RETARGETABLE BINARY TOOLS FOR EMBEDDED SOFTWARE

by

Wai Sum Mong

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Retargetable Binary Tools for Embedded Software

Wai Sum Mong

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2004

Equipping retargetability to the embedded software development tool suite is the enabler of architectural exploration in the context of system-on-chip design. While the study of retargetable compilers has been active for long, retargetting equally important binary tools, including linkers and micro-architecture simulators, has not received enough attention. In this thesis, we propose a unique methodology where the architecture-dependent components of production quality binary tools are automatically generated from abstract architectural models. Our architectural model includes not only the instruction set architecture (ISA) model, but also the application binary interface (ABI) model, a subject not previously reported. With this methodology, we are able to automatically port Free Software Foundation (GNU)'s Binary File Descriptor (BFD) library and GNU linker, the *de facto* standard for linking, as well as SimpleScalar, the *de facto* standard for micro-architecture simulation, all based on a common specification of an arbitrary RISC-like processor.

Acknowledgements

First of all, I would like to express my deepest sense of gratitude to my supervisor Professor Jianwen Zhu for his patient guidance, assistance and encouragement. Keeping the patience to a stubborn student is really difficult. Before joining this group, I never think anyone would give me the opportunities that I have in this few years.

Second, I would like to acknowledge Maghsoud Abbaspour for his contribution of starting up the retargetable binary utilities project.

My thanks also goes to every friend during my master study, special thanks to Zhong, Khushwinder, Fang, Rami and Dennis.

Thanks to all the radio programs that keep me awake when I was working at night.

It is very important to thank my parents for all the support and the education chance they give me. I would also like to thank all the people who have assisted me in adapting life in Toronto, especially my uncle Peter and aunt Angela. Thank to Calven, Sam, Jenny and Vivian, they always listen to me.

Last but certainly not least, special thank to my boy friend, Kelvin, for his love, constant support and understanding. His driving and food delivery service is very important to my master career.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to the increasing market demands of innovative electronic products such as cellular phones, digital cameras and PDAs, embedded system designers are facing challenges of handling growing system complexity under tighter time-to-market constraint. To shorten the design cycle, it is generally conceived that pushing as much functionality as possible to software implementation is one of the must attractive. Unfortunately, this is not absolutely true unless the software development tools including compiler, assembler, linker, debugger as well as simulator, are always readily available.

The implementation of an embedded system can vary from constructing custom circuits in order to fully tailor the target application (namely the hardware approach), to programming all the functions on a standard component such as a digital signal processor (DSP) (namely the software approach). Even though the hardware approach dominates in the past, the embedded system design today is gradually shifting toward the software approach. This is primary due to the fact that while chip manufacturing technologies have made dramatic advances and allowed capturing more functions in one embedded system, the rapidly increasing chip design complexity and narrow time-to-market window become the bottleneck of both productivity and development cost. It is believed that the software approach can improve this problem for the following reasons: 1) the immediate availability of programmable processors and hence short design time, 2) the low cost due to the amortization of non-recurring engineering (NRE) cost of processor design over large number of products, and especially 3) the flexibility, which facilitates late design changes and easy product upgrades. As a result, it is not surprising to find that the amount of software code in embedded system implementation is doubling every two years [16]. For this reason, the development of embedded software becomes more and more important.

However, embedded systems often need the advantages that custom-designed processors offer, such as enhanced performance and lower power consumption. As a compromise between the hardware approach and the software approach, application specific instruction set processors (ASIPs), which are optimized for a particular class of applications, are widely used in such a way that the hardware architecture and the instruction sets of the ASIP are tailored for the critical functions of a specific application class while keeping certain degree of flexibility from the software approach. For example, an image-processing ASIP might include special instructions to optimize the image compression function. Running software in application-specific processors implies that processor-dependent tools including compiler, assembler, linker and instruction-set simulator have to be developed for each architecture. This task is unfortunately non-trivial, which we will elaborate in the next section. This therefore brings in the problem that we tried to contribute in this thesis - What's the methodology to develop production-quality software development tools for application-specific embedded processor architectures in an efficient manner under time-to-market constraint?

In this thesis, we try to address the above problem by equipping a selected set of binary tools and processor simulators, which are in *de facto* standard, with *retargetability*. Portability denotes the ease of modifying a processor-dependent tool to support any other processor architecture. If high effort is required to retarget a tool to a new architecture, the tool is said to be *un-portable*. At the opposite extreme, the tool is said to be *retargetable* if the tool can be automatically ported.

The subsequent sections of this chapter elaborate on the motivations, objectives and methodology, and concludes with a summary of contributions of this thesis.

1.1 Motivation

Two sets of tools play very important roles in embedded processor design; they are *software development tools* and *processor simulators*. Enabling the automatic generation of such tools can boost the productivity and reduce the development cost. Even though this topic has become a very active area of research in the last few years, the problem has not yet been completely solved.

1.1.1 Retargetable Software Development Tools

The software development toolset of a processor usually consists of the compilation tools including the compiler, the assembler and linker, as well as utilities including the debugger and the disassembler.

The importance of retargetable compilers has been recognized and the research of which is becoming mature [10, 9, 13, 6]. However, few efforts have been made in the automatic generation of equally important downstream tools such as assemblers, linkers and debuggers. This is because that most people have such misconception - the downstream tasks, such as assembling and linking, are trivial comparing to the advanced compiler problem of optimizing the code quality for specific embedded application [9]. Two phenomena reflect this. First, little treatment of the relevant techniques has appeared in the computer science curriculum in the schools. Second, virtually no paper has discussed the techniques of the linker/assembler construction [1]. Nevertheless, this perception is no more acceptable in the modern computer system. Let's take the object file linking task as an example. The traditional linker is only responsible for threading the object files and resolving the symbols. However, the modern linker has to handle complex features such as dynamic linking, the routine of which is however processor-dependent.

The manual development of these downstream tools for a new processor architecture is not easy. The most recent version of the *GNU Binutils* package, which delivers exactly these downstream tools, has a daunting size of 250k lines of C code [15]. Until now, the GNU Binutils possesses the best portability among all toolset of this kind. Without any detailed documentation and cleanly defined interface between the processor-dependent and independent part, hacking the GNU Binutils and porting to a new processor is however non-trivial. On the other hand, the black magic involved and the skills required to develop these tools is mastered by a very small group of people such that it has been once asserted the population of the experts in this world could probably fit in one room [11]. Thus, we can conclude that manually hand-crafting the downstream tools for each new application-specific processor is both inefficient and expensive. Additionally that human work in highly-complexed system is always error-prone, automation generation technique of the downstream software development tools is therefore urgently needed.

1.1.2 Retargetable Processor Simulators

To reduce both the design time and the development cost, processor simulators have been widely used as virtual prototypes of the target processors, in order to validate and evaluate the embedded processor architecture without physically implementing the design at cost and risk. Another usage of processor simulators is to test the correctness and evaluate the quality of the compiler by virtually executing the generated code. Two classes of processor simulators are widely used; they are namely *instruction set simulators* and *micro-architecture simulators*.

Instruction set simulation mimics the behavior of each instruction and models the effect on the target processor state at each step. With the instruction set simulator, the instruction set of a new processor can be functionally verified against any real program. *Micro-architecture simulation*, in contrast, mimics the effect of a micro-architectural design to the instruction execution process. While many retargetable instruction set simulators have been reported [8, 20, 14] , the more relevant micro-architecture simulators, which are capable of modeling the detailed machine features such as cache organization, branch prediction and out-of-order scheduler, have not be equipped with retargetability.

Often times, the instruction set architecture (ISA) and the micro-architecture must be designed together to adapt to a specific or a family of applications, as in the case of ASIP. To find the best solution within the design space, the system architects must perform the so-called *architecture exploration*, in which the simulators are the most important, and have to be developed for each architecture configuration. It implies that instruction set simulation is no longer enough. This is especially true for future high-end embedded processors where the abundance of instruction-level parallelism will make the accuracy of instruction set simulation intolerable. On the other hand, towards the growing complexity of micro-architectural designs, modeling a *detailed* micro-architectural design in simulation becomes more challenging. Manually hand-crafting the micro-architecture simulator for each configuration in architecture exploration does hinder the productivity. This necessitates the notion of retargetable micro-architecture simulators.

1.2 Our Approach

In this thesis, we present techniques that lead to automatic porting of a subset of the *GNU Binutils* package and the *SimpleScalar toolset* [2].

The GNU Binutils package includes a suite of downstream software development tools such as assembler, linker, library manager, profiler, object file examiner and manipulator and C++ dismangler [15]. Partly due to the fact that it is designed to be "portable" and partly because it is free software and accessible to everyone in the world, the GNU Binutils has become a *de facto* standard and there is no doubt that it has already reached the production quality stage. Due to the high complexity of the GNU Binutils package, this thesis aims to cover only a subset of provided tools - the *GNU Binary File Descriptor Library (BFD)* [3] and the *GNU Linker* [4]. As the heart of the GNU Binutils package, the GNU BFD library provides services on which most other downstream tools in the package are highly dependent.

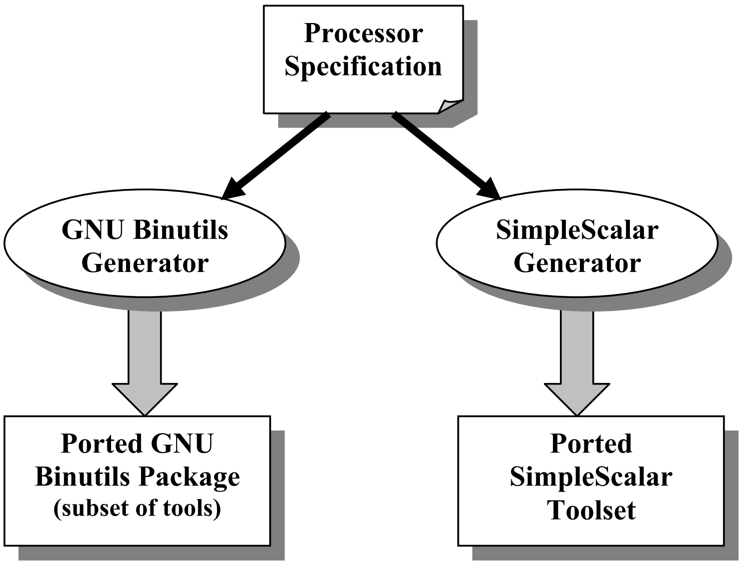


Figure 1.1: Automatic Generation of the GNU Binutils Package and the SimpleScalar Toolset from a Processor Specification

Due to its micro-architectural modeling capability and extensibility, the SimpleScalar toolset developed at University of Wisconsin emerged as the *de facto* standard of modern micro-architecture simulators. With the rich set of micro-architecture components, such as memory, cache, branch predictor and scheduler, the SimpleScalar toolset can model architectures with varying detail, ranging from the simplest unpipelined processors to the out-of-order superscalar architectures. According to the statistics, more than one half of the papers published in the last Annual International Symposium on Computer Architecture (ISCA 2002) have used SimpleScalar tools to evaluate their designs [17]. For this reason, we can believe that the SimpleScalar toolset provides good simulation ability and comprehensive enough to be applied on architecture exploration. Similar to the GNU Binutils package, manually porting the SimpleScalar toolset is not easy and equipping it with retargetability is necessary. From the author's experience, manually porting the tool takes one month of study time of the open-source SimpleScalar infrastructure, which has 30K lines of C code, and one additional month of development time.

The field of retargetable compilation has evolved to the point where an *architecture description language (ADL)* can be used to model processor architecture, and a compiler can be generated automatically from such an architecture specification [6, 7, 5]. Adopting the same approach, we design a processor abstraction model to generically capture relevant architecture information. Taking the processor specification as input, our tools (or generators) are able to automatically port part of the GNU Binutils package and the SimpleScalar toolset (see Figure 1.1).

The work of this thesis can be divided into three parts. First, we need a processor architecture model which is able to capture all necessary information required to port the GNU BFD library, the GNU linker and the SimpleScalar toolset. Second, sophisticated study has to be done to explore the opportunities and the methodology to retarget the GNU BFD library and the GNU linker. Third, the same is done for the SimpleScalar toolset. Our methodology is to enable automatically generating the processor-dependent part of the target tools (GNU Binutils and SimpleScalar) from a processor specification abstracted in our model.

Due to the space limitation in the thesis and the complexity of the retargetting methodology, we have difficulty to include all details of our methods to retarget both toolsets. Anyway, the processor architecture model that is described in this thesis is the complete input of our methodology.

1.3 Contributions

The contributions of this thesis are summarized below:

- We present a formal and abstract processor architecture model to capture the architecture information required for retargetting different software tools, which include the GNU Binutils package and the SimpleScalar toolset in this study. Our model is not tied to the implementation of only one specific software tool (for example, compiler), and the architectural specification in this model is therefore reusable.

- While application binary interface (ABI) information is essential, it has not been captured comprehensively by previous ADLs. Our ABI model is especially capable to capture the information required for retargetting linker editors with dynamic linking support. This capability has not been found in any previous works.

- We made the first step of automatically generating an array of production-quality downstream tools by making the GNU BFD library retargetable. Our technique is verified by successfully retargetting the GNU linker, which is highly dependent on BFD library.

- The popular SimpleScalar toolset is equipped with satisfactory retargetability by our works. The SimpleScalar simulators are also enhanced with additional features such as delayed branches and windowed registers.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the readers with the background of the linking technique. Chapter 3 describes the abstraction model of embedded processors. Chapter 4 describes our methodology of retargetting the GNU Binary File Descriptor Library and linker. It is followed by the methodology of retargetting the SimpleScalar toolset in Chapter 5. Finally, the implementation and experiment results are described in Chapter 6. At the end, Chapter 7 concludes the thesis along with discussion of future research directions. The processor abstraction model discussed in Chapter 3 is supported by an ADL called *Babel*. The data model written in Babel is given in Appendix A, followed by the sample specification of the SPARC and Intel386 processors.

Chapter 2

Background

While compilers are described in many literatures, there is little about linkers. In order to explain our methodology of retargetting the GNU linker, this chapter gives some background material about the linking technique. We focus the discussion on a specific binary file format called *executable and linking format* (ELF).

2.1 Object Files

Binary tools manipulate binary code commonly known as object files. Many formats for object files have been devised, among them the most recent and sophisticated is called *executable and linking format* or *ELF*.

In term of the usages, ELF object files can be categorized into the following four types:

- **Relocatable file**

  It holds contents suitable for linking with other relocatable files or shared libraries. Compiling a source code file generates a relocatable file.

- **Static executable**

  It holds binary code in ready-to-run state. A static executable is produced by linking all associated relocatable files.

- **Dynamic executable**

  It might contain unresolved symbolic references, which will be resolved from the dependent shared libraries at runtime. In order to execute a dynamic executable file, the intervention of the runtime linker is required .

- **Shared library**

  It is used in dynamic linking by those object files on which they depend. A shared library may also depend on other shared libraries.

2.2 Linking

Generally speaking, *linking* is the process that integrates binary information from several pieces into one logical piece. Two kinds of well-known linkers include:

- The *link-editor* processes one or more input object files (relocatable files or shared libraries) to generate one output file (either a relocatable file, an executable, or a shared library).

- The *runtime linker* is responsible for creating process images from the associated dynamic executables and shared libraries at runtime. This is called *dynamic linking*. The runtime linker is also called *interpreter*.

The link-editor is usually what we mean by *linker*. It is one of the downstream tools provided in the GNU Binutils package. Our discussion in the follows will focus on the link-editors.

2.2.1 The Role of Link-editors

The role played by link-editors is illustrated in the compilation process carried by the *GNU GCC* compiler [18]. Being invoked to compile source code files into an executable, GCC will in turn invoke other tools. Figure 2.1 provides an example of compiling three C source files

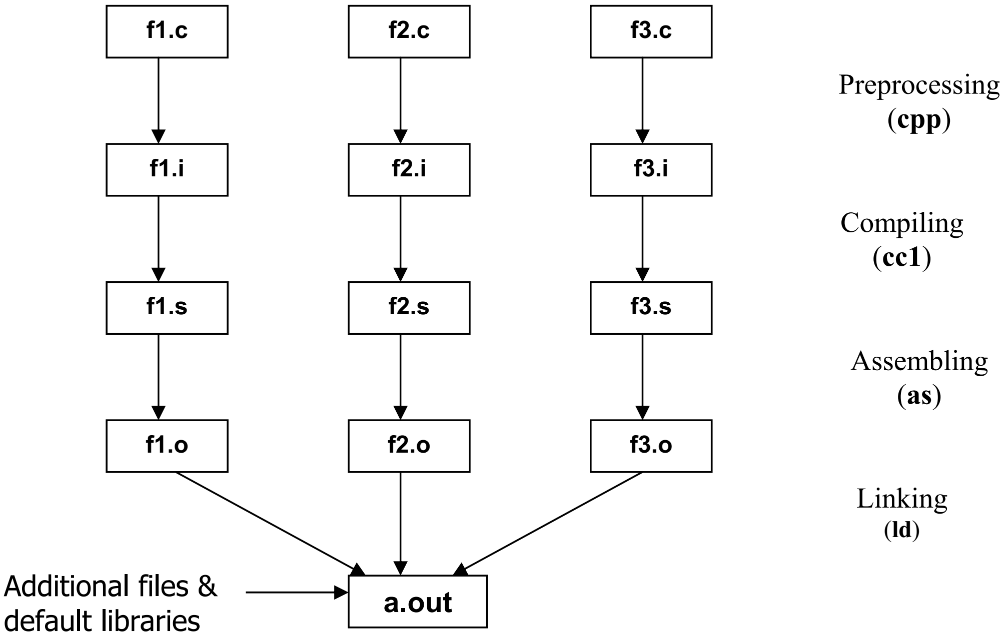


Figure 2.1: Compilation Steps of GNU GCC

into an executable `a.out`. The first set of object files (`t1.o`, `t2.o` and `t3.o`) generated in the process are *relocatable files*, which are not yet ready to be executed. At the last step, the relocatable files, in addition to some default object files and libraries, are linked together to create an executable file, which can be a *static executable* or a *dynamic executable*. The linking process consists of two tasks: *memory address relocation* and *symbol resolution*.

2.2.2 Memory Address Relocation

The assembler assumes the address of any object file starts at zero. When files are linked into one, the sections of the same kind from the input object files will be placed one after the other within a linear address space, as illustrated in Figure 2.2.

During relocation, the link-editor modifies the *base address* of each section. Each entry of the object file keeps a *relative address* from the beginning of its residing section, so that the *absolute address* can be obtained by simple calculation.

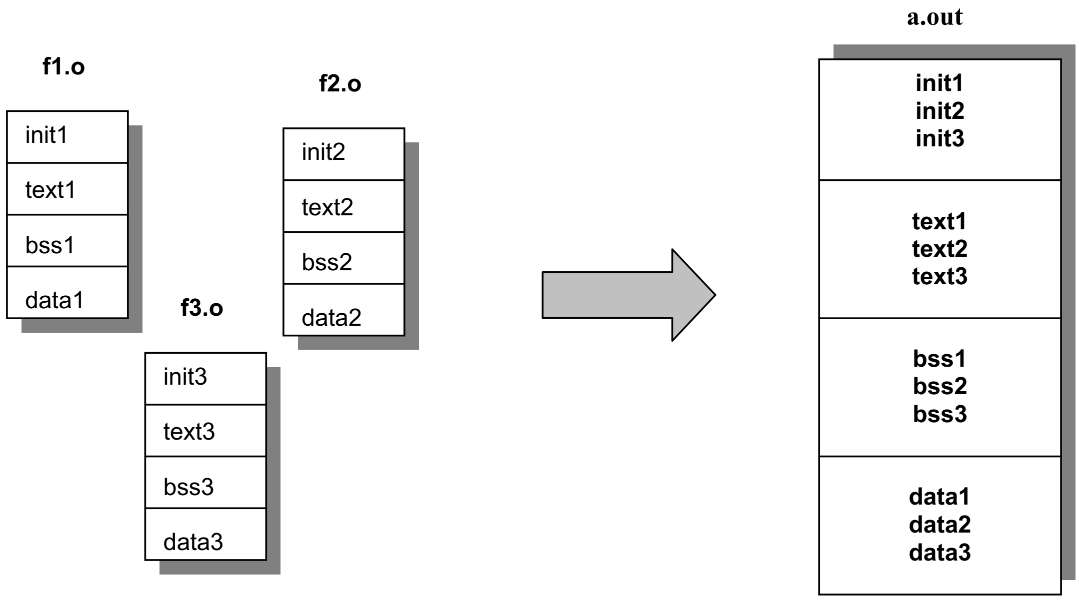


Figure 2.2: Object Files Concatenation

2.2.3 Symbol Resolution

Symbol Table

Each object file has a symbol table. Figure 2.3 gives an example of a symbol table. A symbol table keeps track of all symbols defined or used in the file. In terms of scope and visibility, symbols are categorized as *local*, *global* and *weak*. While local symbols have their scope inside the object file only, global symbols are visible to all other files being processed by the linker. *Weak symbols* are similar to global symbols, except that they have lower priority. For example, a global symbol definition can override a weak symbol definition with the same name.

In term of usage, the global symbols are further classified into *defined symbols*, *external symbols* and *common symbols*. A defined symbol is generated from variable definition in the program. When a symbol is referenced (or being used) but not defined, an undefined symbol, or the so-called external symbol is added to the symbol table. The third case is that a global variable is defined within a file but not initialized and hence not sized. This kind of variable will appear as a common symbol in the symbol table. No storage space is assigned to a common

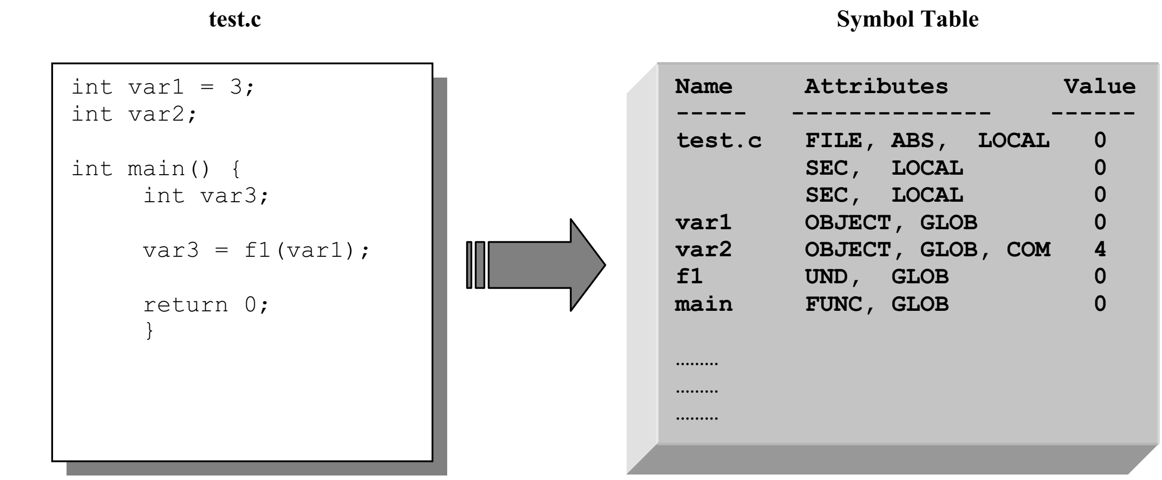


Figure 2.3: An Example of Symbol Table Generation

symbol until the size is known.

Some symbols are generated neither for the data variables nor the functions in the program. For example, a symbol may be generated for the source file name; as illustrated in Figure 2.3. Also, section symbols are generated for relocation purpose. Symbol values that correspond to neither data address nor function address will not be modified in relocation, so this kind of symbol is called *absolute* symbol.

External Symbolic Reference Resolution

External symbols exist to facilitate separate compilation. It is the responsibility of the linker to bind an external symbol to a defined one with the same name from other input files. This process is called *symbol resolution*. The linker will fill the resolved symbol value (address) to the corresponding code contents (see Figure 2.4).

Symbol resolution is usually performed on the fly by adding global symbols from each object file into a hash table. When a global symbol is attempted to add to the hash table and it is found that symbol with the same name exists in the hash table, resolution is performed based on the attributes of the two symbols.

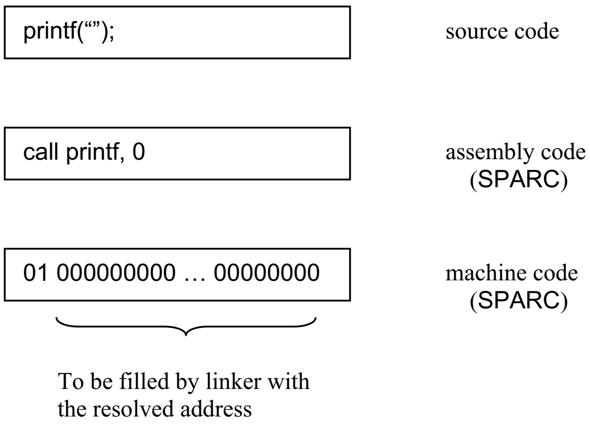


Figure 2.4: Unresolved Symbolic References - An Example in SPARC

2.2.4 ELF Dynamic Linking

An executable is said to be *dynamically linked* with a particular library (shared library) to which it references, if they are not linked together by the link-editor at link-time. The link-editor will write the information about the shared library into the output dynamic executable. At runtime, the runtime linker, the path of which is recorded inside the executable, will link the executable against the shared library on demand during execution.

The so-called dynamic linking often refers to two steps. First, the link-editor generates the dynamic executable and the dependent shared libraries statically. Second, the runtime linker links the dynamic executable and its dependent shared libraries together to generate process image dynamically.

Shared Library

As its name implies, a shared library can be shared by multiple applications. The idea to allow multiple applications to share code sections of a shared library in the process image, but each application must hold its own copy of data sections.

In order to achieve code sharing, the shared libraries must consist of *position-independent code (PIC)*, which means that the code can be loaded at any address in different applications.

The operation of PIC depends on two mechanisms, namely the *global offset table (GOT)* and the *procedure linkage table (PLT)*.

Global Offset Table

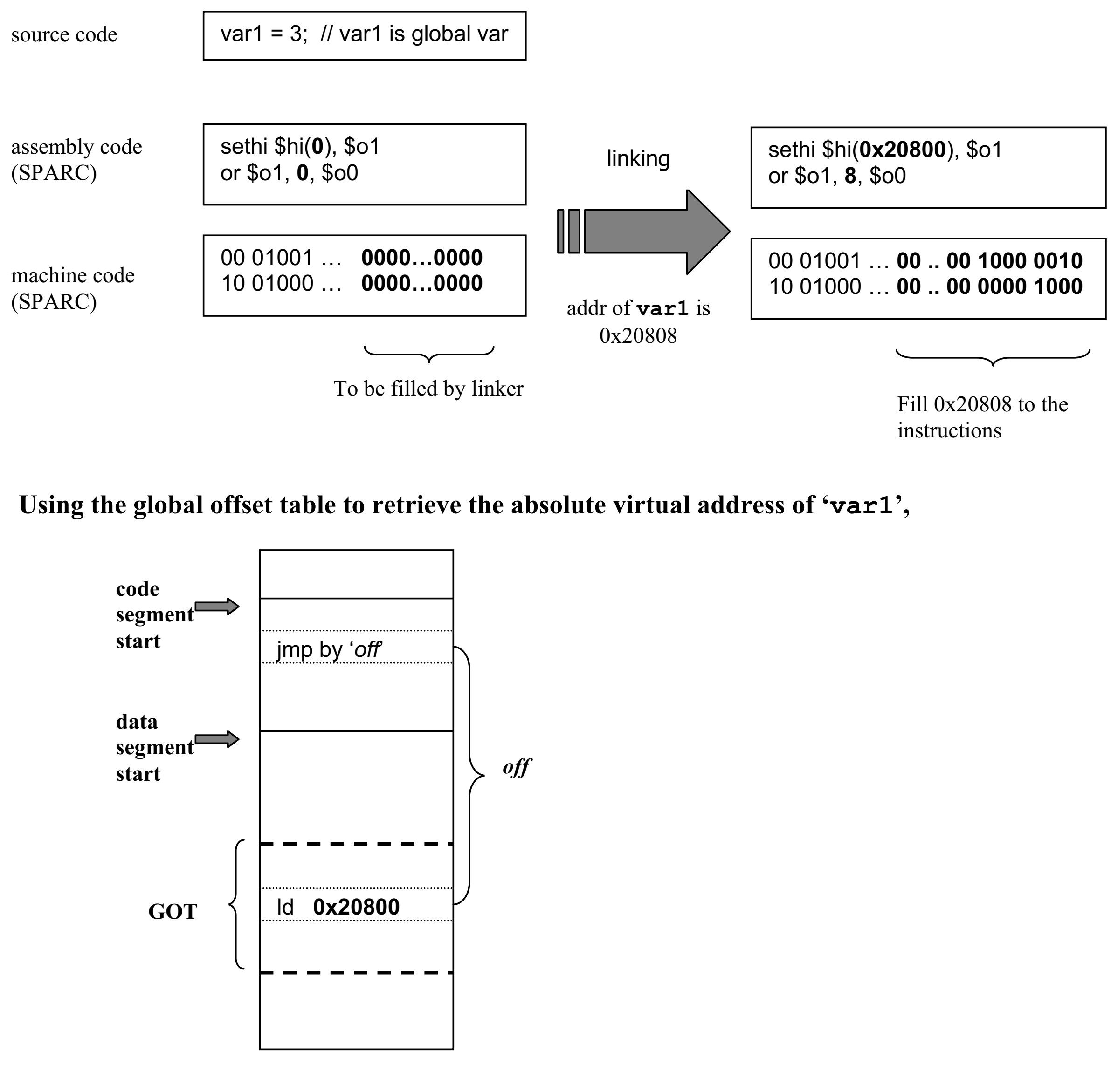


Figure 2.5: The Use of Global Offset Table

No matter whether the variables are defined inside or outside the file, no absolute virtual addresses will be assigned to the global variables before linking. As a result, the code content is incomplete until the linker fills the absolute virtual address values to appropriate places. An example is given in Figure 2.5.

To allow applications to share the code of a shared library, code amendment that might happen during linking must be avoided. With the property that the offset from the instruction to the required data is constant regardless where the program is loaded, ELF object files solve this problem with the global offset table (Figure 2.5). The global offset table is added to the data segment of the shared library. Any instruction requesting the absolute address of a symbol will make through the GOT. Since the offset from the instruction to the corresponding GOT entry can be determined at the static link time and hence a constant, the code segment is sharable. Instead of filling in the absolute virtual addresses to the instructions, the runtime linker fills these to the corresponding GOT entries. Each application has its own GOT, and the GOT therefore holds different absolute virtual address values for the same set of symbols.

Procedure Linkage Table

Resolving the external references to functions defined outside the file causes the same problem. The absolute virtual addresses of external functions are not known until the link-time (the dynamic link-time for shared libraries). An example is given in Figure 2.6 Similar to the GOT redirects to the absolute virtual addresses of global symbols, the procedure linkage table redirects to the absolute locations of function calls.

