HAVE_PRE_INCREMENT,

HAVE_POST_DECREMENT,
HAVE_PRE_DECREMENT,
LEGITIMIZE_ADDRESS,
EXTRA_CONSTRAINTS,
BASE_REG_CLASS,
INDEX_REG_CLASS,
REGNO_OK_FOR_BASE_P,
REGNO_OK_FOR_INDEX_P,
REGNO_MODE_OK_FOR_BASE_P,
PRINT_OPERAND_ADDRESS.

- *Predicates Defined in recog.c*

address_operand, memory_operand, indirect_operand, general_operand.

- *Constraint Letters Defined in constrain_operands() in recog.c*

m: allows a memory operand with any kind of address
o: allows a memory operand, but only if the address is offsettable
V: aloows a memory operand, only if its address is not offesttable
<: allows a memory operand with autodecrement addressing (both predecrement and postdecrement are allowed)
>: allows a memory operand with autoincrement addressing (both preincrement and postincrement are allowed)
p: represents an operand that is a valid memory address

A.9 Translation of C Level Data to Machine Level

Figure A.1 shows the way GCC translates C level data to hard registers, or memory locations, or immediate constants. Information needed in each step are presented below.

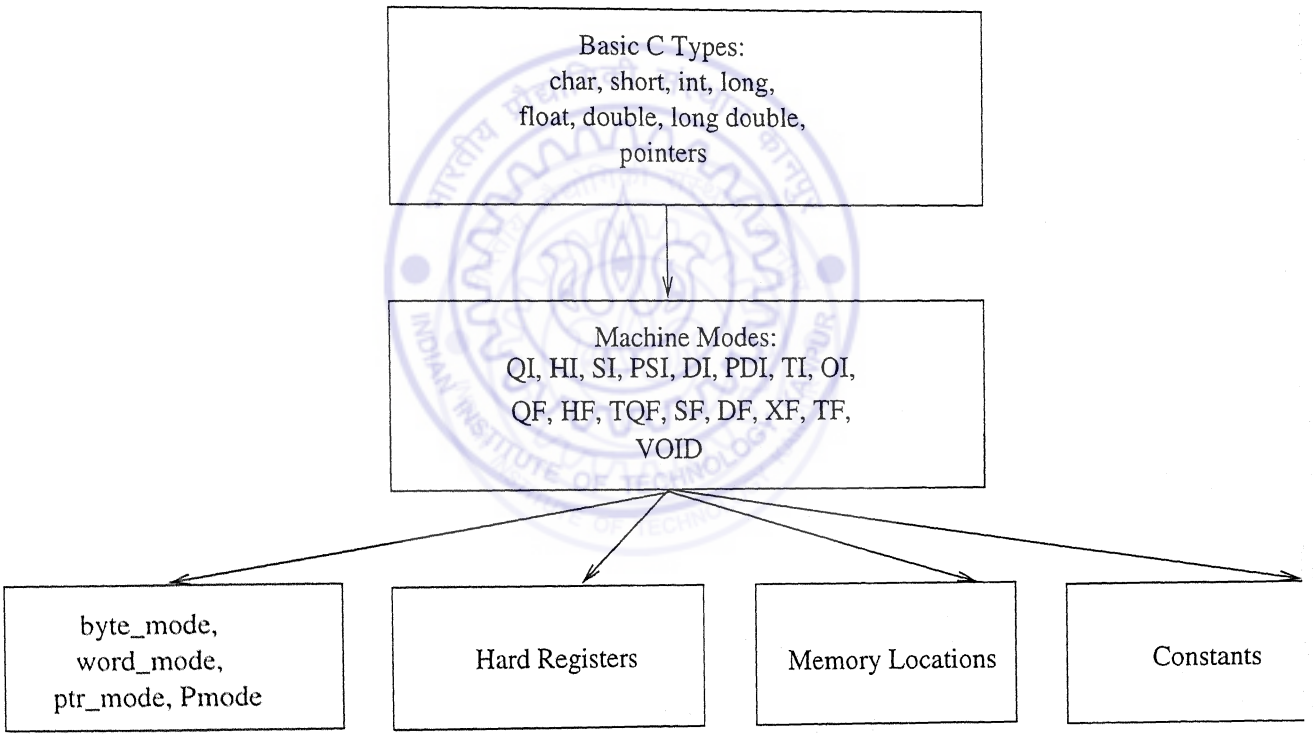


Figure A.1: Translation of Data

A.9.1 Translation to Machine Modes

In this subsection we present a list of macros and definitions which contain the information needed for translating basic C types to GCC machine modes.