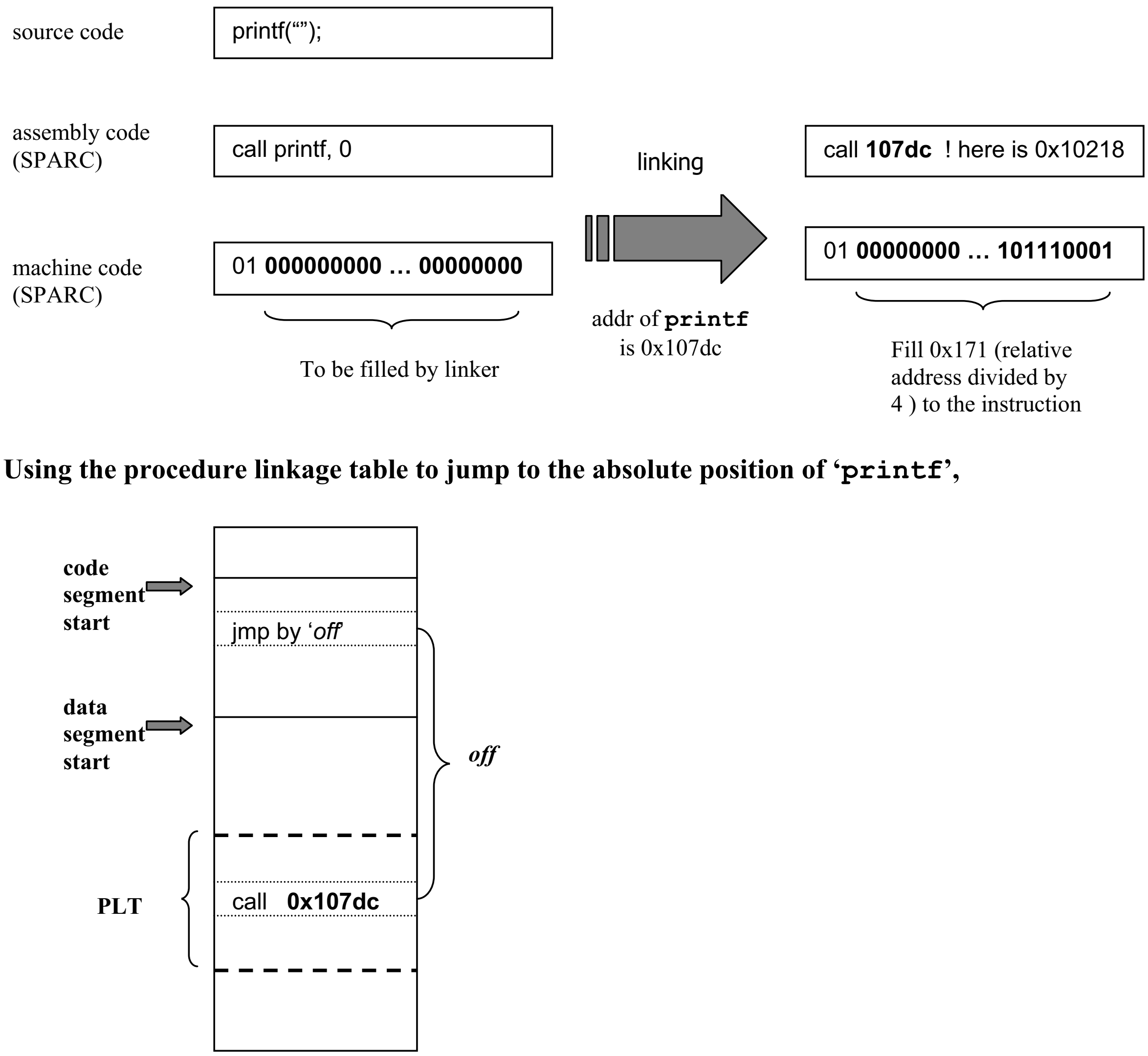


Figure 2.6: The Use of Procedure Linkage Table

Chapter 3

Abstracting Embedded Processors

To retarget the target-dependent tools, a model that holds the abstraction of the target architecture is needed. This chapter will describe the embedded processor model that we use in this thesis. The processor model proposed in this thesis is independent to the architectural description language (ADL) syntax. In this chapter, we will present our processor model with the *formal algorithm notation* (FAN). FAN relies on a type system based on sets. In FAN, we use the notation $\langle\rangle^A$ to represent the power set of A, such that any value of type $\langle\rangle^A$ must be a set of type A values. Similarly, we use the notation $[\,]^A$ to represent the set of all sequences over elements of A, such that any value of type $[\,]^A$ represents a sequence of type A values. Also, we use $\mathcal{Z}$ to represent *integer* value set and $\mathcal{B}$ to represent *boolean* value set.

Definition 1 An Arch *arch : Arch* is a member of

```
Arch = tuple {
  isa          : ISA;
  abi          : ABI;
  }
```

A processor architecture is abstracted in a *Arch* defined in Definition 1. An *Arch* consists of two views: the *instruction-set architecture* (ISA) and the *application binary interface* (ABI);

they are carried by `isa` and `abi` respectively. Presented in section 3.1, the ISA view of the processor architecture includes the register file organization and the instruction-set definition. Following this, section 3.2 presents the ABI view, which captures calling convention information in terms of the register semantics; as well as linking information.

3.1 Instruction Set Architecture

The abstraction of the instruction-set architecture, *ISA*, is defined in Definition 2. The first six members of an *ISA* provide the basic information of the architecture. The `cpu` gives the name of the processor architecture, while the `manufacturer` gives the name of the manufacturer of the processor. The `wordSize` carries the word size in bits (# bits / word), the address size in bits is carried in `addrSize`, and the instruction size in bits is carried in `instrnSize`. The `maxDataAlign` gives the maximum data alignment in bytes. This value can be obtained from ABI standard of the architecture. For example, the floating-point data type has the maximum data alignment value in SPARC - 8 bytes (from [?]). Last but not the least, the `bigEndian` is a boolean value that indicates the byte order of the processor. The rest of the members abstract the register organization and the instruction definition, which will be elaborated in the sequel.

Definition 2 An ISA *isa : ISA* is a member of

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{ISA} = \textbf{tuple}\ \{} \\
\quad & \mathit{cpu} & : \mathit{string}; \\
& \mathit{manufacturer} & : \mathit{string}; \\
& \mathit{wordSize} & : \mathcal{Z}; \\
& \mathit{addrSize} & : \mathcal{Z}; \\
& \mathit{instrnSize} & : \mathcal{Z}; \\
& \mathit{maxDataAlign} & : \mathcal{Z}; \\
& \mathit{bigEndian} & : \mathcal{B}; \\
& \mathit{rfiles} & : \langle\rangle^{\mathit{RegFile}}; \\
& \mathit{ctrls} & : \langle\rangle^{\mathit{CtrlReg}}; \\
& \mathit{instrns} & : \langle\rangle^{\mathit{Instrn}}; \\
& \} &
\end{array}
$$

3.1.1 Register File Model

Definition 3 A register file *rFile : RegFile* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{RegFile = \textbf{tuple}\ \{} \\
\quad gran & : \mathcal{Z}; \\
\quad size & : \mathcal{Z}; \\
\quad win & : WinRegs; \\
\quad uses & : \langle\rangle^{RegUsage}; \\
\quad cells & : [\ ]^{RegCell}; \\
\quad \} &
\end{array}
$$

Definition 3 gives the register file model. A register file is characterized by `size` number of registers, each of which has a bitwidth of `gran`. The `uses` describes the kind of data that the register file may carry and the alignment requirement. The variables representing each register are listed in `cells` in order. The model also supports Sparc-like register window organization. If the register file has windowed characteristics, the organization is specified in `win`, which is a *WinRegs* member. Note that `win` is ignored if the register file is not windowed. In the follows, the register usage, the register cell and the register window model will be explained in detail.

Register Usage

Definition 4 A register usage *rusage : RegUsage* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{RegUsage = \textbf{tuple}\ \{} \\
\quad dataTypeKind & : \{int, unsigned, float, addr\}; \\
\quad bitSize & : \mathcal{Z}; \\
\quad align & : \mathcal{Z}; \\
\quad \} &
\end{array}
$$

Defined in Definition 4, a register usage defines the data type and alignment restriction of a possible kind of operands. The `dataTypeKind` defines the data type of the operands, it can be signed integer (*integer*), unsigned integer (*unsigned*), floating point (*float*) or pointer (*addr*).

The data size in bit is given by `bitSize`. When the size of the operand is greater than the size of one register in the file, it takes more than one consecutive registers to carry the operand. Alignment restriction may apply in such situation, and it is carried in `align`. For example, a register file with 8 32-bit registers may restrict even-odd register alignment when storing a 64-bit value. The `align` data is 2 in this case.

Register Cell

Definition 5 A register cell *rcell : RegCell* is a member of

```
RegCell = tuple {
  name          : string;
  type          : Type;
  }
```

A register cell is a variable that associates with a register, which can be a cell in a register file, a control register or even a control register field. Each register cell associates to one and only one register, and it can be used to refer as the associated register for some other purposes such as expressing the behavior of an instruction.

Like a variable in a program, each register cell is characterized with its `name` and `type`. A *Type* abstracts data type, which is characterized with the data kind (integer or float) and the size.

Register Window

The motivation of modeling the possible windowed organization of the register files stems from SPARC. This is however not a unique feature for SPARC, we do also find the SPARC-like windowed registers in other processor architectures; the Nios embedded processor from ALTERA [**?**] for example.

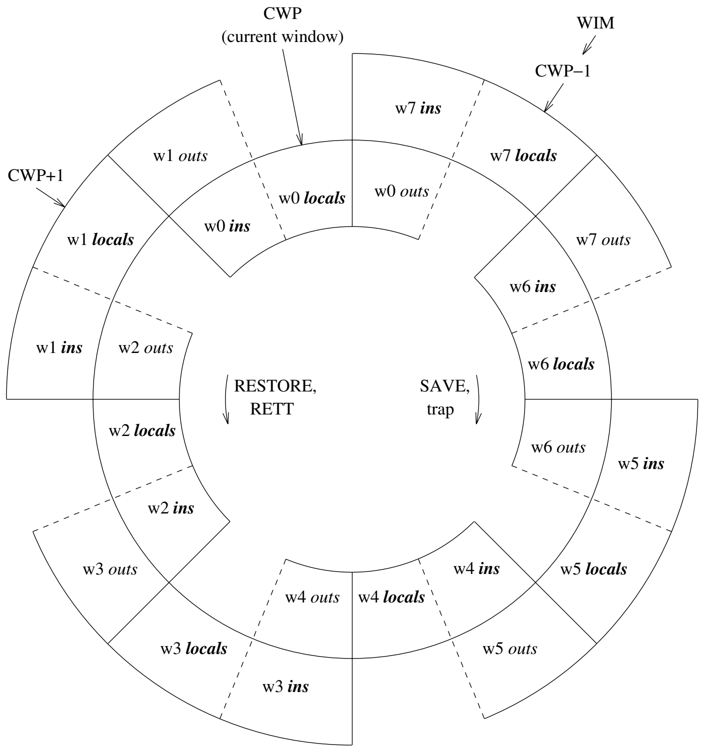


Figure 3.1: The SPARC Windowed Register (from [**?**])

As shown in Definition 6, our register window model is based on the SPARC register window feature with added flexibility. A register window is a set of virtual registers, which is implemented by windows of partially overlapping physical registers. The abstracted view of the physical registers is circular such that the first window overlaps with the last window. An example of SPARC register windows is illustrated in Figure 3.1. In SPARC, a register window consists of 24 registers (three groups of 8 registers, the *out*, *local* and *in* registers). There are 8 windows in Figure 3.1, they are numbered from 0 to 7.

Definition 6 A windowed registers *winRegs : WinRegs* is a member of

WinRegs = **tuple** {
  pos : $\mathcal{Z} \times \mathcal{Z}$;
  depth : $\mathcal{Z}$;
  overlap : $\mathcal{Z}$;
  ptr : *RegCell* $\times$ *Dag*;
  invalid : *RegCell* $\times$ *Dag*;
  saveDir : $\mathcal{B}$;
  overflowCond : *Dag*;
  underflowCond : *Dag*;
  overflowUpdate : *Dag*;
  underflowUpdate : *Dag*;
  }

Each *WinRegs* data is associated with a register file (*RegFile*). The position of the virtual windowed registers in the register file is given in `pos`, which corresponds to one window. For example, the position of the virtual windowed registers in the general purpose register file of SPARC is <8:31>. The `depth` corresponds to the number of windows that make up the physical registers and the `overlap` specifies the number of overlapping registers between consecutive windows. According to the order of the physical registers, the windows are numbered from 0 to (#window - 1). There should have two control registers that assists the operation of the register window at runtime. One carries the number of the current window, the corresponding register cell and its initial value before execution is identified by `ptr`. It is responsible by the CWP register in SPARC. The other one determines whether a window overflow or underflow trap is generated by the window shifting request, the corresponding register cell and its initial setup value is identified by `invalid`. It is responsible by the WIM in SPARC. An initial value is carried as an expression, which is abstracted in a type called *Dag*. Different to the current window pointer, which is assumed to carry the current window ID in any architecture, the interpretation of the window invalid register may be architecture-dependent and must be specified in the model. The register window feature is to facilitate the procedure call and return. The *SAVE* instruction would shift the register window by 1 for procedure call, while the *RESTORE* instruction would shift the register window by 1 in the opposite direction for procedure return. The shifting direction of SAVE is identified in `saveDir`. If the SAVE instruction causes shifting to greater ID of window (e.g. anti-clockwise in Figure 3.1, `saveDir` is true. Otherwise, `saveDir` is false. The SAVE and the RESTORE instructions will identify themselves in the instruction definition model.

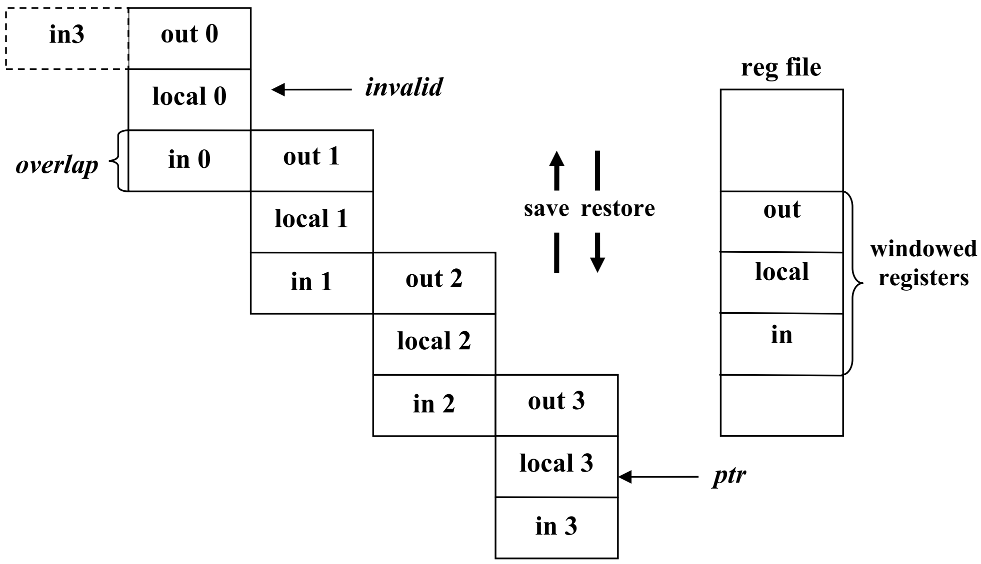


Figure 3.2: A Depth-4-register Window

Using a depth-4-register window in Figure 3.2, we explain the operation of the register window. The current window pointer is initially pointing to window 3. The invalid pointer is therefore initially pointing to window 0 because it is overlapping with the window 3, which will be being used when the SAVE instruction attempts to shift to this window. The invalid pointer is always pointing to the first invalid window. When a SAVE instruction attempts to shift from window 1 to window 0 (invalid set), window overflow trap is generated. The overflow trap saves the *in* and *local* registers of window 3 to the stack and moves the invalid pointer to window 3. When the overflow trap returns, the window 0 becomes available. The same overflow handling action will be taken if another SAVE instruction attempts to shift from window 0 to window 3. When the procedure call returns, the RESTORE instruction shifts the window in the opposite direction. Assuming that the invalid pointer is pointing to window 3, the window underflow is generated when a RESTORE instruction attempts to shift from window 2 to window 3. The underflow trap restores the *in* and *local* registers that were saved by the previous overflow trap from the stack to the corresponding physical registers. The underflow trap also moves the invalid pointer to window 0. When the underflow trap returns, the RESTORE is allowed to shift to window 3.

As mentioned, the register cell indicated in `ptr` will always carry the current window ID. The interpretation and the use of the register cell indicated in `invalid` is however architecture-dependent. This is concluded from our observation when comparing different architectures using windowed registers. Nevertheless, the purpose served by the invalid is unique in any architecture. Our model describes the interpretation of the window invalid register value with its initial value, the window overflow/underflow condition expressed in terms of register value (`overflowCond` and `underflowCond`), and the value update that is done to the register by the window overflow/underflow trap (`overflowUpdate` and `underflowUpdate`). The SPARC window invalid register, WIM(window invalid mask), is described in Figure 3.3.

Window invalid register: `WIM`

W31	W30	W29	……	W1	W0
31	30	29		1	0

- There is an active state bit in the `WIM` for each register window
- `WIM`[n] corresponds to the window n
- There is only one state bit in active in the `WIM` at any time, it implies that the corresponding window is invalid.

Figure 3.3: The SPARC Window Invalid Register - `WIM`

Example 1 shows the specification of the SPARC general-purpose register file.

Example 1 *The SPARC general-purpose register file description*

```
sparcGPR = ⟨
  gran = 32,
  size = 32,
  win = ⟨
    pos = ⟨8,31⟩,
    depth = 32,
    overlap = 8,
    ptr = ⟨cwp,'31'⟩,
    invalid = ⟨wim,'0x1'⟩,
    saveDir = false,
    overflowCond = '((wim >> cwp) &1) != 0',
    underflowCond = '((wim >> cwp) &1) != 0',
    overflowUpdate = '((wim == 0x1)?0x80000000:(wim>>1))',
    underflowUpdate = '((wim == 0x80000000)?0x1:(wim<<1))'
    ⟩,
  uses = {
    u8 = ⟨dataTypeKind = unsigned,bitSize = 8,align = 1⟩,
    i8 = ⟨dataTypeKind = integer,bitSize = 8,align = 1⟩,
    u16 = ⟨dataTypeKind = unsigned,bitSize = 16,align = 1⟩,
    i16 = ⟨dataTypeKind = integer,bitSize = 16,align = 1⟩,
    u32 = ⟨dataTypeKind = unsigned,bitSize = 32,align = 1⟩,
    i32 = ⟨dataTypeKind = integer,bitSize = 32,align = 1⟩,
    u64 = ⟨dataTypeKind = unsigned,bitSize = 64,align = 2⟩,
    i64 = ⟨dataTypeKind = integer,bitSize = 64,align = 1⟩,
    addr = ⟨dataTypeKind = addr,bitSize = 32,align = 1⟩
    },
  cells = [
    g0, g1, g2, g3, g4, g5, g6, g7,
    l0, l1, l2, l3, l4, l5, l6, l7,
    i0, i1, i2, i3, i4, i5, i6, i7,
    o0, o1, o2, o3, o4, o5, o6, o7
    ]
  ⟩
```

3.1.2 Control Register Model

Definition 7 A control register *creg : CtrlReg* is a member of

$$
\begin{array}{ll}
\mathit{CtrlReg} = \textbf{tuple}\ \{ & \\
\quad \mathit{kind} & : \{\mathit{none}, \mathit{pc}, \mathit{npc}, \mathit{winptr}, \mathit{wininvalid}\}; \\
\quad \mathit{cell} & : \mathit{RegCell}; \\
\quad \mathit{fields} & : \langle\rangle^{\mathit{CtrlRegField}}; \\
\quad \} &
\end{array}
$$

Definition 8 A control register field *cregField : CtrlRegField* is a member of

$$
\begin{array}{ll}
\mathit{CtrlRegField} = \textbf{tuple}\ \{ & \\
\quad \mathit{pos} & : \mathbb{Z} \times \mathbb{Z}; \\
\quad \mathit{creg} & : \mathit{CtrlReg}; \\
\quad \} &
\end{array}
$$

The control register model is given in Definition 7 and Definition 8. A control register is modeled with its kind, the register cell and meaningful fields. If the control register is a program counter, a next program counter, a register window pointer or a register window invalid pointer, it is required to identify itself in `kind`. The register cell associated to the control register is given in `cell`, in which we can find the name and the size of the register. If the control register consists of fields that hold different processor status information, the hierarchical organization can be described by `fields`, which is a list of control register fields. A control register field is modeled by a *CtrlReg* and its position at the parent.

Example 2 shows the specification of the SPARC processor state register (PSR), which is described in Figure 3.4.

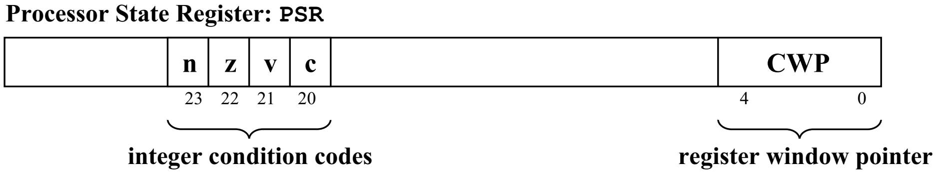


Figure 3.4: The SPARC Processor State Register - `PSR`

Example 2 *The SPARC processor state register (PSR) description*

$$
\begin{array}{l}
\mathit{sparcPSR} = \langle \\
\quad \mathit{kind} = \mathit{none}, \\
\quad \mathit{cell} = \mathit{psr}, \\
\quad \mathit{fields} = \{ \\
\qquad \langle pos = \langle 23,23 \rangle, creg = \langle kind = none, cell = icc\_n \rangle \rangle, \\
\qquad \langle pos = \langle 22,22 \rangle, creg = \langle kind = none, cell = icc\_z \rangle \rangle, \\
\qquad \langle pos = \langle 21,21 \rangle, creg = \langle kind = none, cell = icc\_v \rangle \rangle, \\
\qquad \langle pos = \langle 20,20 \rangle, creg = \langle kind = none, cell = icc\_c \rangle \rangle, \\
\qquad \langle pos = \langle 4,0 \rangle, creg = \langle kind = winptr, cell = cwp \rangle \rangle \\
\qquad \} \\
\quad \rangle
\end{array}
$$

3.1.3 Instruction Model

Each instruction in ISA is modeled with an *Instrn* data as defined in Definition 9. An instruction is characterized by its binary encoding format, assembly format and behavior. In the following, we will discuss each of these by taking the SPARC instructions as modeling example.

Definition 9 An instruction *instrn : Instrn* is a member of

$$
\begin{array}{ll}
\mathit{Instrn} = \textbf{tuple}\ \{ & \\
\quad \mathit{asmFormat} & : \mathit{string}; \\
\quad \mathit{opfmt} & : \mathit{opFormat}; \\
\quad \mathit{opcodes} & : [\,]^{\mathbb{Z}}; \\
\quad \mathit{beh} & : \mathit{InstrnBeh}; \\
\quad \} &
\end{array}
$$

Instruction Format

The *instruction format* defines the binary encoding of an instruction. Instructions of an architecture usually share several common formats. Figure 3.5 summarizes the instruction formats of SPARC. An instruction format consists of an sequence of *instruction fields*. An instruction field is the smallest meaningful unit in the instruction, and it is characterized by its role playing in the instruction semantics. Each instruction field falls into one of the 3 categories: *opcode*, *register file index* and *immediate value*. The combination of the opcode fields is used to identify an instruction type, while the register fields and immediate fields are operands or destinations of the instruction behavior. Instead of abstracting the instruction formats explicitly, which is defining each format as a sequence of instruction fields, our model abstracts the formats with two parts: 1) identifying the opcode format in terms of the position of the opcode fields and defining the value combination of the instruction, and 2) identifying the position and the interpretation of the register fields and immediate fields, which will be operands or destinations of the instruction behavior.

Format 1 (*op* = 1): CALL

op	disp30
31	*29* … *0*

Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)

op	rd		op2	imm22
op	a	cond	op2	disp22
31	*29*	*28*	*24*	*21* … *0*

Format 3 (*op* = 2 or 3): Remaining instructions

op	rd	op3	rs1	i=0	asi	rs2
op	rd	op3	rs1	i=1	simm13	
op	rd	op3	rs1	opf		rs2
31	*29*	*24*	*18*	*13*	*12*	*4* … *0*

Figure 3.5: The Instruction Format of 32-bit SPARC (from [**?**])

An instruction field is characterized by both of the location and the interpretation by an instruction. To make the modeling easier, we split these two concepts in the instruction field abstraction. The location of the instruction field is abstracted by an *instruction field accessor* as defined in Definition 10. An instruction field accessor is used to extract the value of the instruction field from an instruction. Example 3 gives t he specification of the instruction field accessors of SPARC.

Definition 10 An instruction field accessor *acc : InstrnFieldAccessor* is a member of

$$
\begin{array}{ll}
\mathit{InstrnFieldAccessor} = \textbf{tuple}\ \{ & \\
\quad \mathit{name} & : \mathit{string}; \\
\quad \mathit{pos} & : \mathbb{Z} \times \mathbb{Z}; \\
\quad \} &
\end{array}
$$

Example 3 *The SPARC instruction field accessors*

$$
\begin{array}{lll}
\mathit{sparcFieldAccessors} = \ \{ & & \\
\quad a\_op & = \langle name = \text{"}op\text{"}, pos = \langle 31, 30 \rangle \rangle & , \\
\quad a\_disp30 & = \langle name = \text{"}disp30\text{"}, pos = \langle 29, 0 \rangle \rangle & , \\
\quad a\_rd & = \langle name = \text{"}rd\text{"}, pos = \langle 29, 25 \rangle \rangle & , \\
\quad a\_op2 & = \langle name = \text{"}op2\text{"}, pos = \langle 24, 22 \rangle \rangle & , \\
\quad a\_imm22 & = \langle name = \text{"}imm22\text{"}, pos = \langle 21, 0 \rangle \rangle & , \\
\quad a\_a & = \langle name = \text{"}a\text{"}, pos = \langle 29, 29 \rangle \rangle & , \\
\quad a\_cond & = \langle name = \text{"}cond\text{"}, pos = \langle 28, 25 \rangle \rangle & , \\
\quad a\_disp22 & = \langle name = \text{"}disp22\text{"}, pos = \langle 21, 0 \rangle \rangle & , \\
\quad a\_op3 & = \langle name = \text{"}op3\text{"}, pos = \langle 24, 19 \rangle \rangle & , \\
\quad a\_rs1 & = \langle name = \text{"}rs1\text{"}, pos = \langle 18, 14 \rangle \rangle & , \\
\quad a\_i & = \langle name = \text{"}i\text{"}, pos = \langle 13, 13 \rangle \rangle & , \\
\quad a\_asi & = \langle name = \text{"}asi\text{"}, pos = \langle 12, 5 \rangle \rangle & , \\
\quad a\_rs2 & = \langle name = \text{"}rs2\text{"}, pos = \langle 4, 0 \rangle \rangle & , \\
\quad a\_simm13 & = \langle name = \text{"}simm13\text{"}, pos = \langle 12, 0 \rangle \rangle & , \\
\quad a\_opf & = \langle name = \text{"}opf\text{"}, pos = \langle 13, 5 \rangle \rangle & \\
\quad \} & &
\end{array}
$$

With the instruction field accessor defining the location, the instruction field is then built upon the corresponding field accessor by adding the semantics. The opcode fields and the register/immediate fields are handled with different ways.

The opcode fields are not defined individually in the model. Instead, the *opcode format* which consists of a sequence of field accessor is defined. Defined in Definition 11, an opcode format gives the location of the opcode fields. The identification of an instruction can further be defined by an instruction opcode format and the values at each field. They are given by the `opfmt` and the `opcodes` respectively in an *Instrn*, the definition of which can be found in Definition 9. Example 4 gives the specification of the instruction opcode formats of SPARC.

Definition 11 An instruction opcode format *opfmt : InstrnOpFmt* is a member of

```
InstrnOpFmt = tuple {
  opcodes        : [ ]^InstrnFieldAccessor;
  }
```

Example 4 *The SPARC instruction opcode formats*

```
sparcOpFmts = {
  opf1A = [ a_op ],
  opf2A = [ a_op, a_op2 ],
  opf2B = [ a_op, a_a, a_cond, a_op2 ],
  opf3A = [ a_op, a_op3, a_i ],
  opf3C = [ a_op, a_op3, a_opf ],
  opf3D = [ a_op, a_cond, a_op3, a_i ]
  }
```

The register fields and the immediate fields are defined explicitly by adding the semantic information upon the corresponding field accessor. The instruction field model defined in Definition 12 is used to abstract the register/immediate fields. The `kind` identifies whether it is a register field or an immediate field. The position and the name of field is then given by the corresponding field accessor (`fieldAccessor`). If it is a register field, the carried value is interpreted as a register file index, so the referenced register file has to be given by `rfile`. If it is an immediate field, the `signed` indicates whether the immediate value has to be sign-extended before being used as operand. Example 5 gives the specification of the required instruction field definitions in the SPARC model.

Definition 12 An instruction field *field : InstrnField* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{InstrnField} = \textbf{tuple}\ \{} \\
\quad \mathit{kind} & : \{\mathit{register}, \mathit{immediate}\}; \\
\quad \mathit{accessor} & : \mathit{InstrnFieldAccessor}; \\
\quad \mathit{rfile} & : \mathit{RegFile}; \\
\quad \mathit{signed} & : \mathcal{B}; \\
\quad \} &
\end{array}
$$

Example 5 *The SPARC instruction fields*

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{sparcFields} = \ \{} \\
\quad f\_disp30\_s & = \langle kind = immediate, accessor = a\_disp30, signed = true\rangle & , \\
\quad f\_disp22\_s & = \langle kind = immediate, accessor = a\_disp22, signed = true\rangle & , \\
\quad f\_imm22 & = \langle kind = immediate, accessor = a\_imm22, signed = false\rangle & , \\
\quad f\_simm13 & = \langle kind = immediate, accessor = a\_simm13, signed = false\rangle & , \\
\quad f\_simm13\_s & = \langle kind = immediate, accessor = a\_simm13, signed = true\rangle & , \\
\quad f\_rs1\_g & = \langle kind = register, accessor = a\_rs1, rfile = sparcGPR\rangle & , \\
\quad f\_rs2\_g & = \langle kind = register, accessor = a\_rs2, rfile = sparcGPR\rangle & , \\
\quad f\_rs1\_f & = \langle kind = register, accessor = a\_rs1, rfile = sparcFPR\rangle & , \\
\quad f\_rs2\_f & = \langle kind = register, accessor = a\_rs2, rfile = sparcFPR\rangle & , \\
\quad f\_rd\_g & = \langle kind = register, accessor = a\_rd, rfile = sparcGPR\rangle & , \\
\quad f\_rd\_f & = \langle kind = register, accessor = a\_rd, rfile = sparcFPR\rangle & \\
\quad \} & &
\end{array}
$$

Instruction Behavior

Each instruction has an assigned *behavior* which abstracts the operation performed by the instruction. An instruction behavior is abstracted in a *Instrn*, which is defined in Definition 13. For some special-purpose instructions, the model requires the instruction identify its characteristics by `kind`. This includes register window SAVE instructions (`winsave`), register window RESTORE instructions (`winrestore`), delayed branch instructions (`delayed`) and system call trap instructions (`trap`). If the instruction behavior falls into the kind of either `winsave` or `winrestore`, the associated register file, which contains the windowed registers, must be specified by `win`. The `dst` and the `srcs` identify the instruction fields which are taken as the destination and operands respectively. The destination is normally a register field while a source can be a register field or an immediate fields. Our model assumes there is at most one destination in the behavior operation. In addition to the special behavior characteristics implied by `kind`, the behavior operation is further described by a function given by `beh`. The `beh` function takes the `srcs` as parameters in its sequence order. The taken operand values at execution are described by each instruction field. If a source is a register field, the operand value is taken from the corresponding register file at the index specified by the field. If a source is an immediate field, the operand value is directly taken from the field and sign-extended if it is specified in the *InstrnField* abstraction. If the `dst` is given, the return value of the `beh` function is written to the corresponding register file at the index specified by the destination register field.

Definition 13 An instruction behavior *instrn : Instrn* is a member of

$$
\begin{array}{ll}
\mathit{InstrnBeh} = \textbf{tuple}\ \{ & \\
\quad \mathit{kind} & : \{\mathit{none}, \mathit{winsave}, \mathit{winrestore}, \mathit{delayed}, \mathit{trap}\}; \\
\quad \mathit{win} & : \mathit{RegFile}; \\
\quad \mathit{dst} & : \mathit{InstrnField}; \\
\quad \mathit{srcs} & : [\,]^{\mathit{InstrnField}}; \\
\quad \mathit{beh} & : \mathit{Function}; \\
\quad \} &
\end{array}
$$

Assembly Format

The `asmFormat` of the *Instrn* defines the assembly syntax of the instruction. Instead of following the classical way that specifying the syntax of all instructions collectively with BNF formalism, we choose a more natural approach that specifying the assembly syntax of an instruction in terms of the usage pattern. The assembly format given in the model is a template which is used to disassemble the binary. No restriction is applied on the assembly syntax style, but some rules should be followed.

The assembly syntax of an instruction is a mnemonic followed by the argument pattern. The destination and operands are identified by the special character % for register fields and # for immediate fields. The % and # characters are followed by a number. When the number is 0, it is referring to the destination field which is as specified by `dst` of the instruction behavior. When the number is non-zero, it is referring to a source field in `srcs` by assuming that the fields in `srcs` are numbered from 1.

Example 6 shows the specification of 3 SPARC instructions - load signed byte (`ldsb`), branch if not equal (`bne`) and register window SAVE (`save`). The description of the instructions can be found in [**?**]. The `ldsb` instruction gets a signed byte value from the address calculated by adding the values obtained from the two specified source registers, and writes it to the destination register. The `bne` is a delayed branch, which is the instruction will be followed by a delay slot. The branch decision is made from checking whether the state value holding in `icc_z` is not equal to zero. If this is a delayed branch instruction, the model assumes that the behavior operation includes appropriate update of the PC (program counter) and NPC (next program counter). After the execution of a delayed branch, the advance of PC and NPC can be skipped. The last instruction in the example is a register window SAVE instruction. The register window shifting operation is implicit and should not be specified in the `beh` function in the instruction behavior. However, operations other than shifting the register window should be specified in the `beh` function. We assume that the source values, if any of them comes from register fields, are obtained before window shifting and the returned value, if there is any, is written to the destination after window shifting.

Example 6 *The SPARC instructions - load signed byte (*`ldsb`*), branch if not equal (*`bne`*) and register window SAVE (*`save`*)*

```
ldsb = ⟨
  asmFormat = "ldsb [%1+%2], %0",
  opfmt = opf3A,
  opcodes = [ 0x3, 0x9, 0x0 ],
  beh = ⟨
    kind = none,
    dst = f_rd_g,
    srcs = [f_rs1_g, f_rs2_g],
    beh = func( src1 : ℤ, src2 : ℤ ) : ℤ {
            return *(src1+src2) ;
            }
  ⟩
⟩

bne = ⟨
  asmFormat = "bne #1",
  opfmt = opf2B,
  opcodes = [ 0x0, 0x0, 0x9, 0x2 ],
  beh = ⟨
    kind = delayed,
    srcs = [f_disp22_s],
    beh = func( disp : ℤ ) : ∅ {
            pc = npc;
            npc = (!icc_z)?(pc + (disp << 2)):(npc+4);
            }
  ⟩
⟩

save = ⟨
  asmFormat = "save %1,%2,%0",
  opfmt = opf3A,
  opcodes = [ 0x2, 0x3c, 0x0 ],
  beh = ⟨
    kind = winsave,
    win = sparcGPR,
    dst = f_rd_g,
    srcs = [f_rs1_g, f_rs2_g],
    beh = func( src1 : ℤ, src2 : ℤ ) : ℤ {
            return src1+src2 ;
            }
  ⟩
⟩
```

3.2 Application Binary Interface

Definition 14 An ABI *abi : ABI* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{ABI = \textbf{tuple}\ \{} \\
\quad e\_mach & : \mathcal{Z}; \\
\quad maxPageSize & : \mathcal{Z}; \\
\quad pageSize & : \mathcal{Z}; \\
\quad startAddr & : \mathcal{Z}; \\
\quad dynLinkerPath & : string; \\
\quad reloca & : \mathcal{B}; \\
\quad localSymPrefixes & : \langle\rangle^{string}; \\
\quad zero & : \langle\rangle^{RegCell}; \\
\quad relocs & : \langle\rangle^{Reloc}; \\
\quad dyns & : \langle\rangle^{Dyn}; \\
\quad got & : Got; \\
\quad plt & : Plt; \\
\quad stack & : Stack; \\
\quad memUses & : \langle\rangle^{memUsage}; \\
\quad convs & : \langle\rangle^{InstrnConvs}; \\
\quad \} &
\end{array}
$$

Definition 14 abstracts the application binary interface, *ABI*. The first member (`e_mach`) gives the ELF machine value of the architecture. A processor architecture that uses ELF as the object file format is assigned an unique value to identify itself in the ELF files. The `maxPageSize` holds the maximum page size in bytes. The `startAddr` provides the start virtual address of the text segment. The `dynLinkerPath` is a string that holds the path of the dynamic linker (e.g. `"/usr/lib/ld.so.1"`) in the system at which the executable file will be running. A dynamic linker is required for supporting dynamic linking at runtime. Relocation entries of a processor architecture may be carried in one of the two different formats in Figure 3.6. The `reloca` is a boolean that indicates which format is adopted by the processor architecture. The architecture uses the `addend` in a relocation entry to carry the partial linking value if `reloca` is true. Otherwise, the value is stored in the relocated field.

The `localSymbPrefixes` carries a set of local symbol prefixes. Typically, the local symbols in an ELF file start with *.L*, *..* or *_L_*. However, some architecture may support

```
Elf32_Rel = tuple {
  r_offset        : Elf32_Addr;
  r_info          : Elf32_Word;
  r_addend        : Elf32_Sword;
  }
```

Figure 3.6: ELF Relocation Entry

```
ELF32_R_SYM = func( r_info : Elf32_Word ) : ℤ {
  return r_info >> 8 ;
  }

ELF32_R_TYPE = func( r_info : Elf32_Word ) : ℤ {
  return r_info &0xf ;
  }
```

Figure 3.7: Definition of the `r_info`

more than these. For example, the i386 architecture also starts local symbols with *.X*. The `localSymbPrefixes` data in the *ABI* of its model has the value of {*".L", "..", "_L_", ".X"*}.

The `zero` carries a list of registers that always hold the value '0'.

The subsequent members of an *ABI* abstract relocations, the dynamic section (`.dynamic`) in the binary file, the global offset table (GOT), the procedure linkage table (PLT), the initial stack layout and the memory usage. These will be discussed in the following subsections.

3.2.1 Relocation Model

In object files that will be processed by a linker (either a link-editor or a runtime linker), there will have some relocation entries in the content. Relocation entries instruct the linker about how to modify the file content in the linking process. Each relocation entry corresponds to one task for one purpose. A relocation entry is carried in a *Elf32_Rel* as defined in Figure 3.6 [19]. In a *Elf32_Rel*, the `r_offset` value gives the location at which to apply the relocation action. The `r_info` value is defined in terms of the symbol table index of the associated symbol and the type of the relocation to apply, as in Figure 3.7. The relocation type value (`ELF32_R_TYPE`), which is encrypted inside the `r_info` member, is processor-specific. Each relocation type defines a calculation formula by using which the linker change the file content accordingly. In other words, the relocation type value of a relocation entry implies the action that should be taken by the linker. The `r_addend` member carries the result value of partial linking. Some architecture ignores the `r_addend` and stores this value implicitly in the location as indicated by the `r_offset` value.

Notation	**Explanation**
`A`	The addend
`B`	The base address at which a shared library object has been loaded into memory during execution
`G`	The offset between the reloctable field and the GOT entry, at which the address of the relocation symbol will reside during execution
`GOT`	The address of the global offset table (GOT)
`L`	The address of the PLT entry of the symbol
`P`	The address of the relocated instruction (current PC)
`S`	The value of the symbol, the index of which is provided by the relocation entry

Table 3.1: Symbol Notation for Table 3.3 and Table 3.2

Name	Value	Field	Calculation	Other action
R_386_NONE	0	none	none	
R_386_32	1	<31:0>	S+A	
R_386_PC32	2	<31:0>	S+A−P	
R_386_GOT32	3	<31:0>	G+A−P	- creates GOT if it doesn't exist - creates a new GOT entry
R_386_PLT32	4	<31:0>	L+A−P	- creates PLT if it doesn't exist - creates a new PLT entry
R_386_COPY	5	none	none	- copies the value of the associated symbol from the shared library
R_386_GLOB_DAT	6	<31:0>	S	- sets the designated symbol addr. to the GOT entry
R_386_JMP_SLOT	7	<31:0>	S	- modifies the PLT entry to transfer control to the designated symbol addr.
R_386_RELATIVE	8	<31:0>	B+A	- updates a relative addr.
R_386_GOTOFF	9	<31:0>	S+A−GOT	
R_386_GOTPC	10	<31:0>	GOT+A−P	

Table 3.2: i386 Processor (32-bit) Relocation Types

Name	**Value**	**Field**	**Calculation**	**Other action / Notes**
`R_SPARC_NONE`	0	none	none	
`R_SPARC_8`	1	*V*-<7:0>	`S+A`	
`R_SPARC_16`	2	*V*-<15:0>	`S+A`	
`R_SPARC_32`	3	*V*-<31:0>	`S+A`	
`R_SPARC_DISP8`	4	*V*-<7:0>	`S+A-P`	
`R_SPARC_DISP16`	5	*V*-<15:0>	`S+A-P`	
`R_SPARC_DISP32`	6	*V*-<31:0>	`S+A-P`	
`R_SPARC_WDISP30`	7	*V*-<29:0>	`(S+A-P)>>2`	
`R_SPARC_WDISP22`	8	*V*-<21:0>	`(S+A-P)>>2`	
`R_SPARC_HI22`	9	*T*-<21:0>	`(S+A)>>10`	
`R_SPARC_22`	10	*V*-<21:0>	`S+A`	
`R_SPARC_13`	11	*V*-<12:0>	`S+A`	
`R_SPARC_LO10`	12	*T*-<12:0>	`(S+A)&0x3ff`	
`R_SPARC_GOT10`	13	*T*-<12:0>	`G&0x3ff`	- creates GOT if it doesn't exist - creates a new GOT entry
`R_SPARC_GOT13`	14	*V*-<12:0>	`G`	- creates GOT if it doesn't exist - creates a new GOT entry
`R_SPARC_GOT22`	15	*T*-<21:0>	`G>>10`	- creates GOT if it doesn't exist - creates a new GOT entry
`R_SPARC_PC10`	16	*T*-<12:0>	`(S+A-P)&0x3ff`	
`R_SPARC_PC22`	17	*V*-<21:0>	`(S+A-P)>>10`	
`R_SPARC_WPLT30`	18	*V*-<29:0>	`(L+A-P)>>2`	- creates PLT if it doesn't exist - creates a new PLT entry
`R_SPARC_COPY`	19	none	none	- copies the associated symbol value from the shared library
`R_SPARC_GLOB_DAT`	20	*V*-<31:0>	`S+A`	- sets the designated symbol addr. to the GOT entry
`R_SPARC_JMP_SLOT`	21	none	see notes	- modifies the PLT entry to transfer control to the designated symbol address
`R_SPARC_RELATIVE`	22	*V*-<31:0>	`B+A`	- updates a relative addr.
`R_SPARC_UA32`	23	*V*-<31:0>	`S+A`	- refers to an unaligned word

Table 3.3: SPARC Processor (32-bit) Relocation Types

In the ABI model as defined in Definition 14, the relocation information is abstracted in the `relocs` data as a set of *Reloc*, which is defined in Definition 15. A *Reloc* is an abstraction of a relocation type. In the rest of the description, we will take the i386 and the SPARC as examples in the explanation. Table 3.2 and Table 3.3 list the relocation types of the i386 architecture and the SPARC architecture respectively, and the symbols used in the *Calculation* column are explained in Table 3.1. The information from both tables are taken from the ABI standards on the System V Application Binary Interface, which are found in [**?**] and [**?**] respectively.

Definition 15 A relocation *reloc : Reloc* is a member of

Reloc = **tuple** {
  name : *string;*
  value : $\mathcal{Z}$;
  kind : $\{none, data, func, got, plt, copy, globdat, jmpslot, relative\}$;
  symb : *RelocSymb;*
  }

Definition 16 A relocation symbol *rsymb : RelocSymb* is a member of

RelocSymb = **tuple** {
  pcRel : $\mathcal{B}$;
  gotRel : $\mathcal{B}$;
  addend : $\mathcal{B}$;
  operation : *Dag;*
  extSize : $\mathcal{Z}$;
  rfield : *RelocField;*
  aligned : $\mathcal{B}$;
  }

Definition 17 A relocated field *rfield : RelocField* is a member of

RelocField = **tuple** {
  field : *InstrnField;*
  checkOverflow : $\mathcal{B}$;
  }

The first column of each row in the Table 3.2 and the Table 3.3 gives the name of the relocation type while the value in the second column is the relocation type value (`ELF32_R_TYPE`), which identifies the relocation type in binary files. The third column provides the relocated field, which is the location to which the calculated relocation value will be filled. The position of the relocated field in the instruction is given. For example, the relocated field of `R_SPARC_8` is in the bit range of <7:0>in an 32-bit (<31:0>) instruction. In Table 3.3, the relocated fields of SPARC have a '*V*' or '*T*' attached to the bit range. It tells whether the relocation type checks for overflow. If the relocation value is larger than the size of the relocated field, a relocation type may verify (*V*) the value fits or truncate (*T*) the result. To summarize the action defined by a relocation type, the linker will calculate the *relocation value* with the formula defined in the *Calculation* column of the table and may take some other actions as indicated in the last column. The relocation value is then filled into the relocated field after checking value overflow if required.

In a *Reloc*, the `name` and the `value` correspond to the the first and the second column respectively in Table 3.2 and Table 3.3.

It is one of our contributions in this project that we categorize relocations into 9 kinds. Our experience proves that it facilitates retargetting the GNU BFD library. This will be discussed in Chapter 4. The `kind` identifies the relocation kind to which the *Reloc* belongs. In addition to the `kind`, the `symb` captures the formula that calculates the relocation value (also known as resolved symbol value). The abstraction of a relocation symbol *RelocSymb* is as shown in Definition 16. Tasks except the relocation value calculation are implicit in the relocation kind. The relocated field (`rfield`) is included in the *RelocSymb*, and the abstraction of which, called *RelocField*, is in Definition 17. In a *RelocField*, the `field` and the `checkOverflow`, corresponding to the third column of the tables, give the position of the relocated field in the instruction and the overflow checking information respectively. The `field` is an instruction field defined in the ISA abstraction. The relocated field is always an immediate field, so the `field` also indicates if sign-extension is required when handling the relocation value.

Although the relocation types are processor-dependent, they are actually serving several common purposes even though they may take different action. Our study leads to the result that the different ISA is in fact the major factor that deviates the relocation types in different processor architecture. According to our observation, we suggest categorizing relocations sup-

Relocation Kind	Handled By	Purpose	Implicit Tasks
NONE	link-editor	Taking no action	
DATA	link-editor	Filling data symbol address	
FUNC	link-editor	Filling funcion symbol address info.	
GOT	link-editor	Creating an GOT entry	- creates GOT if it doesn't exist - creates a new GOT entry
PLT	link-editor	Creating an PLT entry	- creates PLT if it doesn't exist - creates a new PLT entry
COPY	runtime linker	Copying data from a shared library to non-PIC dynamic executable at dynamic linking time	- copies the value of the associated symbol from the shared library
GLOBDAT	runtime linker	Calculating symbol address and filling it to the GOT entry at dynamic linking time	- sets the designated symbol addr. to the GOT entry
JMPSLOT	runtime linker	Modifying the PLT entry to allow control transfer to the designated function symbol address	- modifies the PLT entry to transfer control to the designated symbol addr.
RELATIVE	runtime linker	Adding the base address of the loaded shared library to a relative address	- updates a relative addr.

Table 3.4: The 9 Relocation Kinds in the Model

ported in any ELF files into the 9 different kinds as shown in Table 3.4. Each relocation kind is characterized by its purpose. With this categorization, relocations of the same kind serve the same target client, which is either the link-editors or the runtime linkers. Besides, they are responsible for the same implicit tasks, which is as shown in the last column of Table 3.4. Implicit tasks, which are relocation actions other than relocation value calculation, correspond to what listed in the *Other action/Notes* columns of Table 3.2 and Table 3.3. As a result, relocations of the same kind can have different relocation value calculation formula but have the same implicit tasks.

The explanation of each kind is as follows:

- **Relocation kind: NONE**

  Relocations in the NONE kind give no instruction to the linker. Thus, they have no relocation value calculation formula nor relocated field information.

  Both of the `R_386_NONE` and the `R_SPARC_NONE` belong to this kind.

- **Relocation kind: DATA**

  Due to separate compilation, the virtual addresses of data symbols are not available until the linking and relocation is done. As a result, instructions which take memory reference address as operands will have incomplete content before linking and relocation. Relocations in the DATA kind instruct the linker to fill the symbol address, which is calculated with the formula defined in corresponding relocation type, to the appropriate instruction field (relocated field). Depending on the instruction semantics, the storage of the symbol address may be different. This is the reason that relocations in this kind can have different relocation value formula.

  Examples from i386 include `R_386_32`, `R_386_GOTOFF` and `R_386_GOTPC`.

  Examples from SPARC include `R_SPARC_8`, `R_SPARC_16`, `R_SPARC_32`, `R_SPARC_HI22`, `R_SPARC_22`, `R_SPARC_13`, `R_SPARC_LO10`, `R_SPARC_PC22` and `R_SPARC_UA32`.

- **Relocation kind: FUNC**

  This kind is similar to the DATA one, except this is for function symbol addresses instead of data symbol addresses. Function symbol addresses are unknown until the linking and relocation is done. Relocations in the FUNC kind instruct the linker to fill the relative branch offset, which is from the current instruction address to the indicated function symbol address. Also, the relocation value formula depends on the instruction semantics and therefore varies.

  `R_386_PC32` is the example of such kind in the i386.

  Examples from SPARC include `R_SPARC_DISP8`, `R_SPARC_DISP16`, `R_SPARC_DISP32`, `R_SPARC_WDISP30` and `R_SPARC_WDISP22`.

- **Relocation kind: GOT**

  The operation of the GOT (global offset table) has been explained in 2.2.4. When compiling a source file with the position-independent code generation option (e.g. `-fPIC`), the assembler will generate the appropriate GOT relocation to instruct the link-editor to allocate space in the GOT for the corresponding symbol and fill the corresponding field with the offset to the allocated GOT entry. If the GOT doesn't exist yet, the link-editor will also responsible for creating one. This task is usually done when the link-editor is requested to generate a dynamic object from the PIC-compiled object file.

`R_386_GOT32` is the example of such kind in the i386.

Examples from SPARC include `R_SPARC_GOT10`, `R_SPARC_GOT13` and `R_SPARC_GOT22`.

- **Relocation kind: PLT**

  The PLT relocations serve the same purpose as the GOT ones, except this is for creating a PLT (procedure linkage table) entry.

  The `R_386_PLT32` and the `R_SPARC_WPLT30` are examples.

- **Relocation kind: COPY**

  Shared libraries are usually built with position-independent code, but it's not the case for dynamic executables. As a result, the external symbolic resolution task must be done by code modification at dynamic linking time. The idea of COPY relocations are motivated by the desire of avoiding text segment modification at runtime in this situation.

  When the link-editor is making a dynamic executable, and a data reference is found residing in a shared library, space will be allocated in the `.dyn.bss` section of the executable. At the same time, the link-editor also generates a copy relocation that will instruct the runtime linker copying the indicated data from the shared library to the allocated space.

  The `R_386_COPY` and the `R_SPARC_COPY` are examples of copy relocations.

- **Relocation kind: GLOBDAT**

  For each entry in the global offset table, there has a GLOBDAT relocation, which is responsible for instructing the runtime linker to calculate the absolute virtual address of the associated symbol and fill the value to the GOT entry.

The `R_386_GLOBDAT` and the `R_SPARC_GLOBDAT` are examples.

- **Relocation kind: JMPSLOT**

  For each entry in the procedure linkage table, there has a JMPSLOT relocation, which is responsible for instructing the runtime linker to modify codes in the PLT entry in order to achieve control transfer to the resolved function symbol address when the function is invoked through the entry.

  The `R_386_JMPSLOT` and the `R_SPARC_JMPSLOT` are examples.

- **Relocation kind: RELATIVE**

  In a shared library, a RELATIVE relocation is created for each GOT entry. Holding a relative address, a RELATIVE relocation instructs the runtime linker to compute the corresponding virtual address by adding the virtual address at which the shared library was loaded (base address) to this relative address.

  The `R_386_RELATIVE` and the `R_SPARC_RELATIVE` are examples.

The calculation formula of a relocation type (*Reloc*) is abstracted in both of the `kind` member, which represents the relocation kind, and the `symb` member, which carries information about the symbol associated with the relocation. The relocation value calculation can be divided into two steps: 1) resolving the desired value, and 2) transforming the resolved symbol value into a form suitable to be stored in the relocated field. The value obtained from the first step may involve the value of the symbol, the index of which is carried in the relocation entry. In our model, the formula corresponding to the first step is abstracted by the *RelocSymb* members including the `pcRel`, the `gotRel` and the `addend`, in addition to the relocation kind. We use Algorithm 2 to map a *Reloc* into the expression corresponding to the first step. The meanings of the variables (`P`, `GOT`, `A`, `G` and `L`) used in the algorithm are as explained in Table 3.1. At the second step, the relocation value is obtained by transforming the result from the first step into a suitable representation for storage according to the definition of the associated instruction. This formula is carried by the `operation` member of the *RelocSymb*, in an expression form. In the `operation` *Dag*, the value obtained at the first step is represented as either `relocValue` or `gotRelocValue`. Using the *findSymbValue()* in Algorithm 2, Algorithm 1, which corresponds to the second step, maps a *Reloc* into the relocation value calculation formula abstracted in the data. At the end, the least signification number of `RelocSymb.extSize` bits are extracted from the result value of the second step. Figure 3.8 gives an example of abstracting the `R_SPARC_WDISP22` relocation.

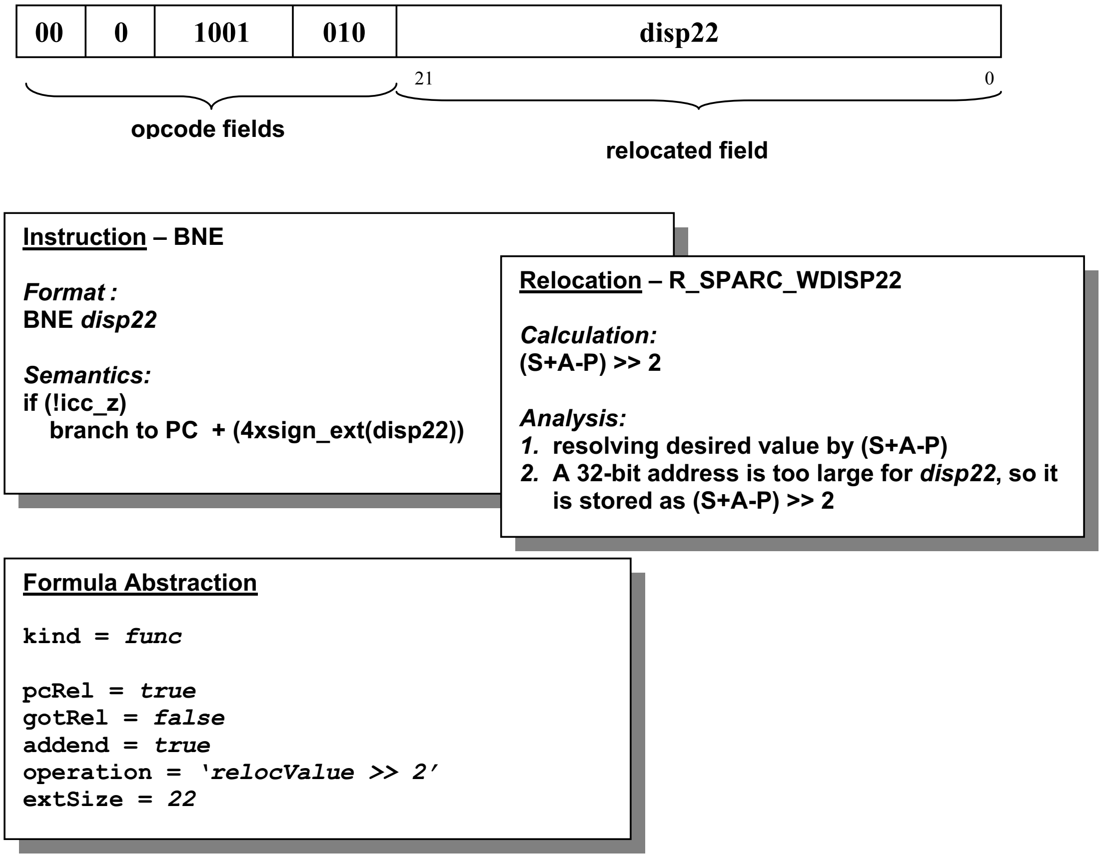


Figure 3.8: An Example of Relocation Formula Abstraction

Algorithm 1 *Applying Relocation Formula*

```
applyRelocFormula = func(
   reloc : Reloc
   ) : Z {
   var relocValue : Z;
   var gotRelocValue : Z;
   var result : Z;


   relocValue = findSymbValue(reloc, S);
   gotRelocValue = findSymbValue(reloc, GOT);

   result = applyRelocOper(reloc.operation, relocValue, gotRelocValue);

   return value ;
}
```

Algorithm 2 *Finding the Symbol Value*

```
findSymbValue = func(
   reloc : Reloc, symbol : Z
   ) : Z {
   var value : Z;

   value = 0;

   if( reloc.symb.pcRel == true )
      value = value - P;
   if( reloc.symb.gotRel == true )
      value = value - GOT;
   if( reloc.symb.addend == true )
      value = value + A;

   if( reloc.kind == data ∨ reloc.kind == func )
      value = value + symbol;
   if( reloc.kind == got )
      value = value + G;
   if( reloc.kind == plt )
      value = value + L;

   return value ;
}
```

To conclude the discussion of the relocation model, Example 7 gives the specification of the i386 `R_386_PC32` relocation and that of the SPARC `R_SPARC_WDISP22` relocation.

Example 7 *The description of the i386* `R_386_PC32` *and the SPARC* `R_SPARC_WDISP22`

```
r_386_pc32 = ⟨
  name = "R_386_PC32",
  value = 2,
  kind = func,
  symb = ⟨
    pcRel = true,
    gotRel = false,
    addend = true,
    operation = 'relocValue',
    extSize = 32,
    rfield = ⟨f_word32,true⟩,
    aligned = true,
    ⟩
  ⟩

r_sparc_wdisp22 = ⟨
  name = "R_SPARC_WDISP22",
  value = 8,
  kind = func,
  symb = ⟨
    pcRel = true,
    gotRel = false,
    addend = true,
    operation = 'relocValue >> 2',
    extSize = 22,
    rfield = ⟨f_disp22_s,true⟩,
    aligned = true,
    ⟩
  ⟩
```

It is assumed that that `operation` in a *RelocSymb* is expressed with simple operations (e.g. '+', '-', and '>>'). Besides the `relocValue` and the `gotRelocValue` used in the relocation abstraction, the other pre-defined variables which will be used in the subsequent discussion is summarized in Table 3.5.

Variable	Usage	Semantics
`dynSectAddr`	ANY	address of the dynamic section (`.dynamic`)
`pltSectAddr`	ANY	address of the PLT section (`.plt`)
`gotSectAddr`	ANY	address of the GOT section (`.got`)
`relocValue`	Reloc	relocation value
`gotRelocValue`	Reloc	relocation value but uses the GOT section address as the symbol value
`pltEntryOffset`	PLT	the offset from the start of the PLT section to the start of the current PLT entry (in bytes)
`pltgotEntryOffset`	PLT	the offset from the start of the `.got.plt` section, which is the GOT used by PLT if the PLT resides in the text segment, to the start of the corresponding `.got.plt` entry of the current PLT entry (in bytes)
`jsrelEntryOffset`	PLT	the offset from the start of the `.rel.plt` section to the start of the corresponding JMPSLOT relocation of the current PLT entry (in bytes)

Table 3.5: The Pre-defined Variables in the Processor Model

3.2.2 Global Offset Table Model

Since the runtime linker is not one of our target tools so far, the global offset table (GOT) model described in this subsection abstracts only the syntax but not the semantics of a GOT.

The global offset table is usually in a section named `.got`. Both of the format and the interpretation of the GOTs are processor-specific. While the link-editor is responsible for building the GOT when making dynamic object files, the runtime linker is informed with how to use it. Responsible for holding the absolute virtual symbol addresses as discussed in 2.2.4, a GOT is simply an array of memory addresses. A GOT entry is to hold a symbol address. Some entries at the beginning and/or at the end of the GOT may be reserved to hold special values. For some processor architectures, a symbol may be used to access the GOT entries from the text segment.