1. Definitions of machine modes (in terms of number of units per mode) in *machmode.def*

2. Macros defined in *target.h*:

BITS_PER_UNIT,
INT_TYPE_SIZE,
SHORT_TYPE_SIZE,
LONG_TYPE_SIZE,
CHAR_TYPE_SIZE,
FLOAT_TYPE_SIZE,
DOUBLE_TYPE_SIZE,
LONG_DOUBLE_SIZE,
MAX_FIXED_MODE_SIZE.

A.9.2 Definitions of *byte_mode, word_mode* etc

Following are the macros which contain the information needed for defining the variables *byte_mode* and *word_mode*.

1. Macros defined in *target.h*

BITS_PER_WORD,
UNITS_PER_WORD,
POINTER_SIZE,
Pmode,
POINTER_EXTENDED_UNSIGNED.

A.9.3 Mapping to Hard Registers

Following macros help in mapping high-level language operands to registers of the machine.

1. Macros and enum types defined in *target.h*

FIRST_PSEUDO_REGISTER,
REGISTER_NAMES,
FIXED_REGISTERS,
CALL_USED_REGISTERS,
HARD_REGNO_NREGS,
HARD_REGNO_MODE_OK,
MODES_TIEABLE_P,
BITS_BIG_ENDIAN,
BYTES_BIG_ENDIAN,
WORDS_BIG_ENDIAN,
FLOAT_WORDS_BIG_ENDIAN,
PROMOTE_MODE,
PROMOTE_FUNCTION_ARGS,
PROMOTE_FUNCTION_RETURN,
PROMOTE_FOR_CALL_ONLY,
TARGET_FLOAT_FORMAT,
REG_CLASS_FROM_LETTER,
N_REG_CLASSES,
enum reg_class,
REG_CLASS_NAMES,
REG_CLASS_CONTENTS,
PREFERRED_RELOAD_CLASS,
PREFERRED_OUTPUT_RELOAD_CLASS,
SECONDARY_INPUT_RELOAD_CLASS,
SECONDARY_OUTPUT_RELOAD_CLASS,
SECONDARY_MEMORY_NEEDED,
CLASS_MAX_NREGS, EXTRA_CONSTRAINT.

2. Predicates defined in *recog.c*

general_operand, register_operand.

A.9.4 Mapping to Memory Locations

Following macros contain information needed for mapping high-level language operands to memory locations.

1. Macros defined in *target.h*

BITS_BIG_ENDIAN,
BYTES_BIG_ENDIAN,
WORDS_BIG_ENDIAN,
PARM_BOUNDARY,
BIGGEST_ALIGNMENT,
MINIMUM_ATOMIC_ALIGNMENT,
BIGGEST_FIELD_ALIGNMENT,
DATA_ALIGNMENT,
STRICT_ALIGNMENT,
ADJUST_FIELD_ALIGN,
EMPTY_FIELD_BOUNDARY,
STRUCTURE_SIZE_BOUNDARY,
PCC_BITFIELD_TYPE_MATTERS,
GO_IF_LEGITIMATE_ADDRESS.

2. Predicates defined in *recog.c*

general_operand, memory_operand,
indirect_operand, address_operand.

A.9.5 Translation of Constants

Constants or literals that appear in a high-level language program are translated to immediate operands of instructions or memory objects. The following macros contain information needed for this translation.

1. Macros defined in *target.h*

LEGITIMATE_CONSTANT_P,
CONSTANT_ALIGNMENT,
REAL_VALUE_TYPE,
TARGET_FLOAT_FORMAT,

CHECK_FLOAT_VALUE,
REAL_VALUE_TO_TARGET_SINGLE,
REAL_VALUE_TO_TARGET_DOUBLE,
REAL_VALUE_TO_TARGET_LONG_DOUBLE,
REAL_VALUE_TO_DECIMAL,
ASM_OUTPUT_ASCII,
ASM_OUTPUT_BYTE,
ASM_OUTPUT_CHAR,
ASM_OUTPUT_SHORT,
ASM_OUTPUT_INT,
ASM_OUTPUT_DOUBLE_INT,
ASM_OUTPUT_QUADRUPLE_INT,
ASM_OUTPUT_BYTE_FLOAT,
ASM_OUTPUT_SHORT_FLOAT,
ASM_OUTPUT_THREE_QUARTER_FLOAT,
ASM_OUTPUT_FLOAT,
ASM_OUTPUT_DOUBLE,
ASM_OUTPUT_LONG_DOUBLE,
PRINT_OPERAND.