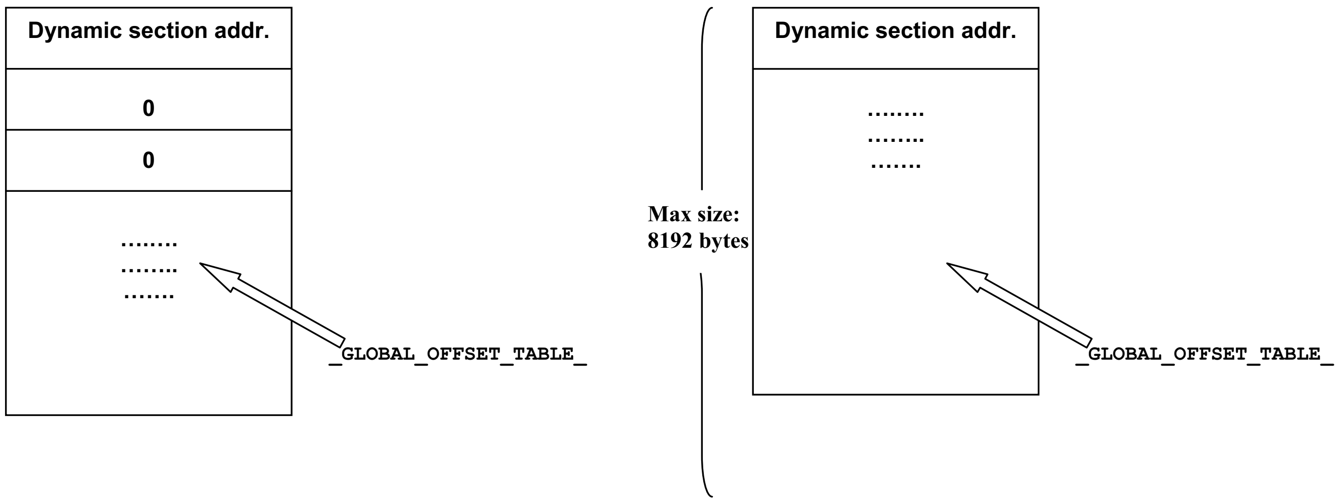
Definition 18 defines the abstraction of a global offset table. The `accessSymb` gives the name of the GOT access symbol if there is any. The `maxSize` gives the maximum size of the table in bytes, if there exists limitation in size. The size of a GOT entry is implied by the address size of the architecture, since each GOT entry is to store a symbol address. Some entries at the start of at the end of the GOT are reserved to carry some important values. Carrying a sequence of expression dags in order, the `startEntries` and the `endEntries` are for this purpose.

Each expression dag specifies a formula, which is used to calculate the value that will be stored in the entry by the linker. Some pre-defined variables in Table 3.5 may be used to specify these expressions.

Figure 3.9 illustrates the GOT of i386 and that of SPARC. Example 8 provides the corresponding specifications.

Definition 18 A global offset table *got : Got* is a member of

Got = **tuple** {
  accessSymb : *string;*
  maxSize : $\mathcal{Z}$;
  startEntries : $[\,]^{Dag}$;
  endEntries : $[\,]^{Dag}$;
  }



(a) i386 (b) SPARC

Figure 3.9: Abstracting the GOTs of i386 and SPARC

Example 8 *The description of the GOT of i386 and the GOT of SPARC*

```
i386GOT = ⟨
  accessSymb = "_GLOBAL_OFFSET_TABLE",
  maxSize = -1,
  startEntries = [ 'dynSectAddr','0', '0' ]
  ⟩

sparcGOT = ⟨
  accessSymb = "_GLOBAL_OFFSET_TABLE",
  maxSize = 8192,
  startEntries = [ 'dynSectAddr' ]
  ⟩
```

3.2.3 Procedure Linkage Table Model

Similar to the global offset tables, procedure linkage tables are built by link-editors and operated by runtime linkers. For the same reason, our procedure linkage table (PLT) model abstracts only syntax but not the semantics of a PLT.

The procedure linkage table is usually in a section named `.plt`. As discussed in 2.2.4, each PLT entry is responsible for redirecting the function call to the runtime-resolved absolute location. Some instructions may precede and/or the PLT entries. For some architectures, a symbol may be used to access the PLT entries.

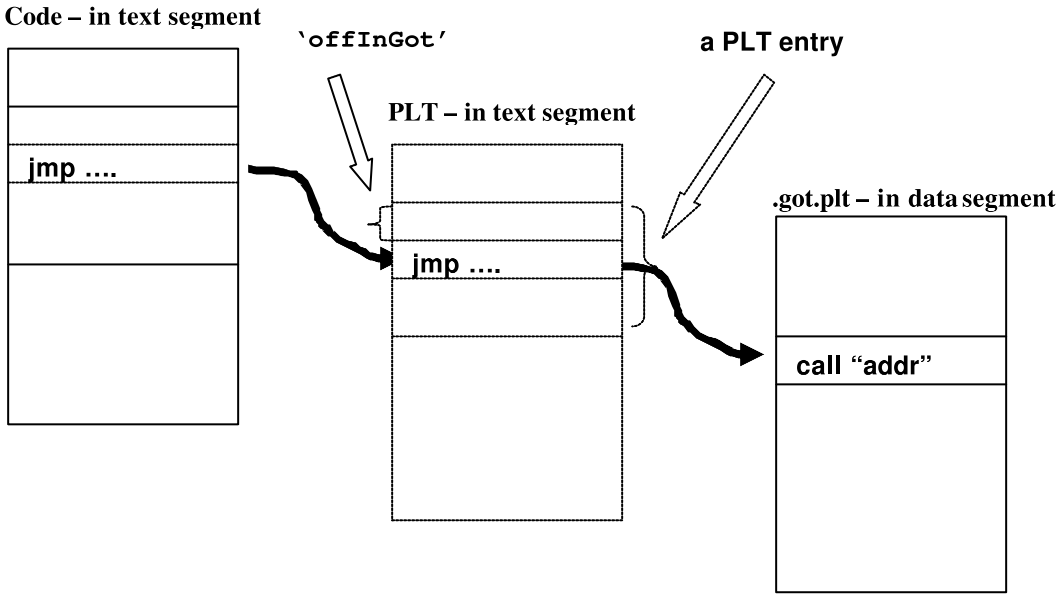


Figure 3.10: The Operation of a PLT in Text Segement

Although the PLT is a PIC feature, some architecture allow PLTs reside in the text segment. We assume that the operation of the PLT in this case will need to depend on a private GOT, usually in a section named `.got.plt`. Each PLT entry has a corresponding `.got.plt` entry, the address of which is hold in the PLT entry. The initialized value of each `.got.plt` entry holds an address pointing to some instruction in the corresponding PLT entry. So, the `.got.plt` is in data segment and the PLT is in text segment. With the intervention of the runtime linker at runtime, the `.got.plt` entry will hold the resolved absolute address of the function. Figure 3.10 shows this scenario. Our model assume PLTs residing in the text segment operate like this, but we cannot conclude that it is true for all architectures before carrying more sophisticated study.

Our PLT model is as defined from Definition 19 to Definition 22. In a *Plt*, the `inTextSeg` tells whether the PLT resides in the text segment. The `accessSymb` provides the name of the access symbol if there is any. The `maxSize` gives the maximum size of the table in bytes, if there exists limitation in size. If `inTextSeg` is true, then `offInGot` must be a non-negative integer since it provides the offset from the start of a PLT entry to the place (inside the entry), to which the initialized value of the corresponding `.got.plt` entry pointing. This is graphically indicated in Figure 3.10. An architecture may have two different set of PLT layouts, one for PIC-compiled files and the other one for non-PIC-compiled files. They are hold in `PICLayout` and the `nonPICLayout` respectively. If there exists one unique PLT layout, it will be given by the `nonPICLayout`.

A *PltLayout* consists of the specifications of the start entry and the end entry if there is any as well as that of the normal entry, which is the PLT entry that is responsible for redirecting call to the absolute location of a function symbol. The `normalEntry` is required while the `startEntry` and the `endEntry` are optional.

A *PltEntry* is characterized by its size in bytes (`size`), a binary code template (`tmpl`) and a set of *PltFill* members that describe how to fill the link time values to the template (`toFill`). The entry template is in the form of a sequence of bytes, the instructions are placed one by one starting from the index 0. Each *PltFill* consists of the bit range position to fill (`pos`) and the expression formula to make the filling (`filling`). Remember that some pre-defined variables in Table 3.5 may be used to specify the expression formulas.

Definition 19 A procedure linkage table *plt : Plt* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Plt} = \textbf{tuple}\ \{} \\
\quad \mathit{inTextSeg} & : \mathcal{B}; \\
\quad \mathit{accessSymb} & : \mathit{string}; \\
\quad \mathit{maxSize} & : \mathcal{Z}; \\
\quad \mathit{offInGot} & : \mathcal{Z}; \\
\quad \mathit{nonPICLayout} & : \mathit{PltLayout}; \\
\quad \mathit{PICLayout} & : \mathit{PltLayout}; \\
\quad \} &
\end{array}
$$

Definition 20 A procedure linkage table layout *pltLayout : PltLayout* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PltLayout} = \textbf{tuple}\ \{} \\
\quad \mathit{startEntry} & : \mathit{PltEntry}; \\
\quad \mathit{normalEntry} & : \mathit{PltEntry}; \\
\quad \mathit{endEntry} & : \mathit{PltEntry}; \\
\quad \} &
\end{array}
$$

Definition 21 A procedure linkage table entry *pltEntry : PltEntry* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PltEntry} = \textbf{tuple}\ \{} \\
\quad \mathit{size} & : \mathcal{Z}; \\
\quad \mathit{tmpl} & : [\,]^{\mathit{byte}}; \\
\quad \mathit{toFill} & : \langle\rangle^{\mathit{PltFill}}; \\
\quad \} &
\end{array}
$$

Definition 22 A procedure linkage table fill *pFill : PltFill* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{PltFill} = \textbf{tuple}\ \{} \\
\quad \mathit{pos} & : \mathcal{Z} \times \mathcal{Z}; \\
\quad \mathit{filling} & : \mathit{Dag}; \\
\quad \} &
\end{array}
$$

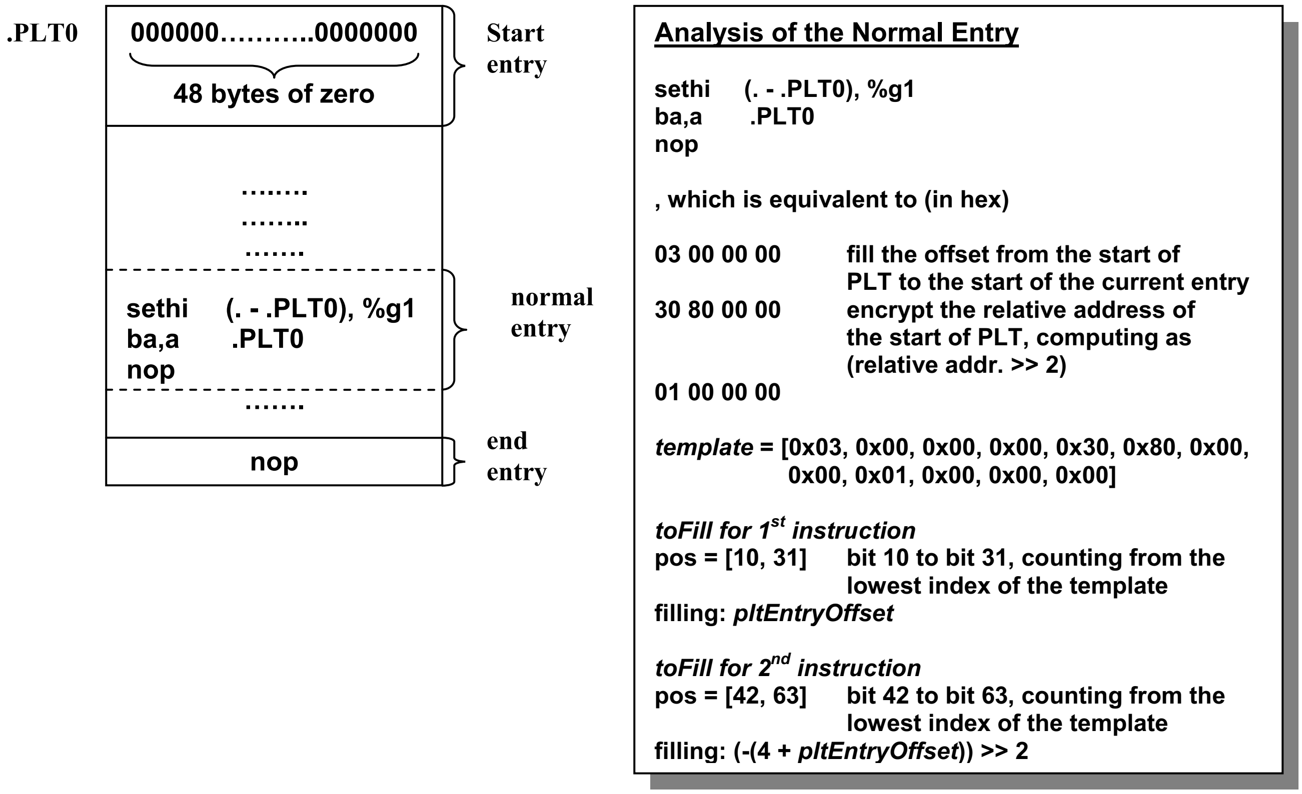


Figure 3.11: Abstracting the PLT of SPARC

Figure 3.11 is the PLT layout of SPARC and the corresponding specification is given in Example 9. The PLT starts with 48 bytes of zero and ends with a `nop` instruction. Initially, each PLT entry consists of three instructions: 1) the `sethi` instruction computes the distance between the start of the current entry and the start of the PLT and stores the most significant 22 bits of the result to `%g1` register, 2) the `ba,a` instruction jumps to `.PLT0`, and 3) the `nop` instruction. The abstraction of the normal entry consists of a template and two filling instructions to fill the location-independent values at the PLT construction time. Figure 3.11 attaches with the detailed analysis of the normal entry abstraction.

Example 9 *The description of the PLT of SPARC*

```
sparcPLT = ⟨
  inTextSeg = false,
  accessSymb = "_PROCEDURE_LINKAGE_TABLE",
  maxSize = -1,
  offInGot = -1,
  nonPICLayout = ⟨
    startEntry = ⟨
      size = 48,
      tmpl = [
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        ]
      ⟩
    normalEntry = ⟨
      size = 12,
      tmpl = [ 0x03, 0x00, 0x00, 0x00, 0x30, 0x80, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00 ]
      toFill = ⟨
        ⟨pos = [10,31], filling = 'pltEntryOffset'⟩,
        ⟨pos = [42,63], filling = '(-(4 + pltEntryOffset)) >> 2'⟩
        ⟩
      ⟩
    EndEntry = ⟨
      size = 4,
      tmpl = [ 0x01, 0x00, 0x00, 0x00 ]
      ⟩
    ⟩,
  PICLayout = ∅
  ⟩
```

3.2.4 Dynamic Section Model

```
typedef struct {
    Elf32_Word      d_tag;      // element type
    union {
        Elf32_Word  d_val;      // an integer value
        Elf32_Addr  d_ptr;      // an virtual address
        } d_un;
    } Elf32_Dyn;
```

Figure 3.12: ELF Dynamic Entries

The dynamic section is named as `.dynamic`. A special symbol, `_DYNAMIC`, labels this section. The dynamic section consists of an array of dynamic entries (`Elf32_Dyn`), the data definition of which is as shown in Figure 3.12. The `d_tag` of a dynamic entry identifies the type of information that it carries. Most of the dynamic entry types are processor-independent. So far, we find that `DT_PLTGOT` dynamic entry, which is entry with the `d_tag` value equals to the identification code of `DT_PLTGOT`, may carry the location of the PLT or that of the GOT, depending on the choice of the architecture. As a result, our model requires the user providing the expression of the `DT_PLTGOT` dynamic entry. Definition 23 defines the dynamic section entry model.

Definition 23 A dynamic section entry *dyn : Dyn* is a member of

Dyn = **tuple** {
  kind : {*pltgot*};
  val : *Dag;*
  }

Example 10 shows the specification of the dynamic section of SPARC and that of i386.

Example 10 *The description of the dynamic section of SPARC and i386*

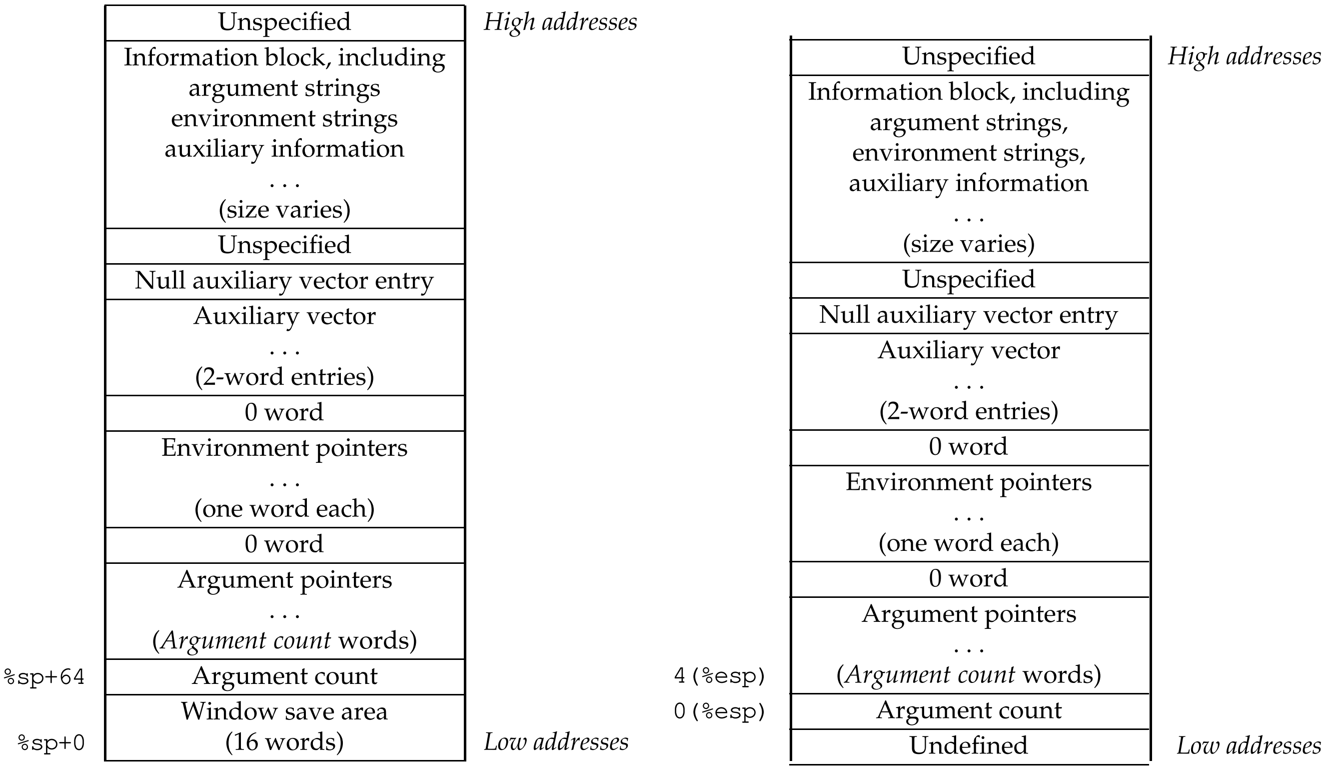
```
sparcDyns = ⟨
  kind = pltgot,
  val = 'pltSectAddr'
  ⟩

i386Dyns = ⟨
  kind = pltgot,
  val = 'gotSectAddr'
  ⟩
```

3.2.5 Stack Model

The objective of our stack model is to describe the initial process stack setting that should be done before execution. Initial stack layout is very important to the instruction set simulation, because it provides access to command line and environment of a program. Please note that our stack model carries no information about how to operate the stack at runtime. The stack is operated by the instructions at runtime, so it is the responsibility of the compiler to generate correct codes to manipulate the stack. Retargetable compiler is not included in this project so far.

Our stack model is designed according to the ABI standards on System V Application Binary Interface. The initial stack layout depends on both of the processor architecture and the operating system environment, but the necessary data is the same. All C programs have the interface - `main(int argc, char *argv[], char *envp[])`, where `argc` is the argument count and `argv` is an array of pointers to the command-line arguments, and the `envp` is an array of pointers to the environment strings. When a process is launched, the command-line arguments including the argument counter, the environment arguments, and the auxiliary vector, which holds startup information that should be passed by the operating system to the runtime interpreter, are pushed onto the stack. This is the initial stack layout that should appear before execution starts. Figure 3.13 are the initial process stack layouts of SPARC and



(a) SPARC (from [?]) (b) i386 (from [?])

Figure 3.13: The Initial Process Stack from System V ABI Standard

i386 from the System V ABI standard.

Our stack model is so far only for retargetting the SimpleScalar simulators, in which the input executables are assumed to be statically linked. As a result, we can ignore auxiliary vector, which carries startup information to the dynamic linker. The stack layout template of our model is shown in Figure 3.14 and the stack model is defined in Definition 24.

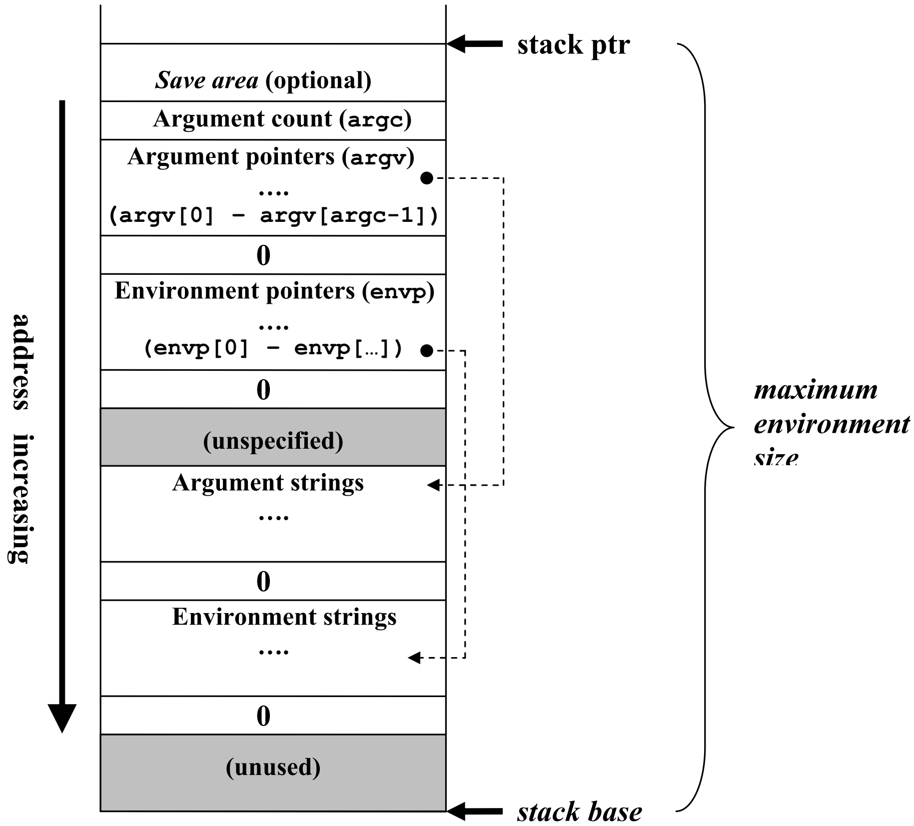


Figure 3.14: Our Stack Model

Definition 24 A stack *stack : Stack* is a member of

```
Stack = tuple {
    baseAddr      : ℤ;
    stackPtr      : RegCell;
    align         : ℤ;
    maxEnviron    : ℤ;
    saveArea      : ℤ;
    }
```

In a *Stack*, the `baseAddr` gives the stack base address (highest possible address in stack), which is OS-dependent. For example, the stack base address of SPARC running on SunOS 5.8 is 0xffbf0000. Usually, a register is responsible for holding the stack pointer value at runtime. This register cell is indicated in `stackPtr` of the model. The `align` gives the stack pointer value alignment in bits. For example, the stack pointer value must be guaranteed to be doubleword aligned in SPARC. The `maxEnviron` gives the maximum environment size requirement in bytes. This value would be required for the SimpleScalar simulators to set up the environment. At startup, the SimpleScalar simulators initialize the stack pointer to be (`stack base address - maximum environment size`), and then pushes the environment data underneath. The user can increase the `maxEnviron` value if stack overflow occurs. At last, the size of the optional *save area* (see Figure 3.14) is carried in `saveArea` in byte.

Example 11 is the specification of the initial stack layout of SPARC.

Example 11 *The description of the initial stack layout of SPARC*

```
sparcStack = ⟨
    baseAddr = 0xffbf0000,
    stackPtr = sp,
    align = 64,
    maxEnviron = 16384,
    saveArea = 64
    ⟩
```

3.2.6 Memory Usage Model

Definition 25 A memory usage *musage : MemUsage* is a member of

$$
\begin{array}{ll}
\multicolumn{2}{l}{MemUsage = \textbf{tuple}\ \{} \\
\quad dataTypeKind & : \{int, unsigned, float, addr\}; \\
\quad bitSize & : \mathbb{Z}; \\
\quad align & : \mathbb{Z}; \\
\quad \} &
\end{array}
$$

Defined in Definition 25, each memory usage defines the type and alignment restriction of a possible kind of data stored in memory. It works like the register usage defined in Definition 4. The `dataTypeKind` defines the data type, the data size in bit is given by `bitSize`. The `align` gives the address alignment restriction of in byte when the data of such type is stored in memory.

Example 12 gives the memory usage specification of SPARC. The memory usage consists of a set of *MemUsage*, each of which is the description of a supported data type and the corresponding alignment restriction.

Example 12 *The description of the initial stack layout of SPARC*

$$
\begin{array}{l}
sparcMemUses = \ \langle \\
\quad \langle unsigned, 8, 1\rangle,\ \langle int, 8, 1\rangle,\ \langle unsigned, 16, 2\rangle,\ \langle int, 16, 2\rangle, \\
\quad \langle unsigned, 32, 4\rangle,\ \langle int, 32, 4\rangle,\ \langle unsigned, 64, 8\rangle,\ \langle int, 64, 8\rangle, \\
\quad \langle float, 32, 4\rangle,\ \langle float, 64, 8\rangle,\ \langle addr, 32, 4\rangle \\
\quad \rangle
\end{array}
$$

3.2.7 Instruction Convention Model

The ISA model requires the users to define the rules to distinguish three kind of instructions: *function call*, *function return* and *indirect jump*. The definition rules of each kind is abstracted by a *InstrnConv*, which is defined in Definition 26.

In a *InstrnConv*, the `kind` identifies the kind of instructions being defined. The `rules` is a set of definition rules used to distinguish the instruction of this kind. An instruction is defined to be belonging to the `kind` if any rule in `rules` is satisfied. Each rule is abstracted by a *InstrnRule*, which is defined in Definition 27. Each *InstrnRule* is given as the instruction type by `instrn` and all the instruction field values that must be matched as given by `fieldValues`.

Definition 26 An instruction convention *def : InstrnConv* is a member of

$$
\begin{array}{ll}
\mathit{InstrnConv} = \textbf{tuple}\ \{ & \\
\quad \mathit{kind} & : \{\mathit{call}, \mathit{return}, \mathit{indirjmp}\}; \\
\quad \mathit{rules} & : \langle\rangle^{\mathit{InstrnRule}}; \\
\quad \} &
\end{array}
$$

Definition 27 An instruction rule *rule : InstrnRule* is a member of

$$
\begin{array}{ll}
\mathit{InstrnRule} = \textbf{tuple}\ \{ & \\
\quad \mathit{instrn} & : \mathit{Instrn}; \\
\quad \mathit{fieldValues} & : \langle\rangle^{\mathit{InstrnField} \times \mathbb{Z}}; \\
\quad \} &
\end{array}
$$

Example 13 shows the specification of the instruction conventions of SPARC. In the convention `d_call`, it is specified that the function call operation is done by `call` instructions. The convention `d_return` specifies that the return of a function call is done by `jumpli` instruction, but it is not the only job responsible by `jumpli`. A `jumpli` instruction is returning from a function only if the field `f_rd_g` is 0, the field `f_rs1_g` is 31 and the field `f_simm13_s` is 8. The convention `d_indir` specifies that there are two possible cases of indirect jump. One is done by the `jmpl` instruction when its field `f_rd_g` is equal to 15. The other one is done by the `jmpli` instruction when its field `f_rd_g` is equal to 15.

Example 13 *The SPARC instruction conventions*

```
sparcConvs = {
  d_call = ⟨
    kind = call,
    rules = {
      ⟨instrn = call, fieldValues = ∅⟩,
      }
    ⟩,
  d_return = ⟨
    kind = return,
    rules = {
      ⟨instrn = jmpli, fieldValues = {⟨f_rd_g, 0⟩, ⟨f_rs1_g, 31⟩, ⟨f_simm13_s, 8⟩}⟩
      }
    ⟩,
  d_indir = ⟨
    kind = indir,
    rules = {
      ⟨instrn = jmpl, fieldValues = {⟨f_rd_g, 15⟩}⟩,
      ⟨instrn = jmpli, fieldValues = {⟨f_rd_g, 15⟩}⟩
      }
    ⟩
  }
```

Chapter 4

Retargetting GNU BFD Library and Linker

Most of the downstream tools in the GNU Binutils package depend on the BFD library. To achieve automatic porting these downstream tools, retargetting the GNU BFD library must be the first step. Among these tools, the GNU linker is especially highly dependent on the BFD library, such that most of the core linking functions, many of which are processor-dependent, are provided by the BFD library. As a result, automatic generating the functions required by the GNU linker becomes an unsplittable part of retargetting the BFD library, and we therefore decide to enable retargetting both of the BFD library and the linker by the same tool.

In this chapter, we discuss the methodology of retargetting both of the GNU BFD library and the GNU linker. The relevant details of both will be briefly described in order to present our methodology. The hacking details can be found in [12]. The description provided in this chapter and [12] is based solely on what we have been able to deduce from studying the source code.

4.1 The GNU BFD Library

The GNU BFD library provides services to manipulate object files, including the basic I/O functions. In this section, we briefly overview the infrastructure and the internals of the BFD library, and then conclude with the porting instructions.

4.1.1 Infrastructure

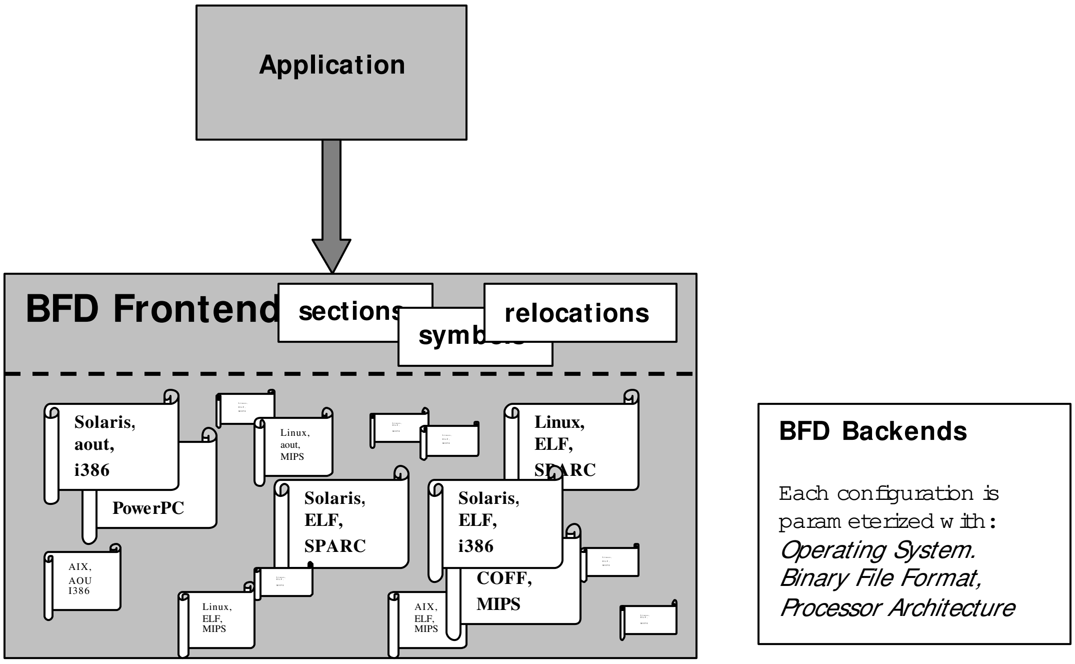


Figure 4.1: The Design of the GNU BFD Library

The *object file configuration* consists of three variables, namely the binary file format, the operation system environment and the processor architecture. The BFD library allows applications to use the same set of APIs to operate on object files in any format and for any OS and processor architecture. To achieve this goal, the BFD library is designed to be consisting of two parts: the frontend and a set of backends (Figure 4.1). The frontend provides an interface to the application, so the object files configuration details is abstracted away. The real views of the object files are handled by the BFD backends. The frontend decides which backend to use and when to call the backend routines to maintain its generic view. A set of pre-defined backends exists in the BFD library. Each backend provides the data and the routines for a unique object file configuration. By calling the routines provided from the backends, the frontend layer is able to correctly manipulate object files in any configuration as long as the corresponding backend exists.

4.1.2 The Internals

Each object file is abstracted as an object with the data type of `struct _bfd` in BFD. We call this as "bfd object" in the rest of this thesis. The users would manipulate the bfd objects with the APIs provided by the frontend.

The Frontend Data

From the uers's perspective, the three important object file elements, which are *sections*, *symbols* and *relocations*, are carried at generic forms in a bfd object. This is the abstracted view of a bfd object at the frontend layer (see Figure 4.1).

While the concept of a *section* in BFD is the same as that in ELF, BFD library handles *relocation* in different way from the ELF. In ELF, if relocation information is required by a section, another section will be created only for carrying the relocation entries. The section pair, (`.text, .rela.text`), is an example. In BFD, these relocation entries are however attached to the section in the form of a data array, each element of which has the data type of `struct reloc_cache_entry`. Figure 4.2 is the definition of `struct reloc_cache_entry` along with the equivalent ELF data. Similar to the `ELF32_R_TYPE` in `Elf32_Rel` and `Elf32_Rela` of ELF, the `howto` field in `reloc_cache_entry` of BFD has knowledge about how to process the relocatable field, and it is in fact a piece of processor-dependent information attached by the BFD backend.

Unlike the ELF object files, the BFD library handles only one symbol table section. The implied symbol table is the one used by the instruction content per object file basis. The other

```
typedef struct reloc_cache_entry {
    struct symbol_cache_entry    **sym_ptr_ptr;   // ELF32_R_SYM in ELF
    bfd_size_type                address;         // r_offset in ELF
    bfd_vma                      addend;          // r_addend in ELF
    reloc_howto_type             *howto;          // ELF32_R_TYPE in ELF
    }
```

Figure 4.2: Data definition of a BFD Relocation Entry

symbol tables, such as dynamic symbol table, if present, are handled by the BFD backend. As a result, the symbol table in BFD is handled per bfd object basis. The BFD symbol table is an array of bfd symbol object. Instead of being carried inside the bfd object, the symbol table is loaded to a specified memory location upon request.

The frontend data in the BFD frontend is summarized in Figure 4.3.

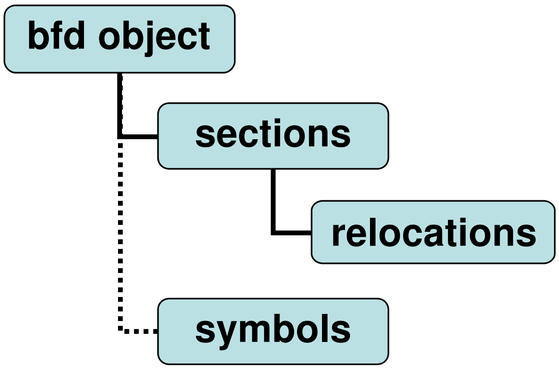


Figure 4.3: The Relationship of the BFD Frontend Elements

The Backend Data

The target-specific information, that cannot be fit into the generic form at the frontend, will be carried by the backend data.

For each distinct BFD backend, which is a configuration consisting of the binary file format, the OS environment and the processor architecture, a set of backend data is built and distributed in different places inside the bfd object data. Figure 4.4 illustrates the locations of the backend data for ELF files, and the role played by each data is summarized in Table 4.1. Only the backend data that is used by the ELF-targets is mentioned here.

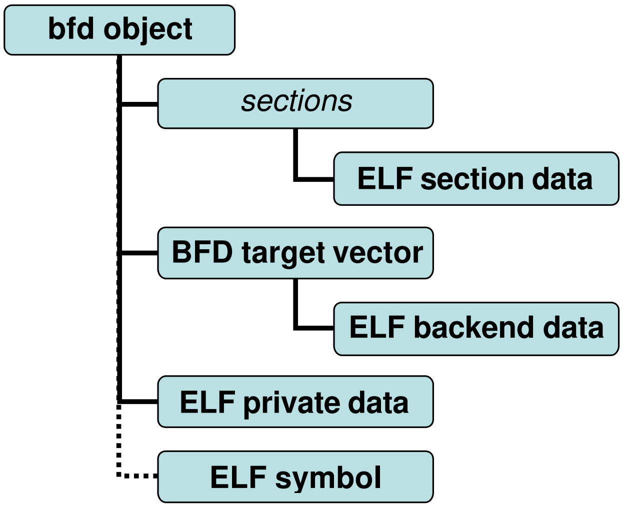


Figure 4.4: The Relationship of the BFD Backend Elements for ELF files

Backend Data	**Target Information**
BFD target vector	- binary file format - processor architecture - function pointers for frontend
ELF backend data	- properties of ELF files - processor-dependent details of ELF - if necessary, probably OS-dependent details of ELF - function pointers for ELF files processing routines
ELF section data	- section data specific to ELF
ELF symbol	- symbol data specific to ELF
ELF private data	- bookkeeping data of the ELF linking process

Table 4.1: A Summary of BFD Backend Data for ELF Files

Besides identifying the binary file format and the processor architecture, the BFD target vector holds 9 sets of function pointers, which are as listed in Table 4.2, regardless of the binary file format. These function pointers are to redirect the invocations of the frontend APIs to the appropriate backend implementations.

Besides the BFD target vector, all other backend data illustrated in Figure 4.4 are ELF-specific, which means that these data are needed if and only if the backend is specifying a ELF-target. The ELF section data is carried by the frontend section as a supplementary of the generic data. However, the ELF symbol is in fact the frontend symbol itself but appears as different form at different abstraction layer. The ELF private data is used by the GNU linker to keep track of the intermediate values in the linking process. This happens when the GNU linker invokes the target-specific linking algorithms in the BFD library. Besides carrying the ELF data, the ELF backend data also possesses some processor-dependent details, such as relocations, GOTs and PLTs, as well as the OS-dependent details if it is required. These information are carried in the form of either as data or functions.

Category	**Purpose**
Copy	copying bfd objects
Core file support	manipulating core files
Archive support	manipulating archive files
Symbol table support	processing backend symbols
Relocation support	processing relocations
Output	writing out the bfd objects
Linker	providing support to the GNU linker
Dynamic linking support	reading dynamic linking information
Generic	carrying functions which don't fit into other categories

Table 4.2: The 9 Categories of Functions in the BFD Target Vector

4.1.3 Adding a New ELF-Target Backend

An *ELF-target backend* is a BFD backend which has ELF as its binary file format in the configuration. In order to assist porting the BFD library to a new ELF-target, the BFD library provides two template files, one for 32-bit machine (`elf32-target.h`) the other for 64-bit (`elf64-target.h`), to facilitate adding new ELF-target backends. As we limit our goal to support only 32-bit machines in this study, we will only discuss the `elf32-target.h` file. We will call this file as "ELF-target template" in the rest of this thesis.

The ultimate purpose of using the ELF-target template is to build the BFD target vector for a new ELF-target. Referring to Table 4.1, we could find that the BFD target vector, which contains also the ELF backend data, is the only backend data varying with more than only the binary file format. The ELF-target template instructs the users to provide the relevant information and uses these to overwrite the default setting in order to create the BFD target vector for the new ELF-target. To construct a BFD target vector with the ELF-target template, these information are written in a file which is named with the convention of `elf32-`*`myarch`*`.c` (e.g. `elf32-sparc.c` for SPARC).

We can conclude that, the problem of retargetting the BFD library to a new ELF-target is almost reduced to be the problem of automatic generating the `elf32-`*`myarch`*`.c` file for the new target. Since all the processor-dependent linking algorithms are implemented in this file, automatic generating this file is also helping us to retarget the GNU linker.

4.2 The GNU Linker

The GNU linker is a link-editor. Similar to the BFD library that it can be used to manipulate object files for various targets, the GNU linker can be configured to perform linking on object files for many different targets.

This section is organized as follows. Section 4.2.1 will overview the dependency of the GNU linker to the BFD library. Section 4.2.2 will then describe the linking services provided by the BFD library and required by the GNU linker. Section 4.2.3 describes the emulation, which is the target-dependent part of the GNU linker. The ultimate goal of this section is to bring out the target-dependent part that has to be automatically generated for the GNU linker in the retargetting process. The understanding of this helps us to retarget a well-equipped BFD library to a new ELF-target.

4.2.1 The Dependency to the BFD Library

As we have mentioned, the GNU linker is highly dependent on the BFD library. This is because that the linking task requires knowledge about the binary file format in order to thread the object files together correctly, and also some processor-dependent details about dynamic linking in order to compose the dynamic entities such as GOTs and PLTs on the output object file. With the BFD library providing the target-dependent linking functions, the GNU linker is only responsible for calling these functions at appropriate time and step to handle each task. In fact, the real jobs of linking are done inside the BFD library.

Figure 4.5 demonstrates the relationship between the BFD library and the GNU linker. To support each target, the GNU linker requires an emulation. An emulation provides target-dependent data and algorithms to enable the GNU linker properly handling object files for a particular target configuration. The BFD backends to the GNU BFD library is the same as the emulations to the GNU linker. Similar to that of the BFD library, the frontend part of the GNU linker is machine-independent. At each linking execution, the GNU linker points to one emulation, which contains information about the corresponding BFD backend that should be used.

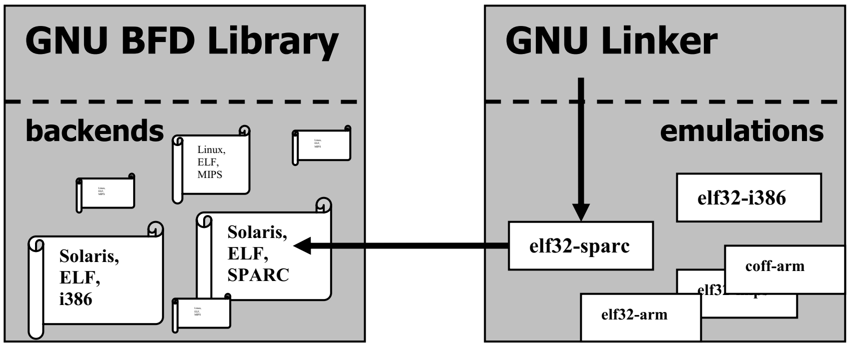


Figure 4.5: The Relationship between the GNU BFD Library and the GNU Linker

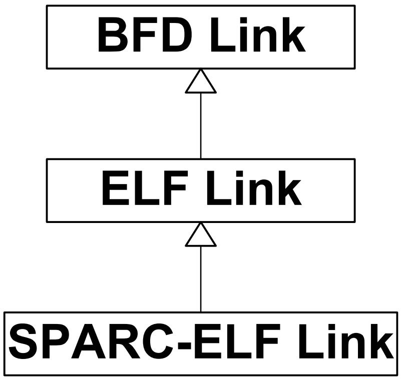
4.2.2 Linking Facilities from GNU BFD

The linking services provided by the BFD library range from the basic generic linking function such as symbol resolution to the target-specific task handling such as dynamic linking. We can consider the linking services are provided from a hierarchy of link classes, which is unique to each target. However, the data and method members of each link class scatter over different places in BFD. The picture of the link class hierarchy of a target is captured in its BFD backend data. The UML diagrams in Figure 4.6 present the link class hierarchies of the SPARC-ELF [1] target and the i386-ELF target in the latest BFD library release (*version 2.13*).

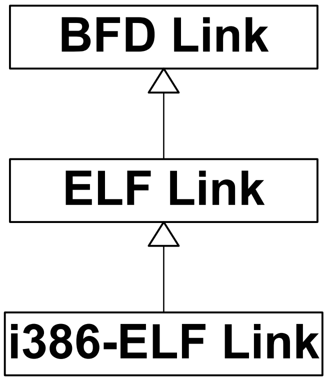
A hash table is essential to symbol resolution, and it is thus a very important member in any link class. Symbol resolution is usually done on the fly when the linker scans through the global symbols in each input object file. In this thesis, we call the hash table used for the linking purpose as "link hash table", and the entry used in such hash table as "link hash entry". Any link hash table is built on a BFD hash table, which is one of the utilities provided from the BFD library. Besides keeping track of the state (e.g. undefined) of a symbol, the link hash table entry in the GNU linker can be customized to carry other target-specific information about a symbol. Provided by the BFD library and being implemented in its belonging link class, the link hash table used by the GNU linker has its ability enhanced at each level of the link class in the hierarchy. Figure 4.7 illustrates the structures of the link hash tables used by the SPARC-ELF target and the i386-ELF target respectively in the latest BFD library release (*version 2.13*). In the current release, the SPARC-ELF target uses the ELF link hash table, while the i386-ELF target has the ELF link hash table enhanced to facilitate some machine-specific linking jobs. It is only a matter of choice for the authors who ported the BFD library to these targets.

The link class hierarchies of all ELF-targets follow the same pattern. Providing support to the basic and generic linking functions, the *BFD link class* is always the root of any link class hierarchy regardless of the target configuration. Subclassing from the BFD link class is usually

[1] Hereafter, we use the convention of *arch_name-binary_format* to represent a 32-bit target with the specified processor type and the specified binary file format.

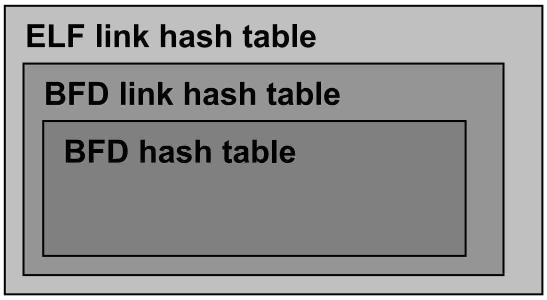


(a) SPARC-ELF

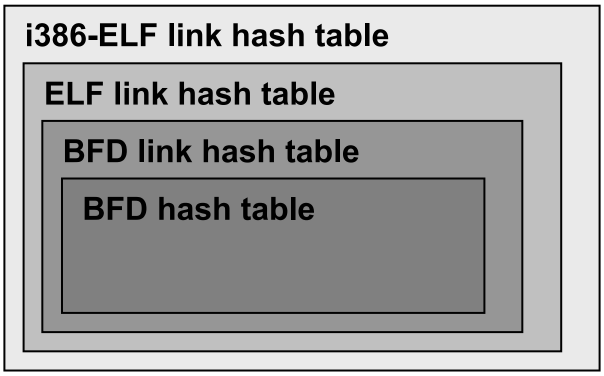


(b) i386-ELF

Figure 4.6: The Link Class Hierarchies Used by the SPARC-ELF Target and the i386-ELF Target



(a) SPARC-ELF



(b) i386-ELF

Figure 4.7: The Structures of the Link Hash Tables

one which provides linking support to satisfy the needs of a specific binary file format. That for the ELF-targets is what we call the *ELF linker class* in Figure 4.6. To be the leaf in the hierarchy, a target-dependent link class (e.g. SPARC-ELF link class) is finally built upon the ELF link class to provide the machine-dependent support for the corresponding ELF-target. After a brief introduction of the BFD linker utility, in the following, we will give more details about the link classes which contribute to the linking functions of the ELF-targets. The hash table support provided by each link class will especially be emphasized.

The BFD Hash Table

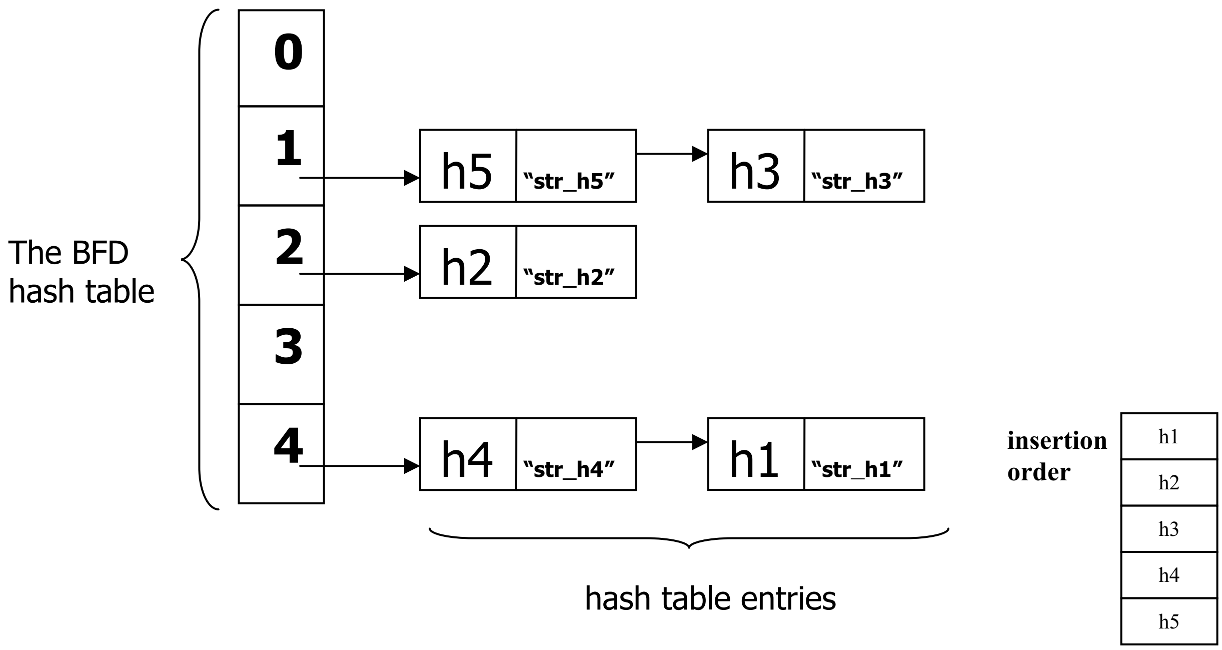


Figure 4.8: The BFD Hash Table

The hash table utility provided by the BFD library consists of the data structure definitions and a set of APIs for manipulating the BFD hash tables and their entries. As we have shown in Figure 4.7, a special-purpose hash table can be derived from the BFD hash table. In this and the subsequent parts, we will illustrate how it can be done.

```
struct bfd_hash_entry {
    struct bfd_hash_entry       *next;          // next entry ptr
    const char                  *string;        // string being hashed
    unsigned long               hash;           // hash code
    }

struct bfd_hash_table {
    struct bfd_hash_entry       **table;        // array of hash entry linked list
    unsigned int                size;           // # buckets
    struct bfd_hash_entry  *(*newfunc) PARAMS
                          ((struct bfd_hash_entry *,
                            struct bfd_hash_table *,
                            const char *));     // table initialization API
    PTR memory;                                 // memory chunk
    }
```

Figure 4.9: Data Structure of the BFD Hash Entry and that of the BFD Hash Table

First of all, Figure 4.8 provides an abstracted picture of the BFD hash table and entries. Each BFD hash entry only carries a *string* (e.g. "str_h1" of entry h1 in Figure 4.8). A BFD hash table is a fixed sized array of BFD hash entry linked list. Each element of the array represents a bucket. The hash code, which is calculated from the string storing inside the entry, determines the bucket into which the hash entry will be stored. The hashing collision problem is handled by storing the collided entries in a dynamically growing linked list in the bucket. This is as illustrated in Figure 4.8.

Figure 4.9 gives the data structure definition of the BFD hash entry and that of the BFD hash table. When deriving a special-purpose hash table class from the BFD hash table class, these data structures will become the parent data of the new hash table data structures. The BFD hash utility also provides a set of API to serve *hash table construction, hash entry construction, hash table destruction, hash entry lookup, hash entry replacement, and hash table traverse*. A link hash table built on the BFD hash table will provide a similar API set, and implementation of each will invoke the corresponding parent API to manipulate the parent data.

The BFD Link Class

Regardless of the target, the BDF link class is always the root of a link class hierarchy. The BFD link class is to provide the most basic symbol handling services - symbol resolution.

In this subsection, we will describe the BFD link hash table provided by the BFD link class, as well as the services provided to do symbol resolution by manipulating the BFD link hash tables.

```
enum bfd_link_hash_type {
    bfd_link_hash_new,          // symbol is newly created
    bfd_link_hash_undefined,    // undefined symbol
    bfd_link_hash_undefweak,    // weakly undefined symbol
    bfd_link_hash_defined,      // defined symbol
    bfd_link_hash_defweak,      // weakly defined symbol
    bfd_link_hash_common,       // common symbol
    bfd_link_hash_indirect,     // symbol is an indirect link
    bfd_link_hash_warning       // like indirect, but warn if referenced
    }
```

Figure 4.10: The BFD Link Hash Entry Type

In the BFD link class, symbols are classified into 7 types: *undefined*, *weakly undefined*, *defined*, *weakly defined*, *common*, *indirect* and *warning*. Section 2.2.3 has reviewed the semantics of some symbol types. The *external symbols* mentioned in section 2.2.3 are equivalent to the *undefined symbols* we discuss here. An indirect symbol is a pseudo symbol which carries a link to a real symbol and the version information about the linking symbol. Similarly, a warning symbol carries a link to a real symbol and a string of warning message about the linking symbol. The symbol data type is represented by `enum bfd_link_hash_type`, which is as shown in Figure 4.10. In addition to the 7 symbol types, the `bfd_link_hash_new` type of `enum bfd_link_hash_type` represents that the symbol has not been added to the hash table and it doesn't belong to any type.

The symbol resolution function provided by the BFD link class is done on the fly when each global symbol is added to the BFD link hash table, which is implemented and operated in the BFD link class. Figure 4.11 gives the data structure definition of the BFD link hash table (`struct bfd_link_hash_table`) and that of the corresponding entry (`struct bfd_link_hash_table_entry`). The BFD link hash table and the BFD link hash entry are built on top of the BFD hash table and the BFD hash entry respectively (refers to Figure 4.7). A BFD hash entry is carried as the parent data in the first element of each BFD link hash entry.

```
struct bfd_link_hash_entry {
    struct bfd_hash_entry       root;          // parent hash entry
    enum bfd_link_hash_type     type;           // link hash entry type
    struct bfd_link_hash_entry  *next;          // next ptr
    union {
        struct {
            abfd                *abfd;          // source of the symbol
            } undef;                            // undefined sym.

        struct {
            bfd_vma             value;          // symbol value
            asection            *section;       // symbol section
            } def;                              // defined sym.

        struct {
            struct bfd_link_hash_entry *link;   // real symbol
            const char          *warning;
            } i;                                // warning/indirect sym.

        struct {
            bfd_size_type       size;           // symbol size;
            struct bfd_link_hash_common_entry {
                unsigned int    alignment_power; // alignment
                asection        *section;       // symbol section
                } *p;
            } c;                                // common sym.
        } u;
    }

struct bfd_link_hash_table {
    struct bfd_hash_table       table;          // parent hash table
    const bfd_target            *creator;       // backend BFD target vector
    struct bfd_link_hash_entry  *undefs;        // undefined symbols
    struct bfd_link_hash_entry  *undefs_tail;   // tail of the above list
    enum bfd_link_hash_table_type   type;       // link hash table type
    }
```

Figure 4.11: Data Structure of the BFD Link Hash Entry and that of the BFD Link Hash Table

In addition to the symbol name string in the parent data, extra symbol-type-dependent information is carried in each entry. Similarly, a BFD hash table is carried as the parent data in the first element of each BFD link hash table. This BFD hash table is to carry a table of *BFD link hash table entries*. A BFD link hash table also carries the pointer of the BFD target vector (`creator`), which belongs to the backend target that creates the BFD link hash table. The backend BFD target vector provides target-specific APIs required in the linking process.

Each BFD link hash entry identifies one symbol. While linking, the global symbols of each input object files are added to the hash table. If no symbol with the same name is found in the hash table before adding a symbol, a new link hash entry with the appropriate type (one of the 7 symbol types that we have discussed) is created and added to the hash table. If the

Previous Type

New Type	*NEW*	*UNDEF*	*UNDEFW*	*DEF*	*DEFW*	*COM*	*INDR*	*WARN*
UNDEF	UND	NOACT	UND	REF	REF	NOACT	REFC	WARNC
UNDEFW	WEAK	NOACT	NOACT	REF	REF	NOACT	REFC	WARNC
DEF	DEF	DEF	DEF	MDEF	DEF	CDEF	MDEF	CYCLE
DEFW	DEFW	DEFW	DEFW	NOACT	NOACT	NOACT	NOACT	CYCLE
COM	COM	COM	COM	CREF	COM	BIG	REFC	WARNC
INDR	IND	IND	IND	MDEF	IND	CIND	MIND	CYCLE
WARN	MWARN	WARN	WARN	CWARN	CWARN	WARN	CWARN	NOACT

UND	Mark as undefined	WEAK	Mark as weak undefined
DEF	Mark as defined	DEFW	Mark as weak defined
COM	Mark as common	REF	Mark as referenced if defined
REFC	Mark the indirect and the pointing symbol as referenced		
IND	Create an indirect symbol	MWARN	Create a warning symbol
CIND	Create an indirect symbol from a common		
CREF	Handle multiple definitions with common		
CDEF	Handle multiple definitions with common		
MDEF	Handle multiple definitions error		
MIND	Handle multiple indirect symbols		
WARN	Issue warning	CWARN	Warn if referenced, else MWARN
WARNC	Issue warning and then CYCLE		
CYCLE	Process the symbol pointed by the indirect link	NOACT	No action
BIG	Use the larger size of both commons		

Figure 4.12: The State Table for Symbol Resolution

a symbol with the same name is found in the hash table, symbol resolution is done according to the type of the existing symbol and that of the new symbol. Usually, the result of symbol resolution is simply changing the symbol type in the BFD link hash entry. The state table in Figure 4.12 summarizes the symbol resolution process. With the existing symbol type and the new symbol type, the state table provides the appropriate resolution action to take. The first column of the state table, which corresponds to the `bfd_link_hash_new` symbol type, provides the resolution action when no symbol with the same name as that of the new symbol exists in the hash table.

Figure 4.13 gives an example of symbol resolution process. There are two input object files - main.c and sum.c in this example. We only focus on the handling of one symbol - *sum*. The first input file (main.c) references the *sum* symbol by calling the external function. So, an undefined *sum* symbol is added to the hash table. The second input file (sum.c) defines a

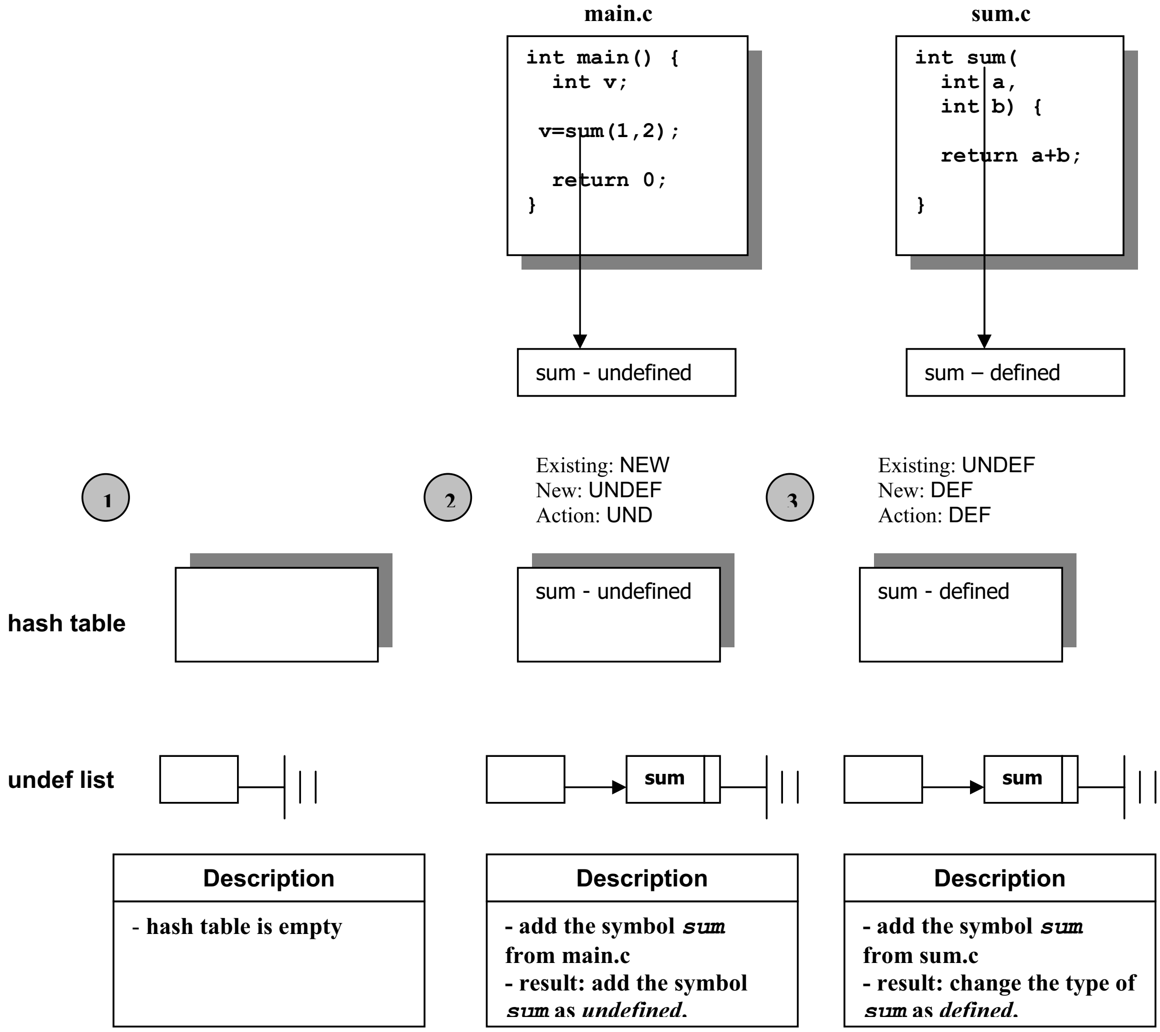


Figure 4.13: Example of Symbol Resolution Using the BFD Link Hash Table

method named *sum*. When processing the defined *sum* symbol in the second input file, symbol resolution is done against the existing undefined *sum* symbol. By checking the state table, the third row and the second column indicates the `DEF` action, which instructs marking the symbol as *defined*.