2. A macro defined in *rtl.h*

CONSTANT_P.

3. Predicates defined in *recog.c*

general_operand, immediate_operand,
const_int_operand, const_double_operand.

Appendix B

genmd2 Maintainer's Guide

This appendix contains some information useful for maintaining the tool *genmd2*. This tool implements the techniques for generating GCC machine description from Sim-nML. This appendix complements the comments associated with the source files.

B.1 Source Files

Source files are stored inside a CVS repository and all the versions of the files can be retrieved from the repository. Log messages associated with the versions may be useful in tracking past changes.

Following files and directories can be found in the root of the distribution of *genmd2*.

- **genbackend.c:** The toplevel module that contains the main() function. It drives all the phases of the backend generator in order as discussed earlier. Further it also parses the command line arguments and reads the configuration file.
- **irview.c:** Reads the Sim-nML IR.
- **registers.c:** Contains code for the register analysis and generation of *target.h* and *target.c*.

- **flattenModes.c:** Contains code for the mode rule analysis.
- **analyze-mode.c:** This code is used for register analysis, mode rule analysis and mode rule synthesis.
- **flatten.c:** *action* flattener.
- **flatten_syntax.c:** *syntax* flattener.
- **analyze-insn.c:** Instruction analysis. Does some work associated with flattening.
- **recog.c:** Instruction recognition. Also assigns constraint letters to 'true operand' mode rules.
- **emit_insn.c:** Generates *target.md.*
- **include:** A directory containing header and definition files.
- **include/systypes.h:** Defines some system specific types used within the sources.
- **include/decls.h:** Declarations of global variables and functions.
- **include/tables.h:** Data structures for Sim-nML IR.
- **include/operands.h:** Data structures for mode tables and register analysis.
- **include/instructions.h:** Data structures for instruction table.
- **include/syntax.h:** Data structures for syntax table.
- **include/md_operands.h:** Data structures for named instruction patterns.
- **include/opcodes.def:** Defines the opcodes used within standard GCC names.
- **include/modifiers.def:** Defines the RTXes used as modifiers of values or operands.
- **Makefile:** Make file.

- **test:** Working directory. 'make' generates the binary executable of the tool in this directory.
- **test/template.conf:** A template for a configuration file.

B.2 Intermediate Dumps

genmd2 produces intermediate dump files at several phases. These files are useful for debugging the tool.

- **meminfo.table:** Produced by register analysis.
- **mode.table:** Dump of mode table, produced by mode rule analysis.
- **sec_mode_.table:** Dump of secondary mode table, produced by mode rule analysis.
- **base_index_disp.table:** Maps of base, index, and window pointer registers and range of valid displacement. It is produced after mode rule analysis.
- **instr.table:** Intermediate version of instruction table, produced during action flattening. May contain some extra instructions and may not show some valid instructions.
- **syntax.table:** Syntax table, which is produced after syntax flattening.
- **instr1.table:** Instruction table, which is produced after flattening is complete.
- **par_stmt1.table:** Instruction table, which is produced after morphing of parameters.
- **par_stmt2.table:** Instruction table, which is produced after all temporaries are removed.
- **par_stmt3.table:** Instruction table, which is produced after PC assignments are deleted.

- **syntax1.table:** Syntax table, which is produced after instruction analysis is complete.
- **third_mode.table:** Mode table, which is produced after instruction analysis is complete.
- **fourth_mode.table:** Secondary mode table, which is produced after instruction analysis is complete.
- **base_index_disp1.table:** Maps of base, index, and window pointer registers and range of valid displacement. It is produced after instruction analysis.