The ELF Link Class

For all ELF-targets, the ELF link class is the second level of the link class hierarchy by sub-classing the BFD link class (see Figure 4.6). The ELF link class provides linking services specifically to handle the ELF object files.

```
struct elf_link_hash_entry {
    struct bfd_link_hash_entry      root;       // parent BFD link hash entry
    long                            indx;       // symbol index for output file
    long                            dynindx;    // symbol index if this is a dynamic symbol;
                                                // -1 otherwise
    union {
        bfd_signed_vma  refcount;               // keeps track the usage of the
        bfd_vma         offset;                 // symbol if it requires an
    }                               got;        // entry in the GOT at output

    union {
        bfd_signed_vma  refcount;               // keeps track the usage of the
        bfd_vma         offset;                 // symbol if it requires an
    }                               plt;        // entry in the PLT at output

    char                            type;       // ELF symbol type (e.g. STT_OBJECT)

    .......

    }

struct elf_link_hash_table {
    struct bfd_link_hash_table      root;       // parent BFD link hash table
    bfd_size_type                   dynsymcount;  // number of dynamic symbols
    struct elf_link_hash_entry      *hgot;      // hash entry of the ``__GOT_OFFSET_TABLE_'' symbol
    struct bfd_link_needed_list     *needed;    // list of dependent shared libraries
    struct elf_link_loaded_list     *loaded;    // list of loaded input bfd objects

    ......

    }
```

Figure 4.14: Data Structure of the ELF Link Hash Entry and that of the ELF Link Hash Table (Partial)

The detailed description of the ELF linking services is out of scope of this thesis. However, Figure 4.14 provides the partial data structure definition of the ELF link hash table and that of the corresponding entry. We will leave the discussion of the relevant data usage to the upcoming sections of this chapter when we describe the linking algorithms.

The *MYARCH*-ELF Link Class

To serve a ELF-target, a target-dependent link class, which is what we so-called *MYARCH*-ELF link class, will be built upon the ELF link class in its link hierarchy. The SPARC-ELF link class and the i386-ELF link class are examples for the SPARC-ELF target and the i386-ELF target respectively (see Figure 4.6).

The *MYARCH*-ELF link class provides the machine-specific linking services for ELF files. To make it clear, the *MYARCH*-ELF link class contains knowledge about the machine-specific details required for linking, with the assumption of handling only ELF files. The contributions of the *MYARCH*-ELF link class to the linking job are in two-fold. First, relocation is machine-dependent and the *MYARCH*-ELF link class can provide service to process the relocation entries in the input object files and take the appropriate action. Second, if the target configuration requires dynamic linking support, the *MYARCH*-ELF link class will provide service to compose machine-dependent dynamic entities on the output object file. For example, the layout of the global offset table (GOT) and that of the procedure linkage table (PLT) are both machine-dependent, and the *MYARCH*-ELF link class will be responsible to compose the GOTs and the PLTs to handle the dynamic symbols.

Depending on the developers of the BFD backend, a machine-specific link hash table may be built on the ELF link hash table, in order to facilitate the machine-specific linking jobs. The scenario is similar to that the ELF link hash table is built on the BFD link hash table.

The implementation of the *MYARCH*-ELF link class resides in the corresponding `elf32-`*`myarch`*`.c` file. We have concluded in section 4.1.3 that the `elf32-`*`myarch`*`.c` is the only file that must be generated automatically in order retarget the BFD library to a new ELF-target. In fact, the linking facilities of the *MYARCH*-ELF link is the biggest part in the `elf32-`*`myarch`*`.c` file. Thus, automatically generating functions that play the role of the *MYARCH*-ELF link class is our critical problem. This leads to the discussion of the methodology of our solution in the subsequent section.

4.2.3 Emulations of the GNU Linker

Each GNU linker target requires an emulation [**?**]. An emulation contains three types of target-dependent information as follows:

- the name of the corresponding BFD backend target, the APIs provided by which will be used in the linking process,

- the linker script, which provides mapping rules between sections from the input object files and sections in the output file,
- some algorithms, which are mostly dependent on the binary file format.

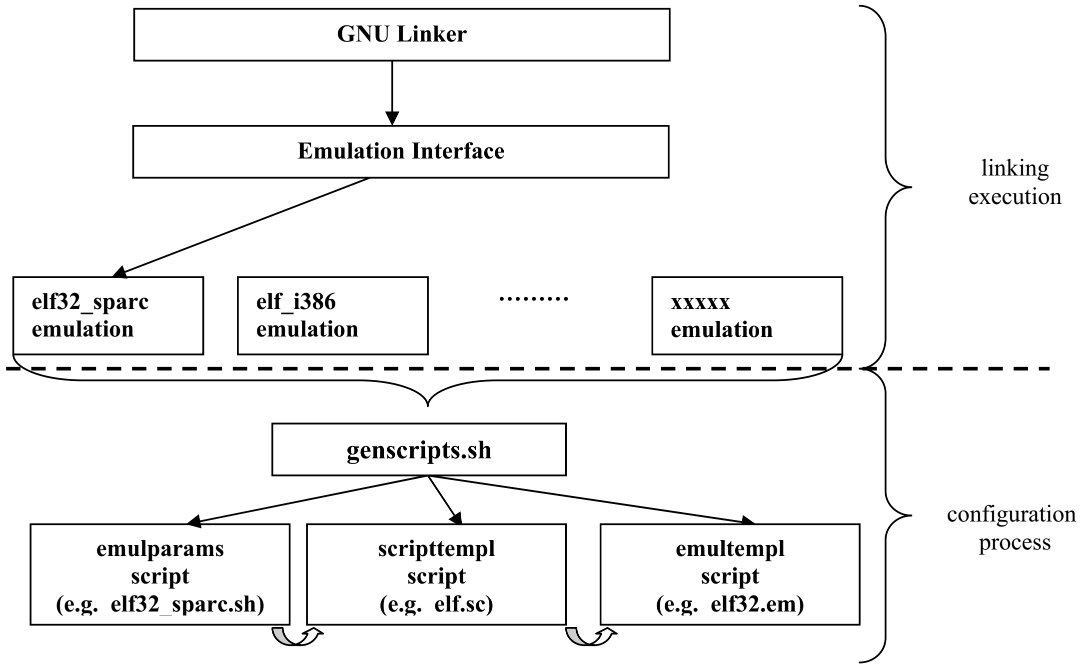


Figure 4.15: The GNU Linker Emulation

Each emulation consists of only one source file. The emulation file is generated by some scripts during the configuration process of the GNU linker. Figure 4.15 demonstrates how the emulation is generated at the configuration time and how it is used at the execution time. At the configuration time, the `genscripts.sh` is invoked with the name of the *emulparams script* as parameter. The `genscripts.sh` script will in turn invoke three script files, namely the *emulparams script*, the *scripttempl script* and the *emultempl script*. The emultempl script file to be used is given in the input parameter, while the scripttempl script and the emulparams script are indicated in the content of the emultempl script file. The emulparams script file is target-dependent while both of the scripttempl script and the emultempl script varies with the binary file format only.

elf32_sparc.sh

```
SCRIPT_NAME=elf
OUTPUT_FORMAT="elf32-sparc"
TEXT_START_ADDR=0x10000
MAXPAGESIZE=0x10000
COMMONPAGESIZE=0x2000
NONPAGED_TEXT_START_ADDR=0x10000
ALIGNMENT=8
ARCH=sparc
MACHINE=
TEMPLATE_NAME=elf32
DATA_PLT=
GENERATE_SHLIB_SCRIPT=yes
NO_SMALL_DATA=yes
```

elf_i386.sh

```
SCRIPT_NAME=elf
OUTPUT_FORMAT="elf32-i386"
TEXT_START_ADDR=0x08048000
MAXPAGESIZE=0x1000
COMMONPAGESIZE=0x1000
NONPAGED_TEXT_START_ADDR=0x08048000
ARCH=i386
MACHINE=
NOP=0x90909090
TEMPLATE_NAME=elf32
GENERATE_SHLIB_SCRIPT=yes
NO_SMALL_DATA=yes
```

Figure 4.16: The emulparams Script Files for SPARC-ELF and i386-ELF

The emulparams script is target-dependent such that each target configuration needs an unique emulparams script. The emulparams script carries a set of parameter values which will be used in the further emulation file generation. Figure 4.16 shows the content of the emulparams script for the SPARC-ELF and that for the i386-ELF in the current release of the GNU linker (*version 2.13*). The name of the scripttempl script is given in the `SCRIPT_NAME` parameter. In Figure 4.16, "`SCRIPT_NAME = elf`" appears in both emulparams scripts, so the scripttempl script file with the name of `elf.sc` is used. The scripttempl script file is responsible for the linker script generation. The scripttempl script file is basically only dependent on the binary file format. In this project, we simply assume fixing the `SCRIPT_NAME` parameter to be `elf` is enough. On the other hand, the name of the emultempl script is given in the `TEMPLATE_NAME` parameter. In Figure 4.16, "`TEMPLATE_NAME = elf32`" appears in both emulparams scripts, so the emultempl script file with the name of `elf32.em` is used. The emultempl script file is used to generate the content of the emulation file. Same as the scripttempl script file, the emultempl script file is dependent on the binary file format only. We fix the `TEMPLATE_NAME` parameter value as `elf32` in this project. Other emulparams script parameters carry some simple data such as the text segment start address (`TEXT_START_ADDR`.

In fact, some parameter values are required and some are optional in the emulation generation process. Please consult [**?**] for the detailed description of the emulparams script parameters.

Besides the BFD backend, the emulation is the other which is required to port the GNU linker to a new target. In this subsection, we conclude that the problem of automatic generating the emulation for a new ELF-target can be reduced to be that of automatic generating the emulparams script file.

4.2.4 The Link-editing Algorithm

Figure 4.17 is an overview of the link-editing methodology used by the GNU linker. The GNU linker takes two types of input - the linker script and the input object files. The linker script is provided by the target-dependent emulation as discussed in Section 4.2.3. Generally speaking, linking is *threading* the input object files together into one piece. The linker script provides the mapping rules that instruct the linker to map each section in an input object file to one of the sections in the output file. The names of the input object files are however provided from the input command. The linker will build a BFD object to represent each input object file, and they will be used in the linking process. Before the process starts, a BFD object is also constructed to represent the output object file, and a link hash table, that will be used for symbol resolution, is created through the backend target vector of the output BFD object. According to the user's request, the linking and relocation task is performed to generate a static/dynamic executable, a relocatable file or a shared library. When the link-editing task finishes, the output BFD is written out into to disk as a file.

The linking and relocation job performed by the GNU linker can be summarized into the following steps:

1. **Processing through each input object file.**

   After a BFD object is created for the input file, two tasks will be done at each iteration. First, the global symbols in the input BFD will be added to the link hash table. The symbol adding request will be redirected to the appropriate function in the BFD backend.

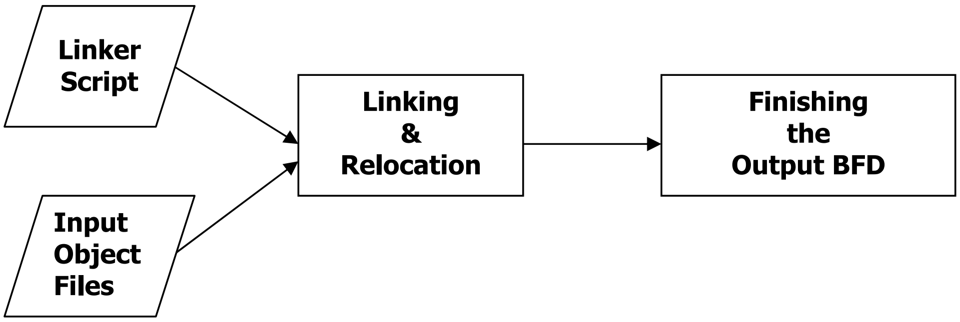


Figure 4.17: The Overview of the GNU Link-editing Methodology

For ELF files, a symbol may come from the `.symbtab` section or the `.dynsymb` section. If the input is a shared library, symbols from the `.dynsymb` (dynamic symbol table) will be processed and they are called *dynamic symbols*. Remember that symbol resolution is performed on the fly when the symbols are added to the link hash table. Second, if the input is not a shared library, the linker will check through all relocation entries in the file. The purpose is to allocate space in the GOT and the PLT of the output when GOT/PLT relocation entry is found.

2. **Garbage collecting unreferenced input sections.**

   Starting from the necessary sections as indicated by the linker script, the input sections required in the output file are marked. A required section is one which defines any referenced symbol in the already marked sections. In the end, the sections that are not marked will be marked as `SEC_EXCLUDE`, which indicates that the content of the section can be excluded by the linker.

3. **Mapping input sections to output sections.**

   By using the rules provided by the linker script. Each input section is mapped to a section in the output file. After this step, the input sections will know the output section in which they will be placed, but not the exact location.

4. **Preparing for section sizing.**

   It is not always the case that the content of an output section comes from the input sections. Dynamic sections such as `.plt` and `.got` are newly created by the linker. At this step, target-dependent functions from the BFD backend are invoked to find out the size of the newly created dynamic sections, even though the content may be filled later.

5. **Sizing up the sections.**

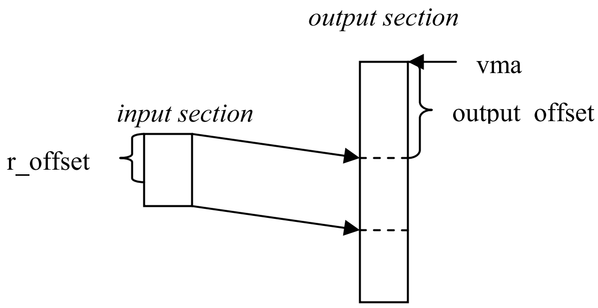


Figure 4.18: The Semantics of the Relocation Formula Carried in a `HOWTO`

   At this step, the start address and the size of each output section will be set. At the same time, each input section is assigned with the *output offset*, which is the offset from the start of the corresponding output section. After this step, the input sections not only know which output section they will be placed, but also the exact location they will be placed. Figure 4.18 illustrates this scenario.

6. **Processing input relocations.**

   Before this step, all the symbol values are known since every input sections have an assigned location its output section. The relocation entries in the input object files are processed. The relocation task is performed by functions in the BFD library. As relocation handling is target-dependent, corresponding backend function in the BFD backend will be invoked. This step also responsible for filling entries (PLT/GOT) in the newly created dynamic sections.

7. **Writing out the output BFD object.**

   Before writing the output BFD object, the content of the dynamic sections will be filled.

.

4.3 Retargetting Methodology

In the previous two sections, we analyze the architecture of the GNU BFD library and that of the GNU linker. It has been concluded that it is necessary to automatically generate the `elf32-`*`myarch`*`.c` file to provide a new BFD backend and the emulparams script file to provide the new GNU linker emulation.

4.3.1 Generation of the `elf32-`*`myarch`*`.c` File

As mentioned in Section 4.1.3, a new BFD target vector is created with the ELF-target template when we write the necessary information to the `elf32-`*`myarch`*`.c` file. The information in the `elf32-`*`myarch`*`.c` file is to fill the processor-independent details needed by the ELF-target template to create the BFD target vector, or it overrides some default setting filled by the ELF-target template. We may have to override some default values initialized by the ELF-target template if the new target has different behavior comparing to most ELF-targets do.

We study some elf32-*myarch*.c files for the supported targets in the BFD library, and derive a methodology to create the elf32-*myarch*.c file in a systematic manner. The elf32-*myarch*.c file generated by our method provides the most basic linking support such that there is no link-time optimization and no special feature is supported.

In the following, we describe our method to generate a elf32-*myarch*.c file for a new ELF-target. Constructing the BFD target vector by using the ELF-target template consists of three parts: 1) setting/overriding parameter values, 2) defining the relocation types, and 3) defining/overriding the functions implementing the backend API.

Setting/overriding Parameter Values

First of all, Table 4.3 summarizes the parameters that have to be set in the file. The last column of the table also lists the values that can be used to generate the corresponding parameter value from our processor model. The `TARGET_BIG_SYM` or the `TARGET_LITTLE_SYM` should be set with the name of BFD target vector being constructed. The `TARGET_BIG_SYM` is defined if the target is a big-endian machine; the `TARGET_LITTLE_SYM` is defined otherwise. Similarly, the `TARGET_BIG_NAME` or the `TARGET_LITTLE_NAME` gives the name that is used to identify the target in BFD library. The name should follow the convention of *elf32-myarch*. If the relocation addend value is not stored in the relocation entry in this target, we have to set `USE_REL` to be 1. Otherwise, we can ignore this parameter. The `ELF_ARCH` carries the enumerator value that identifies the processor architecture in the BFD library. When adding a new target, we have to add the definition of this enumerator in the the corresponding file. The `ELF_MACHINE_CODE` should be set to be the ELF machine value of the architecture. The `ELF_MAXPAGESIZE` gives the maximum page size supported in byte. The rest of the parameters are used by the ELF backend. The `elf_backend_can_gc_sections` acknowledges the ELF backend whether functions for garbage collecting unreferenced sections will be provided by the target. We default it to be 1 as we have method to generate the corresponding functions automatically. The `elf_backend_can_refcount` corresponds to the usage of the ELF backend data in the target functions that will be defined in this file. The `elf_backend_plt_readonly` is set to be 1 if the procedure linkage table resides in the text segment. The `elf_backend_want_plt_sym` is set to be 1 if there is access symbol for the PLT. The `elf_backend_got_header_size` and the `elf_backend_plt_header_size` defines the start entry size of the GOT and the PLT in byte.

Parameter	**Description**	**Input**
`TARGET_BIG_SYM` *or* `TARGET_LITTLE_SYM`	name of the BFD target vector	*isa.cpu* *isa.bigEndian*
`TARGET_BIG_NAME` *or* `TARGET_LITTLE_NAME`	type of the target	*isa.cpu* *isa.bigEndian*
`USE_REL`	sets to 1 if addend is not stored in relocation entry	*abi.reloca*
`ELF_ARCH`	enumerator that identifies the architecture of this backend	*isa.cpu*
`ELF_MACHINE_CODE`	ELF machine code value	*abi.e_mach*
`ELF_MAXPAGESIZE`	maximum page size in byte	*abi.maxPageSize*
`elf_backend_can_gc_sections`	sets to 1 if garbage collecting unreferenced input section is supported	1
`elf_backend_can_refcount`	sets to 1 if counting number of reference for GOT/PLT symbols	1
`elf_backend_want_got_plt`	sets to 1 if `.got.plt` section is needed	*abi.plt.inTextSeg*
`elf_backend_plt_readonly`	sets to 1 if PLT is `.plt` is a readonly section	*abi.plt.inTextSeg*
`elf_backend_want_plt_sym`	sets to 1 if there is PLT access symbol	*abi.plt.accessSymb*
`elf_backend_got_header_size`	size of the GOT header (start entry) in byte	*abi.got.startEntries*
`elf_backend_plt_header_size`	size of the PLT header (start entry) in byte	*abi.plt.nonPICLayout.startEntry*

Table 4.3: The Parameters defined in `elf32-`*`myarch`*`.c`

Defining the Relocation Types

```
typedef struct reloc_howto_type {
    unsigned int            type;           // enumerator
    unsigned int            rightshift;
    unsigned int            size;           // the size to be relocated
    unsigned int            bitsize;
    unsigned int            pc_relative;    // making PC relative value?
    unsigned int            bitpos;         // bit pos of the reloc value
    enum complain_overflow  complain_on_overflow;
    bfd_reloc_status_type   (*special_function)
      PARAMS ((bfd *, arelent *, struct symbol_cache_entry *,
               PTR, asection *, bfd *, char ** ));  // behavior function
    char                    *name;          // relocation name
    boolean                 partial_inplace; // modify the relocation entry
    bfd_vma                 src_mask;       // mask to extract the addend
    bfd_vma                 dst_mask;       // mask to put the relocation value
    boolean                 pcrel_offset;
    }
```

Figure 4.19: Definition of `struct reloc_howto_type`

The relocation types are defined in a table which is an array of `struct reloc_howto_type` elements. Each element in the table abstracts one relocation type of the architecture. Because each table element is defined by macro called `HOWTO`, we also call it a HOWTO relocation table. The definition of `struct reloc_howto_type` is in Figure 4.19. A `HOWTO` macro will directly fill each member in a `struct reloc_howto_type`.

A relocation HOWTO entry may be used by a generic relocation algorithm in BFD to perform relocation. The method in Figure 4.20 shows how such generic relocation algorithm will perform relocation with the information carried in the HOWTO entry. There are 3 input arguments: the `howto` is the HOWTO entry of the type of relocation that will be performed, the `addend` carries the addend value of the processing relocation, and `value` is the partial relocation value provided to the method. For most cases, the `value` would carry the resolved symbol value. In the method, if `special_function` is provided in the HOWTO, it will be called to perform relocation. This approach is used to handle the relocation behavior that cannot be described by only setting values in the HOWTO entry. Otherwise, the relocation value is calculated by 3 steps. First, it adds the `addend` and the `value`. Second, if `howto.pc_relative` is true, it may generate one of the two types of PC relative value. If `howto.pcrel_offset` is true, the relocation value is relative to the where the relocation is applied; it is relative to the start address of the input section otherwise. Third, the relocation value is right-shifting by `howto.rightshift`. In the end, the relocation value will be placed at the relocated field. It is done by left-shifting the relocation value by `howto.bitpos`, which is the bit position of the relocation value in the destination, and masking the value to the destination by `howto.src_mask`.

```
void relocation_with_howto(struct reloc_howto_type howto, bfd_vma addend, bfd_vma value, ) {
    bfd_vma      relocation;

    if(howto.special_function)
        howto.special_function(...value...);
    else {
        relocation = value + addend;

        if( howto.pc_relative ) {
            if( howto.pcrel_offset )
                relocation -= currPC;
            else
                relocation -= start_addr_of_input_sec;
        }

        relocation = relocation >> howto.rightshift;

        /* place the relocation value to the relocated field */
    }
}
```

Figure 4.20: The Use of HOWTO Relocation Entry

To generate the relocation HOWTO table automatically, we have to be able to translate each *Reloc* in our processor model into a relocation HOWTO entry. Among the 9 kinds of relocations categorized by our model, only 5 are processed by the link-editor. They are NONE, DATA, FUNC, GOT and PLT. The function that will use the generic relocation function, the behavior of which is described in Figure 4.20, will be defined in the elf32-*myarch*.c file also. By using the notation in Table 3.1, we define that the `value` argument of `relocation_with_howto()` is `S` if the relocation type is either DATA or FUNC, it is `G` if the relocation type is GOT, and it is `L` if the relocation type is PLT. With this definition, we can map a *Reloc* member to a HOWTO entry by using the algorithm described in Figure 4.21.

```
struct reloc_howto_type map_Reloc_to_howto( Reloc reloc ) {
    struct reloc_howto_type howto;

    howto.type = reloc.name;

    if (reloc behavior cannot be expressed with HOWTO parameters)
        howto.special_function = genRelocBehFunc(reloc);
    else {
        howto.rightshift = right shift amount applied on 'relocValue' or
                           'gotRelocValue' in reloc.symb.operation;

        /* according to the BFD library interpretation */
        if (reloc.symb.extSize <= 8)
            howto.size = 0;
        else if (reloc.symb.extSize == 16)
            howto.size = 1;
        else if (reloc.symb.extSize <= 32)
            howto.size = 2;

        howto.bitsize = reloc.symb.extSize;

        howto.pc_relative = reloc.symb.pcRel;

        howto.bitpos = reloc.symb.rfield.field.accessor.pos.right;

        if(!reloc.symb.rfield.checkOverflow)
            howto.complain_on_overflow = complain_overflow_dont;
        else if(reloc.symb.rfield.field.signed)
            howto.complain_on_overflow = complain_overflow_signed;
        else
            howto.complain_on_overflow = complain_overflow_bitfield;

        howto.name = reloc.name;

        if (addend is stored in relocation entry) {
            howto.partial_inplace = false;
            howto.src_mask = 0;
        }
        else {
            howto.partial_inplace = true;
            howto.src_mask = mask to extract the relocated field;
        }

        howto.dst_mask = mask to extract the relocated field;

        /* we support only one type of relative value */
        howto.pcrel_offset = reloc.symb.pcRel;
    }
    return howto;
}
```

Figure 4.21: The Mapping from a *Reloc* to a HOWTO Entry

Defining/overriding the Backend API

Table 4.4 summarizes the backend APIs that the elf32-*myarch*.c file should implement. Among the 15 APIs listed in the table, only `bfd_elf32_bfd_is_local_label_name` may not be generated, the rest are required, at least in our methodology. On the other hand, 10 out 15 of the APIs are target-dependent function that supports linking. We can see the importance of the BFD library to the GNU linker.

With our method, the *MYARCH*-ELF link hash table will be defined and used to help performing target-dependent task. Except its name, the definition of the *MYARCH*-ELF link hash table is the same for any architecture, it shows in Figure 4.22. The `dyn_relocs` in a link hash entry is a linked list of bookkeeping data, each of which describes a input relocation that may be copied as into the output file when the linker is creating a shared library. The hash table keeps the pointers of the output dynamic sections because they are always used in the target-dependent linking process.

```
struct elf32_myarch_dyn_relocs {
    struct elf32_myarch_dyn_relocs  *next;
    asection                        *sec;
    bfd_size_type                   count;
    bfd_size_type                   pc_count;
};

struct elf32_myarch_link_hash_entry
{
  struct elf_link_hash_entry elf;
  struct elf32_myarch_dyn_relocs *dyn_relocs;
};

struct elf32_myarch_link_hash_table
{
  struct elf_link_hash_table elf;

  asection *sgot;
  asection *sgotplt;
  asection *srelgot;
  asection *splt;
  asection *srelplt;
  asection *sdynbss;
  asection *srelbss;

  struct sym_sec_cache sym_sec;
};
```

Figure 4.22: The *MYARCH*-ELF Link Hash Table

Due to the space limitation, we will focus our discussion on the generation of three APIs only - `bfd_elf32_bfd_reloc_type_lookup`, `elf_backend_finish_dynamic_symbol` and `elf_backend_finish_dynamic_sections`.

```
reloc_howto_type *elf32_myarch_reloc_type_lookup(bfd_reloc_code_real_type code) {
    switch(code) {
        case BFD_RELOC_xxxx:
            return &_bfd_myarch_elf_howto_table[??];
        case ...
        ....
        ....
        }
}
```

Figure 4.23: The `elf32_myarch_reloc_type_lookup`

The `bfd_elf32_bfd_reloc_type_lookup` redirects call that lookups the relocation HOWTO entry with the BFD generic relocation value. It will be implemented by a function called `elf32_myarch_reloc_type_lookup()`, the structure of which is shown in Figure 4.23. With the BFD generic relocation code, it is to return the appropriate relocation entry in the HOWTO table. The BFD library tries to categorize relocations and assigns the same value to relocations with the same behavior regardless the processor type. For example, the `R_SPARC_NONE` (in SPARC) and `R_386_NONE` (in i386) will be assigned with `BFD_RELOC_NONE` in the BFD. This categorization job is maintained by the developers who port the BFD library to a new architecture. In fact, since they are both ISA-dependent and ABI-dependent, relocations of any two different architectures are almost never one-to-one corresponding to each other in terms of behavior. Most of the common relocation types have already had their BFD generic relocation code defined. For example, `BFD_RELOC_NONE` for relocations with no job and `BFD_RELOC_8` for relocations responsible for 8-bit data relocation. When generating this algorithm, we will try to assign each relocation type with a pre-defined BFD generic relocation code. If it fails to do so, we will define a new code in the corresponding file. For example, we have to define a new code named as `BFD_RELOC_SPARC_WDISP22` for the `R_SPARC_WDISP22` relocation type in SPARC.

The `elf_backend_finish_dynamic_symbol` redirects call to a target-dependent func-

```
boolean elf32_myarch_finish_dynamic_symbol(
    bfd *output_bfd, struct bfd_link_info *info,
    struct elf_link_hash_entry *h, Elf_Internal_Sym *sym) {

    if( the symbol needs PLT entry ) {
        create one PLT normal entry;
        fill the appropriate value to the new entry;
        create appropriate 'JMP_SLOT' relocation entry for the entry;
        }

    if( the symbol needs GOT entry ) {
        create a new GOT entry;
        create appropriate 'GLOT_DAT' or 'RELATIVE' relocation entry
            and fills the GOT entry address to the relocation
        }

    if( the symbol needs copy relocation ) {
        create a COPY relocation entry for the symbol
        }
    }
}
```

Figure 4.24: The `elf32_myarch_finish_dynamic_symbol`

tion to fill in the information of a dynamic symbol to the appropriate dynamic section. The pseudo code of the expected implementation of this function is shown in Figure 4.24. This function is called before the link editor is writing out the dynamic symbol to the `.dynsym` section. The symbol is checked whether a PLT entry, a GOT entry or a copy relocation has to be created for the symbol. If a PLT entry is needed, a PLT normal entry is created according to the the normal PLT entry described in our processor model. This can be found from *abi.Plt.nonPICLayout.normalEntry* and/or *abi.Plt.PICLayout.normalEntry*.