B.3 A Grammar for Value Expressions

Following is a partial specification of the grammar used by our tool for a recursive analysis of the prefix expressions which can appear as values of mode *and* rules. This grammar is a subset of the Sim-nML grammar for expressions which can appear as a value. The original Sim-nML grammar is too general, too complex and often, can result in value expressions that are impractical. The names of terminal symbols (all capitalized) conform to the names of OPERATOR_TYPE and BYTE_TYPE enumeration constants, as defined in include/tables.h. ValIndexExpr represents an expression which can be used as an index of a register or a memory location. ValCondExpr represents an expression that can be used as a condition in an *if-then-else-endif* expression.

```
ValExpr :
   ID
   | COERCE ValTypeExpr CARDINAL_CONSTANT CARDINAL_CONSTANT ValExpr
   | .  ID ID
   | DCOLON ValExpr ValExpr
   | INDX ID ValIndexExpr
   | BITLR INDX ID ValIndexExpr ValBitExpr ValBitExpr
   | BITLR ID ValBitExpr ValBitExpr
   | + ValExpr ValExpr
```

```
    | - ValExpr ValExpr
    | * ValExpr ValExpr
    | / ValExpr ValExpr
    | % ValExpr ValExpr
    | EXP ValExpr ValExpr
    | LSFT ValExpr ValExpr
    | RSFT ValExpr ValExpr
    | RLFT ValExpr ValExpr
    | RRHT ValExpr ValExpr
    | < ValExpr ValExpr
    | > ValExpr ValExpr
    | LEQ ValExpr ValExpr
    | GEQ ValExpr ValExpr
    | EQ ValExpr ValExpr
    | NEQ ValExpr ValExpr
    | LAND ValExpr ValExpr
    | LOR ValExpr ValExpr
    | LXOR ValExpr ValExpr
    | BUNOT ValExpr
    | AND ValExpr ValExpr
    | OR ValExpr ValExpr
    | !  ValExpr
    | FIXED_CONSTANT
    | CARDINAL_CONSTANT
    | BINARY_CONSTANT
    | HEX_CONSTANT
    | IF ValCondExpr ValExpr OptValExpr
ValTypeExpr :
    BOOL
    | CARD
    | INT
```

```
    | FIX
    | FLOAT
    | RANGE
    | ENUM
ValBitExpr :
    CARDINAL_CONSTANT
OptValExpr :
    NULL
    | ValExpr
```

Appendix C

genmd2 User's Manual

genmd2 is the tool that implements the techniques for GCC machine description generation from Sim-nML, as outlined in this thesis. The inputs to the tool are a Sim-nML IR file and a configuration file. The tool generates the files *target.h, target.c* and *target.md.*

C.1 System Requirements

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

- **Processor:** Intel Pentium III, 32 bit, Little Endian
- **OS Kernel:** Linux 2.2.15-mdk
- **Compiler:** GCC 2.95.3 19991030 (prerelease), used for building the tool.
- **Libraries:** GNU C Library Version 2.1 Beta, used for building the tool.
- **Binary Utilities:** GNU Binary Utilities Version 2.9.5, used for building the tool.

C.2 Installation

A make file is provided along with the source of *genmd2*. The tool can be built using this make file by giving the following the following command at the root of the source tree

```
make
```

The tool will be built and stored as a binary executable file test/genmd2. The compiled binary can be moved to any directory.

C.3 Running the Tool

Following is the comand line specification for the tool:

```
genmd2 ir_file_name [OPTIONS] [-c config_file_name]
```

`genmd2` is the name of the binary executable file for the tool.

`ir_file_name` is the name of the Sim-nML IR file.

The tool *genmd2* supports the following command line options:

-s SP_REG : Specifies the stack pointer register name.

-f FLAG_REG : Specifies the condition code regsiter name.

-p PC_REG : Specifies the name of the program counter.

The user can optionally specify a configuration file which is a flexible and powerful way to provide additional information about the processor. On a conflict between the information provided through a command line option and the configuaration file the information from the command line is ignored.

C.4 Configuration File

The configuration file consists of a number of sections. Each section can have zero or more entries. A section with zero entries can be omitted.

An entry in a section refers to a single register and is a single line of the following form

```
register_file_name:index_in_the_register_file
```

If the register being referred to is not in a register file then the regsiter_file_name is same as the name of the register and the index is 0.

A configuration file ends with the following line:

`/end`

A template for a configuration file is provided along with the source in test/template.conf file.

C.4.1 PC Section

PC section begins with the line

`pc`

and ends with the line

`/pc`

Its entries refer to PC-class registers e.g. PC, next-PC etc.

C.4.2 CC Section

CC section begins with the line

`cc`

and ends with the line

`/cc`

Its entries refer to the condition code registers.

C.4.3 SP Section

SP section begins with the line

`sp`

and ends with the line

`/sp`

It has an entry for the stack pointer register.

C.4.4 Return Address Pointer Section

Return address pointer section begins with the line

`rap`

and ends with the line

`/rap`

It has an entry for the return address pointer register.