The `elf_backend_finish_dynamic_sections` redirects call to a target-dependent function to finalize the content of all dynamic sections. The pseudo code of the function implementation is shown in Figure 4.25. There are 3 tasks in this function. The first task is to setup the value to some special dynamic entry. The value in the `DT_PLTGOT` is architecture dependent. The correct setup value is abstracted in a *Dyn* member in the processor model. The `DT_RELSZ` entry is handled only if the target architecture doesn't use `addend` in their relocation entries. The second task is to finish up the procedure linkage table by filling the start entry and the end entry of PLT if there is any. The code for this part is generated from the *Plt* member in the *abi*. It is possible that the PLT have two different layouts. Also, the byte order

```
boolean elf32_myarch_finish_dynamic_sections(
    bfd *output_bfd, struct bfd_link_info *info) {

    // first task
    for each 'dyn' in dynamic entries {
        case DT_PLTGOT:
            put the PLT or GOT section virtual addr. to the 'dyn'
        break;

        case DT_JMPREL:
            put the '.relplt' section virtual addr. to the 'dyn'
        break;

        case DT_PLTRELSZ:
            put the '.relplt' section size to the 'dyn'
        break;

        #if !RELOCA
        case DT_RELSZ:
            put the '.relplt' section size to the 'dyn'
        break
        #endif
        }

    // 2nd task
    if (PLT is not empty) {
        fill the content of start entry;
        fill the content of end entry;
        update the PLT size;
        }

    // 3rd task
    if (GOT is not empty) {
        fill the first entry of GOT
        record the virtual addr. to the appropriate ELF data;
        }
    }
}
```

Figure 4.25: The `elf32_finish_dynamic_sections`

problem has to be considered when generating the filling. At the end, the global offset table is handled with the same way that the first entry of the GOT is filled with there is any. The code will be generated from the *Got* member in the *abi*.

BFD Backend API	**Description**
`elf_info_to_howto` *or* `elf_info_to_howto_rel`	- lookups the relocation type (a HOWTO entry) with the architecture-dependent relocation value
`bfd_elf32_bfd_reloc_type_lookup`	- lookups the relocation type (a HOWTO entry) with the BFD generic relocation value
`elf_backend_reloc_type_class`	- given a relocation entry, distinguishes whether it is a RELATIVE, PLT or COPY reloation
`bfd_elf32_bfd_is_local_label_name`	- given a symbol name, checks whether it is a local symbol - it's needed only if the architecture defines more local symbol prefixes other than those used by ELF files
`bfd_elf32_bfd_link_hash_table_create`	- constructs a link hash table - it's needed if *MYARCH*-ELF link hash table is defined
`elf_backend_copy_indirect_symbol`	- copys data from an indirect symbol hash entry to its corresponding direct symbol hash entry - it's needed if *MYARCH*-ELF link hash table is defined
`elf_backend_create_dynamic_sections`	- creates the dynamic sections required for the linking output
`elf_backend_check_relocs`	- checks through all relocations in a input section and allocate space in the GOT and the PLT of the linking output - used by step 1 in Section 4.2.4
`elf_backend_gc_mark_hook`	- returns the section that defines the given symbol - used by step 2 in Section 4.2.4
`elf_backend_gc_sweep_hook`	- discards bookkeeping information that was collected for a excluded input section - used by step 2 in Section 4.2.4
`elf_backend_adjust_dynamic_symbol`	- adjusts a dynamic symbol before sizing output sections in the linking - used by step 4 in Section 4.2.4
`elf_backend_size_dynamic_sections`	- sizes up the dynamic sections created for the linking output - used by step 4 in Section 4.2.4
`elf_backend_relocate_section`	- processes relocation entries in an input object file for linking - used by step 6 in Section 4.2.4
`elf_backend_finish_dynamic_symbol`	- fills the information of a dynamic symbol to the appropriate dynamic section (e.g. `.plt` and `.got`) - used by step 7 in Section 4.2.4
`elf_backend_finish_dynamic_sections`	- finializes the content of all dynamic sections - used by step 7 in Section 4.2.4

Table 4.4: The Backend APIs defined in `elf32-`*`myarch`*`.c`

4.3.2 Generation of the Linker Emulparams Script File

To generate the linker emulparams script file, we find that filling values for a fixed set of parameters is enough. Figure 4.26 and Figure 4.27 present the emulparams script files generated from the processor model for SPARC and i386. The mapping from the processor model to the values is straightforward.

```
SCRIPT_NAME=elf
OUTPUT_FORMAT="elf32-sparc"
TEXT_START_ADDR=0x10000
MAXPAGESIZE=0x10000
NONPAGED_TEXT_START_ADDR=0x10000
ALIGNMENT=8
ARCH=sparc
MACHINE=
TEMPLATE_NAME=elf32
DATA_PLT=
NOP=0x01000000
GENERATE_SHLIB_SCRIPT=yes
NO_SMALL_DATA=yes
```

Figure 4.26: `elf32_sparc.sh`

```
SCRIPT_NAME=elf
OUTPUT_FORMAT="elf32-i386"
TEXT_START_ADDR=0x8048000
MAXPAGESIZE=0x1000
NONPAGED_TEXT_START_ADDR=0x8048000
ALIGNMENT=4
ARCH=i386
MACHINE=
TEMPLATE_NAME=elf32
NOP=0x90909090
GENERATE_SHLIB_SCRIPT=yes
NO_SMALL_DATA=yes
```

Figure 4.27: `elf32_i386.sh`

Chapter 5

Retargetting a Micro-architecture Simulator

In this chapter, we discuss the methodology of generating retargetable processor simulators by automatically porting the SimpleScalar toolset. After being ported, the SimpleScalar toolset can provided processor simulators ranging from simple functional simulators to detailed micro-architecture simulators.

In the text that follows,an anatomy of the latest SimpleScalar toolset is first described, followed by our retargetting methodology.

5.1 The SimpleScalar Toolset

The SimpleScalar toolset [17] is an infrastructure developed at University of Wisconsin for micro-architectural modeling and simulation. The current release (version 3.0) provides 7 simulators at different level of micro-architecture detail, as summarized in Figure 5.1.

Micro-architectural detail

Simulator	Descirption	#line
`sim-fast`	Simple functional simulator	402
`sim-safe`	Speed-optimized functional simulator	307
`sim-profile`	Functional simulator with profiling	812
`sim-cache`	Hierarchical memory simulator	782
`sim-cheetah`	Single-pass multi-configuration cache simulator	479
`sim-bpred`	Customizable branch prediction simulator	513
`sim-outorder`	Detailed micro-architectural simulator with dynamic instruction scheduler and multi-memory hierarchy	4555

Figure 5.1: SimpleScalar Simulators

5.1.1 Supported Architectures

The Portable Instruction Set Architecture (PISA), which is a derivate of the MIPS architecture, is the primary architecture supported by SimpleScalar. Besides, Alpha is another architecture supported in the current release.

Supporting only one real processor architecture (Alpha) until the recent release, the SimpleScalar team has been spending a lot of efforts to manually port the SimpleScalar infrastructure to other processor architectures.

5.1.2 Infrastructure

Figure 5.2 shows the infrastructure of the SimpleScalar toolset. The target processor environment is modeled in terms of 3 aspects - ISA, ABI and micro-architecture. The micro-architecture modeling depends on the detailed level of the simulation, and it is included within the simulator engine. The modeling of ISA and ABI is however simulator-independent; it is the most fundamental required to construct a software processor. For example, a functional simulator only requires the model of ISA and ABI running on a simple *fetch-decode-execute* loop for each instruction, while a micro-architecture simulator adds the micro-architecture behavior to the simulation loop. In the follows, we will discuss the architecture modeling in SimpleScalar.

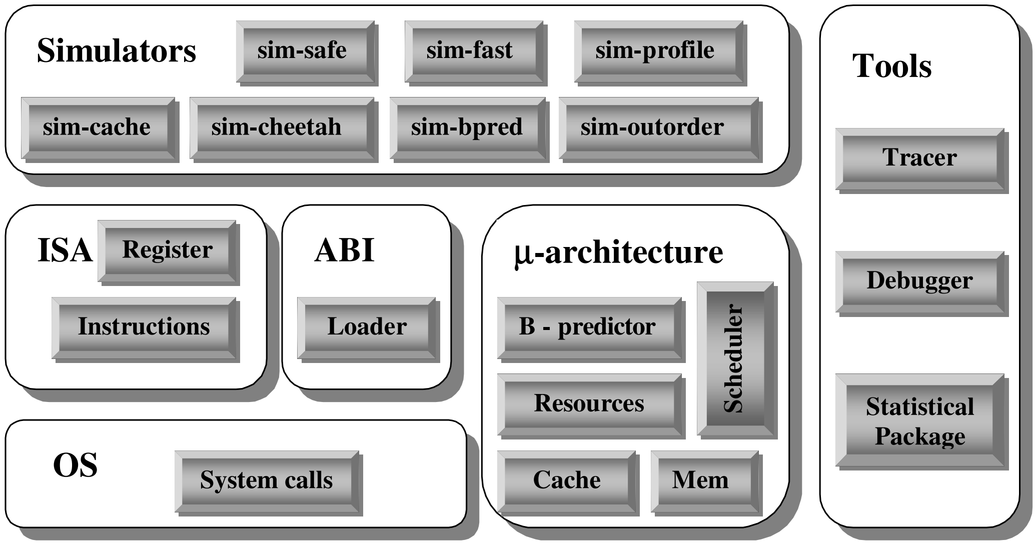


Figure 5.2: SimpleScalar Infrastructure

ISA-dependent Code

The register and instruction definitions abstract the ISA.

In any simulator engine, all of the simulated target processor registers are collectively carried in `regs`, which is a `struct regs_t` member (Figure 5.3). The `regs` carries the integer register file, the floating point register file, all control registers as well as the program counter and next program counter. The processor model will define the `md_gpr_t`, the `md_fpr_t`, the `md_ctrl_t` and the `md_addr_t` data structures to define the structures of the integer register file and floating point register file, the control registers and the size of program counter respectively.

The simulator engine will define its own register access macros. This is to simulate the micro-architecture behavior. For example, a functional simulator may have register access macros as simple as directly accessing the target register in `regs`, but a micro-architecture simulator with speculative execution property may carry a speculative copy of registers and the access macros have to get value from the correct copy (`regs` or `spec_regs`) based on the execution mode.

```
struct regs_t {
    md_gpr_t  regs_R;      // integer register file
    md_fpr_t  regs_F;      // floating point register file
    md_ctrl_t regs_C;      // control registers
    md_addr_t regs_PC;     // program counter
    md_addr_t regs_NPC;    // next program counter
    };
```

Figure 5.3: Register Definition

The instruction set model include the a set of instruction definition and a software decoder. Each instruction definition has a pre-defined (or recommended) format; each of which contains the assembly format, execution unit component, register dependency information and instruction type as well as the semantic action statement which mimics the effect on the processor state. This is as defined in Figure 5.4 and an example is described in Figure 5.5. The `<enum>` is an enumerator that is the decoded value returned by the software decoder to access the instruction definition. The `<opname>` and `<assembly fmt>` defines the assembly format, which is used by the SimpleScalar debugger. The functional units available in the target processor are listed in architecture model for micro-architecture simulation, and the `<func unit>` identifies the functional unit used to execute the instruction. The `<inst type>` is an integer flag which describe the properties of the instruction. The `<reg dependency>` consists of a list of designators, each of which gives the register input/output dependency information. The dependency information is useful for instruction scheduling in out-of-order execution.

Access macros are also defined to access the instruction fields, and they are simulator-independent and unique to each processor model. When expressing the instruction semantics, accesses to the registers, memory and instruction fields are expressed with the corresponding access macros. The purpose is to eliminate the dependency between the instruction semantic action and the micro-architecture simulation. For example, memory access request in a cache simulator may be served by accessing the memory hierarchy while that in a functional simulator may accessing the simulated memory directly. Handled by the access macros defined in the simulator engines, all of these details will be transparent to the instruction semantics.

The instruction identification job is handled by a software decoder. At each simulation iteration, the instruction is decoded and an enumerator is generated by the software decoder. The generated enumerator is used to access the instruction definition and semantics in a big switch table.

Other information in the ISA includes the address size and the instruction length.

```
DEFINST( <enum>, <opname>,
         <assembly fmt>, <func unit>,
         <inst type>, <reg dependency> )

#define <enum>_IMPL {
  <semantic action>
}
```

Figure 5.4: Instruction Definition

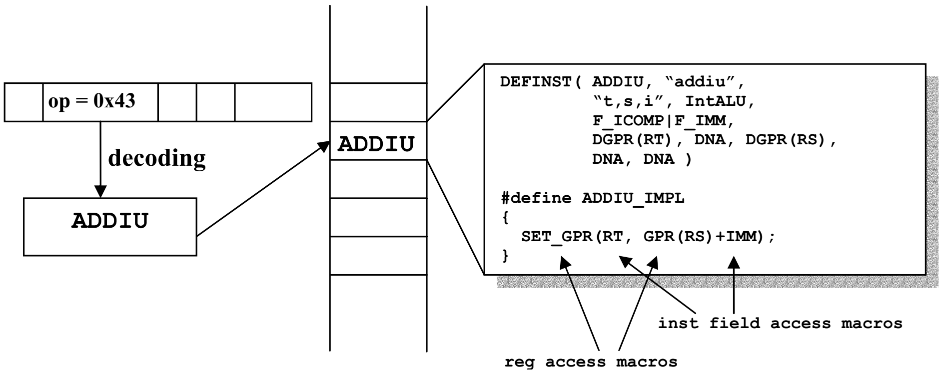


Figure 5.5: Instruction Definition Example

ABI-dependent Code

Before the simulation starts, the software program loader copies the instructions reading from the input executable into the simulated memory, and then initializing the execution environment (e.g. stack). The input is assumed to be statically linked executable and dynamic linking is not supported by the software program loader. In order to correctly interpret the binary stream, the software program loader requires understanding the binary file format used by the architecture.

Generally, the SimpleScalar simulators use the provided `COFF` file loader to load the PISA executables, although these exists another choice of using the GNU's BFD library.

Operating System-dependent Code

The behavior of trap instructions (system calls) is emulated by the system call package. According to the trap number carried in a destinated simulated register (e.g. `GPR(2)` in PISA), the appropriate handling is selected and the required input argument values are obtained from the simulated registers. The equivalent system call is then made at the host machine, and the returned value is copied back to the simulated registers if there is any. Similar to accessing registers and memory, the system call handler macro is defined in the simulator engine because system call should not be made when simulating speculative execution.

Other OS information includes the memory page size, which is used to mimics the memory behavior.

Micro-architecture-dependent Code

The micro-architecture affects the behavior of the simulator, so it is included in the simulator engine. The micro-architecture model can be constructed with the components provided by the SimpleScalar, including cache, memory, functional unit resource, scheduler and branch predictor. As part of the micro-architecture, the memory hierarchy is simulator-dependent and memory access macros are therefore provided by the simulator.

Figure 5.6 summarizes how a simulator engine looks like. Any simulator engine must provide 3 ports, which are micro-architecture-dependent, to access the registers, the memory hierarchy and the system call package.

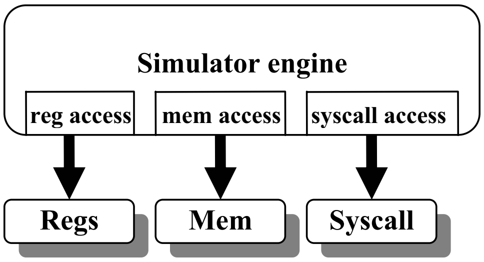


Figure 5.6: The 3 Access Ports at a Simulator Engine

5.1.3 The Simulation Flow

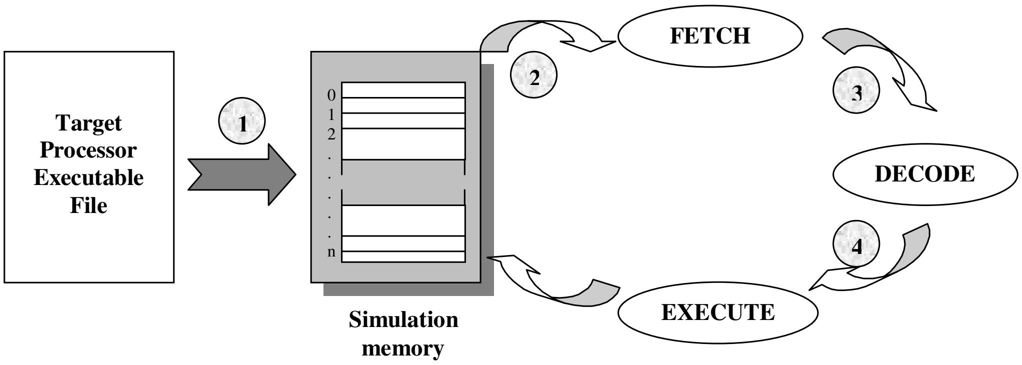


Figure 5.7: The Simulation Process

The SimpleScalar simulators are interpretation-based. The simulation flow of the simplest unpipelined functional simulation is illustrated in Figure 5.7. **(1)** Before the simulation starts, the instruction and data are loaded to the simulated memory. The program counter (PC) holding at a simulated register is initialized with the program entry point address after the input binary data is loaded. The simulator then loops through the *fetch-decode-execute* process at each instruction simulation. **(2)** At fetch, the instruction addressed by the PC register is copied from the simulated memory to the simulated instruction register (IR). **(3)** The fetched instruction is then decoded. **(4)** By using the decoded value, the appropriate instruction semantics definition is selected and executed.

5.2 Retargetting Methodology

In this section, we present the methodology to retarget the SimpleScalar toolset from the embedded processor model described in Chapter 3. In the following subsection, we will discuss our method to retarget or handle each processor-dependent part in the SimpleScalar framework, including the software program loader, the registers, the instructions, the software instruction decoder and the system call emulation. Due to the space limitation, the discussion is not complete.

5.2.1 Retargetting Software Program Loader

The software program loader is responsible for loading the input executable stream into the simulated memory and initializing the execution environment.

Program Loading

The software program loader requires the understanding of the binary file format in order to load the instructions and appropriate data to the simulated memory. Not all binary data has to be copied to the memory. but only those in the loadable sections is copied. Therefore, the software program loader has to know how to extract the loadable sections from the input stream and where the section data should be loaded. We retarget the program loading task by the retargetable BFD library as discussed in Chapter 4. By using the retargeted BFD library generated from the same processor model, program loading can be done by an architecture-independent algorithm, which is given in Figure 5.8. The algorithm is written with the BFD API. Given the executable file name (`input_file`) and the architecture BFD target name (e.g. `elf32-sparc`), the binary data is read by the BFD library. If the BFD library is able to support the target architecture, a BFD object representing the input file will be returned. Then, the algorithm loops through all the sections in the file. The loadable section can be detected by checking if the section flag indicates that memory for the section data

```
void load(char *input_file) {
    bfd             *abfd;
    asection        *sect;
    char            *p;
    struct mem_t    *mem;   // simulated memory

    /* e.g. <arch bdf target name> = "elf32-sparc" */
    abfd = bfd_openr(input_file, <arch bfd target name>);

    /* read all sections in file */
    for( sect = abfd->sections; sect; sect=sect->next ) {
        /* memory should be allocated for this section when loading */
        if ( bfd_get_section_flags(abfd, sect) & SEC_ALLOC ) {

            /* allocate memory to get section content */
            p = calloc(bfd_section_size(abfd, sect), sizeof(char));

            /* get the section content */
            bfd_get_section_contents( abfd, sect, p, 0,
                                      bfd_section_size(abfd, sect) );

            /* copy the section content to the simulated memory at
               the specified virtual memory */
            mem_bcopy( mem_access, mem, Write,
                       bfd_section_vma(abfd, sect), p,
                       bfd_section_size(abfd, sect) );

            free(p);
        }
    }
}
```

Figure 5.8: The Architecture-independent Program Loading Algorithm using BFD

should be allocated (`SEC_ALLOC`). For each loadable section, memory as large as the section size (`bfd_section_size()`) is allocated, and we request loading the section data to the allocated memory by calling `bfd_get_section_contents()`. Then, the data is copied to the simulated memory at the specified virtual memory address, which is obtained from `bfd_section_vma()`. The `mem_bcopy` function is used to access the simulated memory in SimpleScalar.

Setting Up Execution Environment

The execution environment has to be initialized before the simulation starts. It includes setting up the stack at the simulated memory and initializing the register state.

With the *stack* (stack abstraction) of the *abi* member of the processor model, program code is generated to copy the command-line arguments and environment data to the appropriate place of the simulated memory. Figure 3.14 shows how the stack is initialized according to the architecture-dependent parameters in the model. After initialization, the stack pointer value will be stored to the simulated stack pointer register, which is indicated in the stack abstraction as well.

In addition to the stack pointer register, other simulated registers that have to be initialized include the program counter (PC), the next program counter (NPC) and the register window control registers if there is any windowed registers in the architecture. In the *ctrls* of the *isa* member of the model, the PC and NPC register abstractions are identified by their `kind`. The simulated PC register should be initialized with the address of the start instruction. Through the BFD API, the start address can be obtained from the BFD object, which is generated at program loading time. This is done by `bfd_get_start_address(abfd)`. Then, the NPC is initialized to hold the address of the next instruction, which is the sum of the start address and the instruction size as carried in *instrnSize* of the *isa* member. If any register file of the architecture has the windowed property, we have to initialize the corresponding control registers also. In the *WinRegs* of the abstraction of such register file, the window pointer register and the window invalid register are indicated and their initial values are also included in the model.

5.2.2 Retargetting Register Manipulation

There are two aspects of retargetting registers: 1) defining the structures of the register files and the control registers, and 2) generating register access macros for each simulator engine.

Defining Register Structure

As mentioned, the target processor registers are collectively carried in a `struct regs_t` member (Figure 5.3). In the SimpleScalar framework, it is assumed that the supported target processor has at most one integer register and at most one floating point register file. The structures of the integer register file, the floating point register file, the control registers and the size of PC and NPC are defined by the `md_gpr_t`, the `md_fpr_t`, the `md_ctrl_t` and the `md_addr_t` respectively.

A register file is abstracted by a *RegFile* in *rfiles* of the *isa* member of the processor model. For the integer register file, the `md_gpr_t` is defined as an array with the size as large as the number of physical registers required including the concern of the register window. Each element in the array represents a register, so the type of each array element is determined by the `gran` of the *RegFile*. If the register file doesn't have any windowed registers, the number of physical register is equal to `size` of the *RegFile*. Otherwise, the number of physical register corresponding to the windowed registers part is equal to $(window\_size - overlap) \times number\_of\_window$. To facilitate the simulation, we treat the overlapping registers between the first window and the last window as two different set of registers. When the window is shifted from the last window to the first window or vice versa, we copy the overlapping register values from one to another in the simulation. The register file index is directly corresponding to the array index of the simulated register file. However, if windowed registers are contained in the file, a mapping function must be generated to map the virtual register index to the array index to access the appropriate physical register in the simulated register file. Figure 5.9 is the definition of the SPARC general-purpose register file generated from model in Example 1. `MD_NUM_IREGS` is the number of simulated physical register. The `md_gpr_t` is defined to be an array of integer with the size of `MD_NUM_IREGS`. The `REAL_GPR_POS()` is the mapping of the virtual register index to the array index to access the appropriate simulated register. Similar approach is used to define the floating point register file.

```
/* -> size - 32
   -> windowed reg
        -> pos - <8, 31>
        -> depth - 32
        -> overlap - 8
*/
#define MD_NUM_IREGS    (8 + 32 * (24-8) + 8 )
typedef int md_gpr_t[MD_NUM_IREGS];

/* 'CWP' is current window pointer */
#define REAL_GPR_POS(N)      ( ((N)<8)? (N):(((CWP)*16)+N))
```

Figure 5.9: The Definition of the SPARC General-purpose Register File in SimpleScalar

The `md_ctrl_t` is defined as a structure of simulated control registers. Except the PC and NPC, a simulated control register is defined for each *CtrlReg* found in the *ctrls* of the *isa* member. The name and the size information obtained from the `cell` of a *CtrlReg* is used to generate the definition. Figure 5.10 is the definition of the control registers of SPARC.

The `md_addr_t` is the type of the PC and NPC. It represents the size required to hold an address of the target processor. The address size information can be obtained from the *addrSize* of *isa*.

```
typedef struct {
    sword_t psr;        /* processor state register */
    sword_t wim;        /* window invalid mask */
    sword_t tbr;        /* trap base register */
    sword_t y;          /* multiply/divide register */
    sword_t fsr;        /* floating-point state register */
    } md_ctrl_t;
```

Figure 5.10: The Definition of the SPARC Control Registers in SimpleScalar

Generating Simulator-dependent Register Access Macros

```
/* 1. functional simulation */
#define GPR(N)          (regs.regs_R[N])
#define WIM             (regs->regs_C.wim)

/* 2. speculative simulation */
#define GPR(N)  \
   BITMAP_SET_P(use_spec_R, \
     R_BMAP_SZ,(N)) \
     ? spec_regs_R[N] \
     : regs.regs_R[N] )
#define WIM     \
    (BITMAP_SET_P(use_spec_C, C_BMAP_SZ, 1)\
                 ? spec_regs_C.wim          \
                 : regs.regs_C.wim)
```

Figure 5.11: The Register Access Macros at Different Engines

To simulate different micro-architecture behavior, the register access macros are different at each simulator engine. However, the access macros in each simulator engine have the same pattern independent to the ISA. For example, Figure 5.11 shows two register access macros at each of the two different types of simulation. To do a basic functional simulation, the first set of macros is to directly access the values in `regs`. The second set of maros is however simulating the speculative execution behavior. Besides the `regs`, a speculative copy of registers is carried by `spec_regs`. Based on the execution mode, the access macros determine which set of registers from which the value should obtain.

At each simulator engine, the *SET* and *GET* access macro for each register file and each control register will be generated. If the control register consists of fields of processor status, which means that the `fields` of the *CtrlReg* is not empty, access macros are generated for each field to extract value from the parent register.

5.2.3 Retargetting Instructions

For each instruction, an instruction definition in the format as shown in Figure 5.4 must be generated. The instruction semantics is expressed in terms of the access macros of registers, memory and instruction fields. So, the retargetting task is started with generating instruction field access macros.

Generating Instruction Field Access Macros

An instruction field access macros is simulator-independent and responsible for extracting the field value from an instruction. This exactly matches with the instruction field accessor (*InstrnFieldAccessor*) of our processor model. Therefore, this is as simple as generating an instruction field access macro for each *InstrnFieldAccessor* in the model. According to the `pos` in an *InstrnFieldAccessor*, a mask is generated to extract the field value from an instruction. The generated access macros is named by the `name` in the *InstrnFieldAccessor*. Figure 5.12 is the definition of the instruction field access macros of SPARC generated from the model in Example 3.

```
#define INSN_OP                        ( (inst & 0xc0000000) >> 30 )
#define INSN_DISP30                    ( (inst & 0x3fffffff) )
#define INSN_RD                        ( (inst & 0x3e000000) >> 25 )
#define INSN_OP2                       ( (inst & 0x01c00000) >> 22 )
#define INSN_IMM22                     ( (inst & 0x003fffff) )
#define INSN_A                         ( (inst & 0x20000000) >> 29 )
#define INSN_COND                      ( (inst & 0x1e000000) >> 25 )
#define INSN_DISP22                    ( (inst & 0x003fffff) )
#define INSN_OP3                       ( (inst & 0x01f80000) >> 19 )
#define INSN_RS1                       ( (inst & 0x0007c000) >> 14 )
#define INSN_I                         ( (inst & 0x00002000) >> 13 )
#define INSN_ASI                       ( (inst & 0x00001fe0) >> 5 )
#define INSN_RS2                       ( (inst & 0x0000001f) )
#define INSN_SIMM13                    ( (inst & 0x00001fff) )
#define INSN_OPF                       ( (inst & 0x00003fe0) >> 5 )
```

Figure 5.12: The Definition of the Instruction Access Macros of SPARC

Defining an Instruction

Referring to Figure 5.4, the values of `<enum>`, `<opname>`, `<assembly fmt>`, `<func unit>`, `<inst type>`, `<reg dependency>` and `<semantic action>` are generated for each instruction definition. In Figure 5.13 is the definition of the SPARC `ldsb` instruction generated from the specification in Example 6. In the follows, we will discuss the methodology to generate each value.

```
#define LDSB_IMPL                              \
  {                                            \
    sbyte_t _result;                           \
    enum md_fault_type _fault;                 \
    word_t  _addr;                             \
                                               \
    _addr = GPR(INSN_RS1) + GPR(INSN_RS2);     \
    _result = READ_BYTE(_addr, _fault);        \
    if (_fault != md_fault_none)               \
        DECLARE_FAULT(_fault);                 \
    SET_GPR(INSN_RD, (word_t)(sword_t)_result); \
  }
DEFINST( LDSB, "ldsb",
         "", RdPort,
         F_MEM|F_LOAD|F_DISPRR,
         DGPR(INSN_RD), DNA,
         DNA, DGPR(INSN_RS1), DGPR(INSN_RS2))
```

Figure 5.13: The `ldsb` Instruction Definition of SPARC

- `<enum>`

  An unique enumerator is generated from the instruction mnemonic. The instruction mnemonic is obtained from the first word of the `asmFormat` of the *Instrn*. In the example, the `asmFormat` of the `ldsb` *Instrn* is `"ldsb [%1+%2],%0"`, and `LDSB` is used as the enumerator. The taken word is captialized. In addition, duplicated enumerator will be prevented.

- `<opname>`

  The `<opname>` value is directly taken from the mnemonic of the first word of the `asmFormat`. The same `<opname>` value in two different instructions is allowed.

- `<assembly fmt>`

  In SimpleScalar, the assembly format is used to instruct the debugger to do instruction disassembly. We use different approach to disassemble instructions and we can ignore the `<assembly fmt>` value.

- `<func unit>`

  The `<func unit>` value identifies the functional unit used to execute the instruction. We assume a fixed class of functional unit in any processor architecture, it is as listed in Figure 5.14. Each instruction will be assigned with one functional unit. The assignment decision is made from the analysis of the instruction behavior.

- `<inst type>`

  The `<inst type>` is a flag which describes the properties of the instruction. The fixed definition of instruction flags is in Figure 5.15. The `<inst type>` is a combination of all instruction flags with matched properties. Similar to the generation of the `<func unit>`, the `<inst type>` flag is generated through analysis of the instruction behavior.

- `<reg dependency>`

  The `<reg dependency>` is a list of designators which give the register input/output dependency information. In Figure 5.13, the first 2 designators represent the output dependency while the rest of 3 represent the input dependency. The name of dependency designator starts with `D` and followed by the corresponding *GET* register accessor name. The register dependency designators are defined in the simulator engine if needed, The `sim-outorder` engine is an example. The register fields used as operands and destination are identified by the `srcs` and the `dst` in the *InstrnBeh* member. So, dependency information can be easily generated from the abstraction. However, the control registers involved in the operation are not explicitly included in the `srcs` and the `dst`. We have to traverse instructions in the `beh` function of the *InstrnBeh* member, in order to collect the dependency information of the control registers.

- `<semantic action>`

  The `<semantic action>` is a set of statements that mimics the effect of the instruction on the processor state. This is generated from the *InstrnBeh* member of the instruction. According to the behavior `kind`, simulation of the implicit characteristics is generated. For example, invocation to the system call emulation function is generated for trap instructions. The explicit behavior is abstracted in the `beh` function. The operations in the `beh` function will be translated into corresponding statements which interact with the simulated processor state. The accesses to the registers, memory and instruction fields are done through the access macros. The `srcs` and the `dst` of an *InstrnBeh* member identify themselves as register fields or immediate fields. The use of register fields will lead to accessing values through register access macros with the argument obtained from instruction access macros, while the use of immediate fields will lead to accessing values through instruction access macros. On the other hand, the load/store operations will done through memory access macros.

  If register access or memory access is involved in the behavior, statements are also generated to verify that the register and the memory are used properly, and the binary code produced from the compiler and assembler doesn't break the alignment restriction. It is the responsibility of the processor compiler to test the correctness of the compilation tools. The alignment requirement of the registers and memory are respectively carried in the `uses` of the corresponding *RegFile* member and the `memUses` of the *abi* member.

```
enum md_fu_class {
  FUClass_NA = 0,    /* inst does not use a functional unit */
  IntALU,            /* integer ALU */
  IntMULT,           /* integer multiplier */
  IntDIV,            /* integer divider */
  FloatADD,          /* floating point adder/subtractor */
  FloatCMP,          /* floating point comparator */
  FloatCVT,          /* floating point<->integer converter */
  FloatMULT,         /* floating point multiplier */
  FloatDIV,          /* floating point divider */
  FloatSQRT,         /* floating point square root */
  RdPort,            /* memory read port */
  WrPort,            /* memory write port */
  NUM_FU_CLASSES     /* total functional unit classes */
};
```

Figure 5.14: The Functional Unit Class in SimpleScalar

```
#define F_ICOMP     0x00000001  /* integer computation */
#define F_FCOMP     0x00000002  /* FP computation */
#define F_CTRL      0x00000004  /* control inst */
#define F_UNCOND    0x00000008  /* unconditional change */
#define F_COND      0x00000010  /* conditional change */
#define F_MEM       0x00000020  /* memory access inst */
#define F_LOAD      0x00000040  /* load inst */
#define F_STORE     0x00000080  /* store inst */
#define F_DISP      0x00000100  /* displaced (R+C) addr mode */
#define F_RR        0x00000200  /* R+R addr mode */
#define F_DIRECT    0x00000400  /* direct addressing mode */
#define F_TRAP      0x00000800  /* traping inst */
#define F_LONGLAT   0x00001000  /* long latency inst (for sched) */
#define F_DIRJMP    0x00002000  /* direct jump */
#define F_INDIRJMP  0x00004000  /* indirect jump */
#define F_CALL      0x00008000  /* function call */
#define F_FPCOND    0x00010000  /* FP conditional branch */
#define F_IMM       0x00020000  /* instruction has immediate operand */
#define F_DELAYED   0x00040000  /* delayed execution */
```

Figure 5.15: The Instruction Flags

5.2.4 Retargetting Software Instruction Decoder

Software instruction decoding is done by a hierarchy of C switch statements. This is done by constructing a decoding tree from the *InstrnOpFmt* members in *isa* and the instructions associating with each of them. The C switch statements are then emitted through preorder traversal of the decoding tree.

5.2.5 Porting System Call Emulation

The system call package is the only component that fails to be retargetted from the processor abstraction. This part must be manually ported by the user.

Chapter 6

Experiments and Results

With the methodologies discussed in Chapter 4 and Chapter 5, we implemented a system that automatically ports the GNU BFD library and linker as well as the SimpeScalar toolset, by taking a processor model specification that we describe in Chapter 3. This chapter describes the implementation details of the system. In the end, we demonstrate the feasibility of our approach with the experimental results collected from the tools generated by our system.

6.1 Implementation

Figure 6.1 illustrates the complete design of the implemented system. The processor specification is written in *Babel* [**?**], a general-purpose language that can be used as architectural description language (ADL). The *Babel language compiler* compiles the input specification into a form of internal abstract data, which can be easily accessed through an API. By accessing the architectural data captured by the internal data, the *BFD & linker generator* and the *SimpleScalar generator* generate the processor-dependent portion of the target tools. In the system, the retargetable BFD and linker part can be standalone. However, the retargetable SimpleScalar tool depends on the ported GNU BFD library generated from the retargetable BFD and linker tool. The detailed description of the processor model and both generators are as follows:

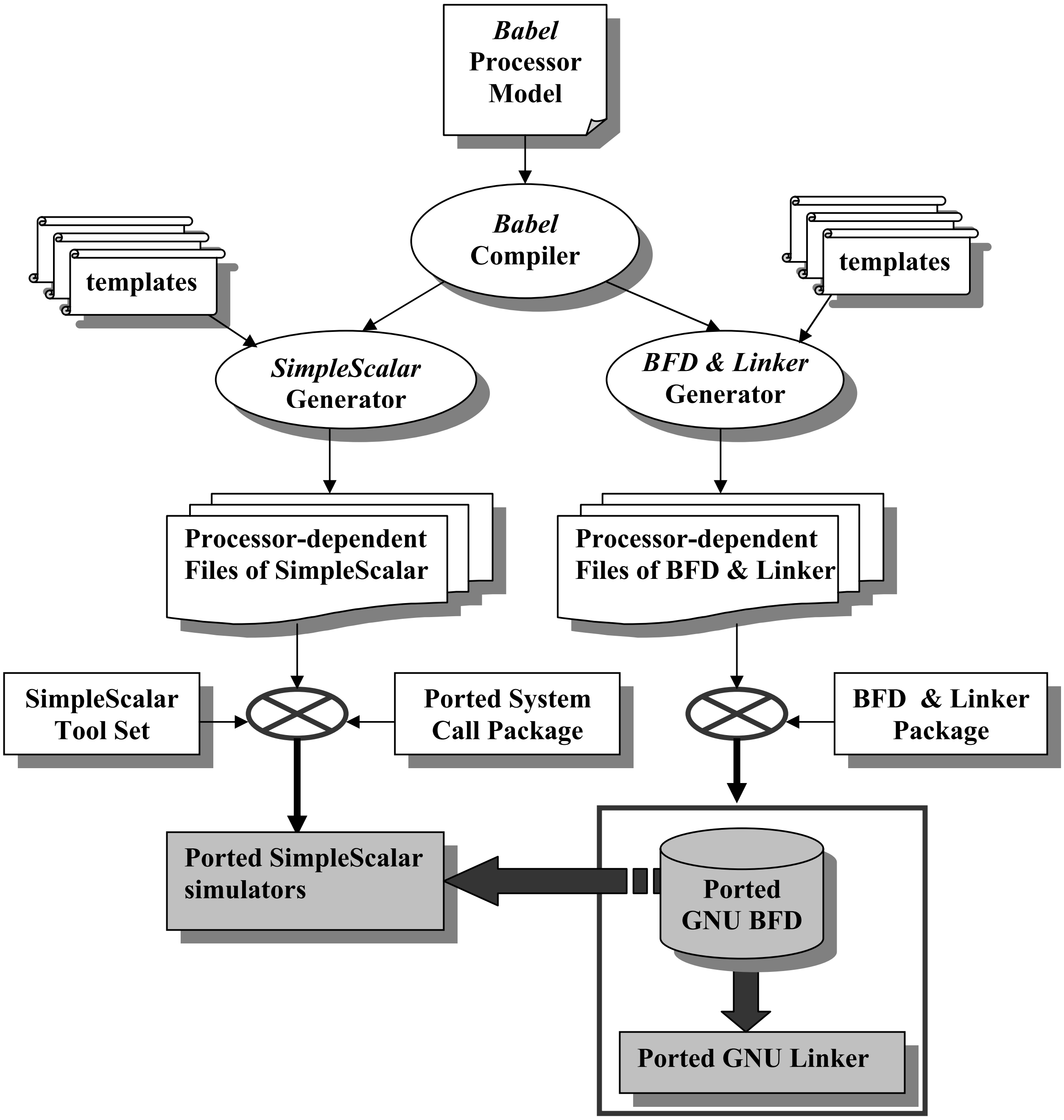


Figure 6.1: The System Overview of the Implementation

6.1.1 *Babel* Processor Model

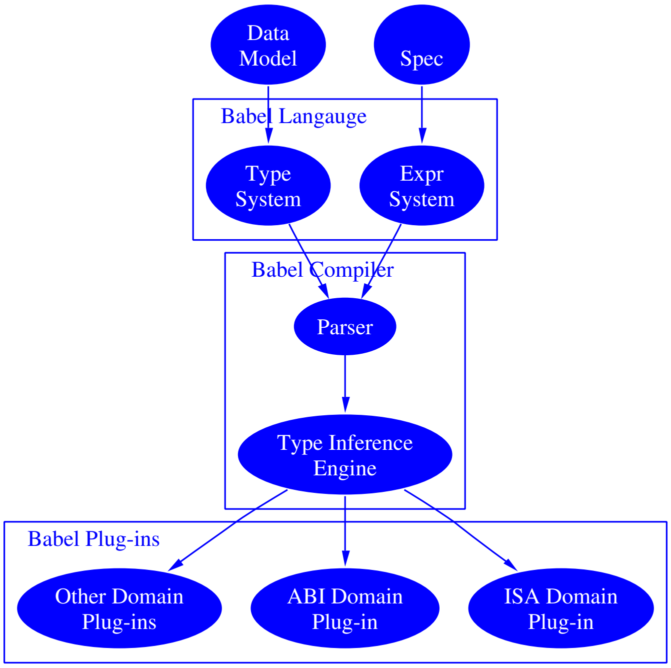


Figure 6.2: Babel Language Architecture

The processor specification is written in *Babel*. Babel is *general-purpose* because it allows user to define the semantics by providing a data model of arbitrary complexity. A data model describes an abstracted view of a design component, such as ISA and ABI of a processor architecture. The data that abstracts a view is called a *domain*. Babel is an *extensible* language because ability of capturing a new abstracted view can be added without changing the language syntax.

Figure 6.2 shows the architecture of the Babel language system. The user defines the data model by using the type system of Babel. Also, Babel provides an expression system for the specification of the actual data. The Babel compiler is designed in such a way that both of the data model and the specification are translated into an intermediate representation, after which the soundness of the specification data is checked against the data model through a type inference engine. The intermediate representation is compiled and captured by the corresponding domain plug-ins, each of the which captures data of an abstracted view.

To extend Babel with a new domain, one only has to add a header file containing the data model and corresponding domain plug-in. To define our processor abstraction model in Babel, we built the ISA domain and the ABI domain to define the *ISA* (Definition 2) and the *ABI* (Definition 14) respectively. The header file (`arch.bbh`) containing the data model of both domains can be found in Appendix A.1. In `arch.bbh`, the `class domain.ISA` and the `class domain.ABI` are the data model of the ISA domain and the ABI domain respectively. The type system of Babel is very closed to the FAN notation, so mapping the processor model in Chapter 3 to Babel is straightforward. On the other hand, the domain plug-ins for ISA and ABI are also constructed to capture the processor specification.

The processor specification in Babel is distributed into three files. To abstract one view of the processor architecture, each file carries specification of one domain. The three views are *behavior*(BEH), *ISA* and *ABI*. In addition to the *ISA* and the *ABI* domains that we constructed, the *behavior* domain is pre-constructed and it helps capturing computation-centric data such as instruction semantics. As a result, the following three files are needed to construct a Babel specification for a processor architecture *myarch*:

- *`myarch`*`.bbl`, which captures the behavior view of the architecture. It consists of the data types of registers and complex behavioral semantics definitions on which some instructions in the *myarch*.isa.bbl file will depend.

- *`myarch`*`.isa.bbl`, which captures the ISA view of the architecture. It includes the register file organization and the instruction set definition.

- *`myarch`*`.abi.bbl`, which captures the ABI view of the architecture. The current ABI model gives information about the calling convention in terms of the register usage; and on the other hand the information of linking such as relocation and dynamic linking rules.

The processor specification for SPARC and Intel386(i386) written in Babel can be found in the Appendix: the behavior domain files are given in Appendix A.2, the ISA domain files are given in Appendix A.3 and the ABI domain files are given in Appendix A.4. To avoid confusion between the tools generated by our system and those already provided by the target toolset, we rename the processor names as *rsparc* and *ri386* respectively. The *rsparc* specification is complete such that it can be used to generate ported BFD library, linker and SimpleScalar toolset. However, the *ri386* specification contains data to generate only ported BFD library and linker. This is because that the i386 is too complex that it beyond our goal to retarget tools for embedded processors. The reason that we choose i386 in this thesis is to study the feasibility of our approach.

6.1.2 The GNU BFD & Linker Generator

The architectural data in the domains (BEH, ISA and ABI) can be accessed through the domain APIs. The GNU BFD & linker generator will load the necessary data from the domains to its own representation, which is defined in a form that facilitates the data manipulation inside the generator. The generator then produces the processor-dependent portion of the code from the architectural data with the assistance of a set of template files, which are processor-independent. The template files include not only partial implementation, but also the placeholders which can instruct the generator to emit codes at appropriate locations.

The output of the generator is a set of processor-dependent files, which include C headers and source files as well as the appropriate configuration scripts. When added to the proper place in the GNU BFD and linker package, support to the target processor architecture will be added to the package. We also provide a script to automate this process. The GNU BFD library and the GNU linker is only two of the tools in the GNU Binutils. We reduced the GNU Binutils (version 2.13) into a package which includes only the BFD library and the linker.

6.1.3 The SimpleScalar Generator

The SimpleScalar generator takes exactly the same approach as the GNU BFD & linker generator. The SimpleScalar generator loads the necessary data to its own representation, that facilitates the generation of the SimpleScalar processor-dependent files. The architectural data required by the two generators are different. For example, the SimpleScalar generator requires data of the instruction sets, but it is not the case for the BFD & linker generator. Incomplete processor specification is allowed (e.g. the ri386 specification). When loading data from domains, the generator will verify that all the necessary data is available.

The output of the SimpleScalar generator is also a set of processor-dependent files, which include both the architectural definition files and the modified simulator-engines. Our tool so far retargets three simulator engines: *sim-safe*, *sim-cache* and *sim-bpred*. Adding the generated files to the SimpleScalar toolset is also automated by script. It is however not enough for porting SimpleScalar. Besides the generated files, ported system call package must be provided by the user and ported BFD library must be available for the ported program loading function. Our current SimpleScalar generator supports SimpleScalar toolset 3.0.

6.2 Experiments

To verify the feasibility of our approach, we tested our implemented system with a subset of SPEC2000 testbenches, including 3 integer benchmarks (181.mcf, 197.parser and 164.gzip) and 3 floating-point benchmarks (183.equake, 188.ammp and 179.art). All experiments are performed on a 750Mhz SunBlade 1000 workstation, except the testing of ri386 GNU linker.

The retargetable BFD & linker system is tested with the SPARC and i386 architectures, while the retargetable SimpleScalar system is tested with the SPARC and PISA architectures. The system is fully tested with the SPARC architecture, while the other two architectures are used to test the partial system for convenience. To avoid confusion, we rename them as *rSPARC*, *ri386* and *rPISA* respectively in the experiments. The discussion of the experimental result is follows.

6.2.1 Testing the Retargetable BFD & Linker System

We tested the retargetable BFD & linker system with rSPARC and ri386. The Table 6.1 shows the processor-dependent files generated from the BFD & linker generator (rSPARC and ri386) and compare them with the ones provided by the current Binutils package (SPARC and i386). In the table, *myarch* represents the architecture name (e.g. rsparc). The `bfd/elf32-`*myarch* is the most important processor-dependent file generated from the system, as the other are either configuration scripts or files containing a few parameter files. From the table, we can see that the `bfd/elf32-`*myarch* file generated from the system is shorter. It is because that the generic algorithms cannot support any architecture-dependent optimization. Also, the generated file only has basic linking functionality while the manually-crafted file may contain special feature support.

File	# lines			
	SPARC	rSPARC	i386	ri386
`config.sub`	1443	1451	1443	1451
`include/elf/common.h`	672	677	672	677
`include/elf/`*myarch*`.h`	152	68	67	67
`bfd/Makefile.am`	1566	1580	1566	1580
`bfd/archures.c`	1087	1096	1087	1096
`bfd/config.bfd`	1181	1192	1181	1192
`bfd/configure`	7150	7156	7150	7156
`bfd/configure.in`	866	871	866	871
`bfd/cpu-`*myarch*`.c`	169	46	101	46
`bfd/elf32-`*myarch*`.c`	2151	1883	3153	1908
`bfd/reloc.c`	3774	3807	3774	3795
`bfd/targets.c`	1280	1287	1280	1287
`ld/Makefile.am`	1393	1401	1393	1401
`ld/configure.tgt`	503	520	503	520
`ld/emulparams/elf32_`*myarch*`.sh`	13	17	12	16

Table 6.1: Manually-made vs. Generated Files - GNU BFD & Linker

To verify the automatically generated BFD library and linker, we used the generated GNU linker to link object files of the testing benchmarks, and run the output executable. We tried both static linking and dynamic linking. We found that the automatically generated GNU linkers for rSPARC and ri386, which uses the corresponding automatically generated BFD libraries, function the same as the manually-crafted ones provided by the GNU Binutils.

6.2.2 Testing the Retargetable SimpleScalar System

We tested the retargetable SimpleScalar system with rSPARC and rPISA. The Table 6.2 shows the processor-dependent files generated from the SimpleScalar generator (rPISA and rSPARC) and compare them with the original ones provided by SimpleScalar (PISA). In the table, *myarch* represents the architecture name (e.g. rsparc).

File	# lines		
	PISA	rPISA	rSPARC
target-*myarch*/loader.c	615	283	284
target-*myarch*/arch.h	737	505	473
target-*myarch*/arch.c	671	1014	1624
target-*myarch*/arch.def	2067	1863	3059
sim-safe.c	307	279	449
sim-cache.c	782	756	910
sim-bpred.c	513	490	658

Table 6.2: Manually-made vs. Generated Files - SimpleScalar

In addition to the processor-dependent files generated from the SimpleScalar generator, we need the ported BFD library and the system call package to make the toolset completely ported. For rSPARC, we use the BFD library generated from the retargetable BFD & linker system. The system call package for rSPARC is manually ported. For rPISA, we use the BFD library provided from the SimpleScalar compilation toolset. The system call package for PISA in the original system can be used by rPISA also. In our experiments, the binary file format of rSPARC is ELF while that of rPISA and PISA is COFF.

We compare the performance of the automatically-ported simulators with the manually-crafted simulators provided from SimpleScalar. Also, we present some simulation data collected from running the generated simulators. Our first experiment demonstrates that our tool can successfully perform instruction set simulation. Figure 6.3 shows the simulation performance in terms of simulated instructions per second for the *rPISA* and *rSPARC*, compared against *PISA*. Table 6.3 shows the number of instructions executed for each benchmark in terms of different ISA. It can be observed that the performance of our generated instruction simulator is comparable to the original one provided by the SimpleScalar simulator. The performance lag of the automatically generated simulator can be explained by the fact that the PISA simulator pre-decode all the instructions at the loading time and store the decoded value at a extra field space in the instruction. However, the automatically generated rPISA simulator cannot take this architecture-dependent advantage, and has to do instruction decoding at each simulation iteration.

We then demonstrate the performance of our retargeted simulator for detailed micro-architecture simulation: Figure 6.4 for cache simulation and Figure 6.5 for branch prediction simulation. Again, the performance of our generated simulators is comparable to the original ones. Figure 6.6 is shown to demonstrate the different simulation speed for different level of architectural detail for the same processor.

Benchmark	# instructions executed	
	rPISA	rSPARC
181.mcf	419,091,512	497,727,974
197.parser	46,080,766,461	61,881,001,366
164.gzip	147,714,434,639	204,889,958,252
183.equake	3,277,913,476	2,927,744,817
188.ammp	25,016,922,286	16,563,751,978
179.art	28,986,818,426	36,118,444,032

Table 6.3: Number of instructions executed - SimpleScalar

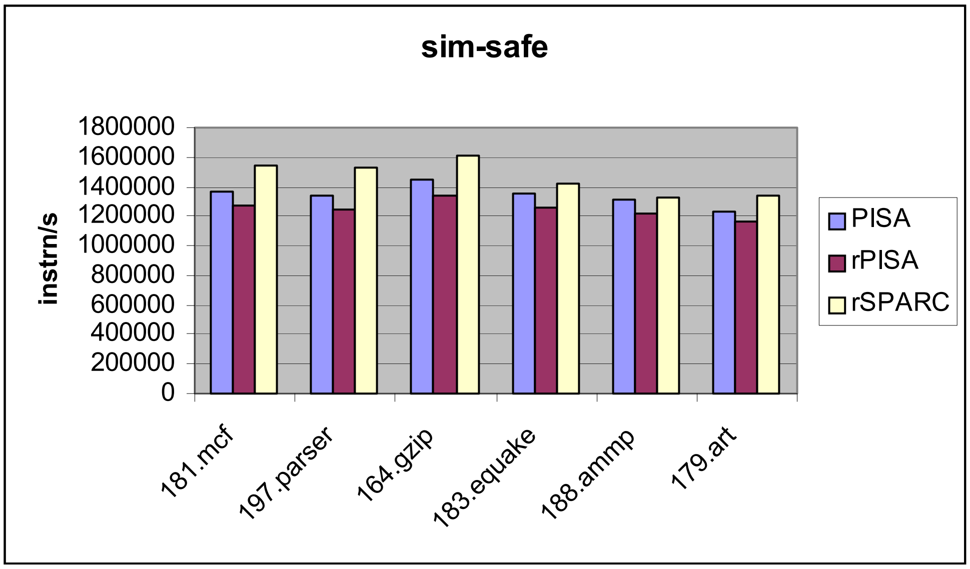


Figure 6.3: Performance of *sim-safe*

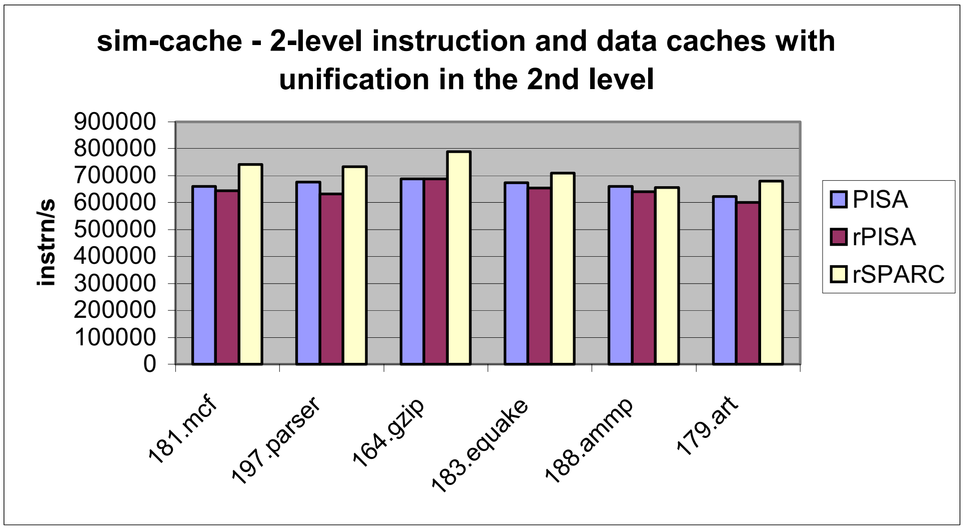


Figure 6.4: Performance of *sim-cache*

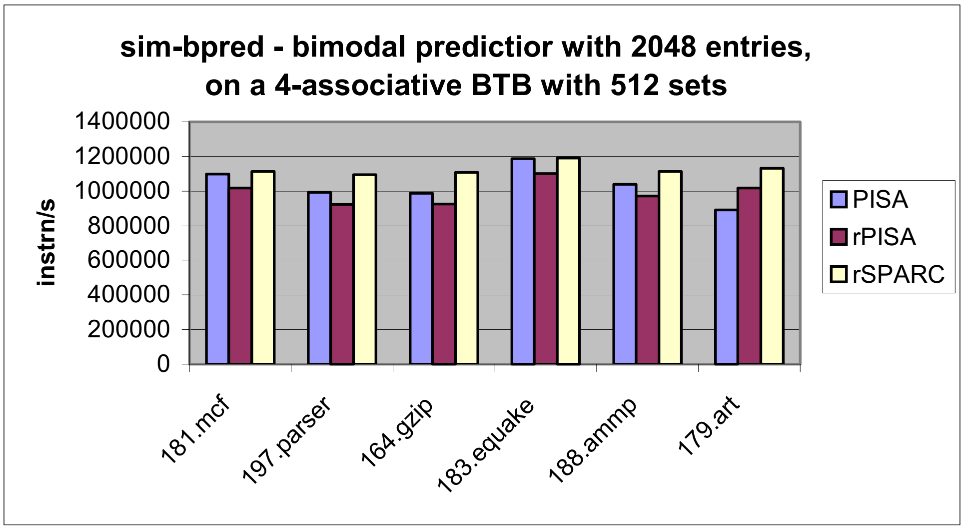


Figure 6.5: Performance of *sim-bpred*

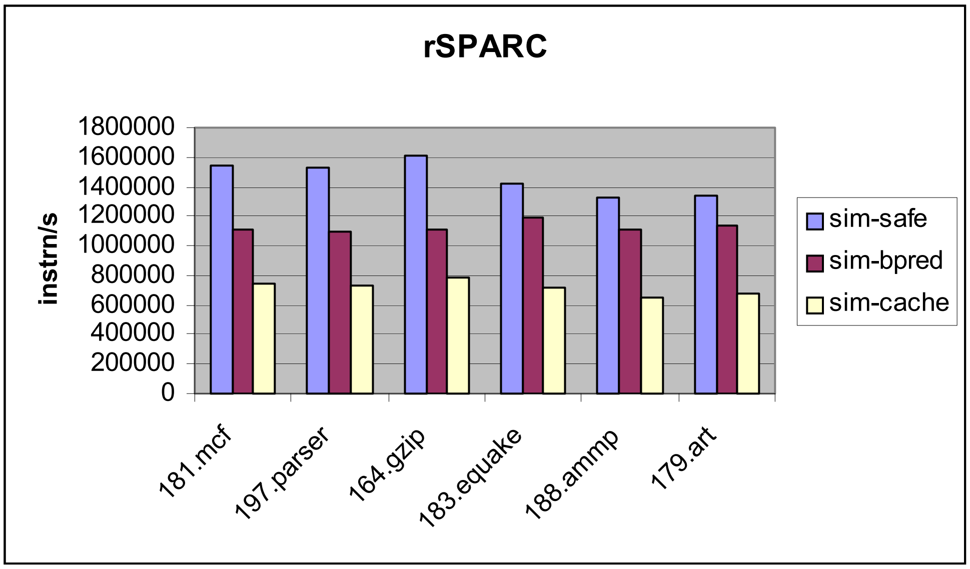


Figure 6.6: Performance of rSPARC

Chapter 7

Conclusion and Future Work

In this thesis, we have argued the importance of the role that embedded processors play in system-on-chips of today and the future. An important characteristics of embedded processors is the need to adapt the architecture to the application. Often times system architects have to iterate the architectural exploration process to find the best trade-off. An enabling technology to make rapid architecture exploration possible is the automatic generation of architecture-dependent embedded software development tools. This thesis addresses two of such important tools, namely linker and micro-architecture simulator.

We draw several conclusions from our experience in the study presented in this thesis.

First, our decision to conform to *de facto* standards of embedded software development tools is *expensive*, but it leads to a *practical* solutions. In fact, the most time-consuming and tedious component of this work is the understanding of the GNU code base. Perhaps due to the continuously evolving nature of free software contributed by decentralized programmers all over the world, the architecture-dependent interface is poorly documented. Furthermore, the implementations of this obscure interface is *ad hoc*: it is not uncommon to find that seemingly different implementations are performing the same job. Our contribution of identifying and distilling a minimum architecture-dependent API from an existing but proven code base and finding an implementation as generic as possible might not appear as elegant as the option of starting completely from scratch, but it is of practical importance: Developing from scratch a modern linker supporting modern programming languages, such as C++, and modern operating systems, such as those with dynamic linking, will be prohibitively high. Developing from scratch a modern micro-architecture simulator supporting cache organization, branch prediction and out-of-order issues is equally unnecessary.

Second, our decision to focus on architectural *modeling* proves beneficial. The architecture descriptions of many previous work are tied to specific tools, for example, the code generator's generator. While intellectually involving, architectural modeling forces us to concentrate on the semantics, rather than the utility of architectural description. Therefore, our contribution on architectural model is not necessarily tied to our implementation. More importantly, different retargetable tools, such as compilers, assemblers, linkers and simulators, can derive their needed information from a single source.

Third, our decision to adopt Babel, a general-purpose *language* where data model can be captured as a first-class citizen using its type system proves beneficial. In fact, as our research proceeds, our architectural model has been refined and revamped and we do not expect the model presented in this thesis is carved in stone. Had our architecture model been hardwired into the ADL syntax, as all previous work do, our implementation and experiments would have taken longer time.

Fourth, our effort on abstracting the ABI information, especially the linking related information, is successful. A systematic treatment of ABI has not been reported before other than our preliminary study on this subject. Our ABI model can help driving the generation of completely different tools: in this thesis linkers and simulators, and in the future register allocators and assemblers.

Finally, our work is not without limitations. Some limitations are by design. For example, one of the most difficult decision we made in developing the instruction set model is to give up the modeling of x86-based processors. The decision is made to keep our model simple enough so that all instructions can be decoded with a tree-based algorithm. The inclusion of x86-based processor requires a finite-state based algorithm, a complication we choose to avoid with the assumption that an x86-based processor is unlikely used in an embedded environment, or its development tool is unlikely unavailable.

Some limitations extend naturally to future work. We have limited our ABI model to linking related information. Other important information, such as calling convention, which is important for retargetable compiler, is not yet mature enough to be included in this thesis. Other tools contained in the binary utilities, among the most important the GNU assembler, are not included in this study. However, given our group's experience in a previous study, this retargetting task is feasible since the majority of their architecture-dependent code is within the BFD library, a task already addressed by this thesis. Another important tool that could benefit from the retargetting of BFD library is the GNU debugger.

After this study, we are convinced that the paradigm of retargetable development tools are feasible. However, it is not realistic to expect all porting can be performed automatically. One such example is the C library, which contains a minimal amount of architecture-dependent code, typically written in assembly. Another such example is the embedded operating system. We envision that architectural exploration should start with a base processor equipped with a generic, base instruction set. While new instructions can be added to the processor to accommodate applications, the library code and the OS code can be based on the base instruction set. The cost of porting C library and OS can therefore be amortized in this fashion.

Bibliography

[1] M. Abbaspour and J. Zhu. Retargetable binary utilities. In *Proceeding of the 39th Design Automation Conference*, June 2001.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342, University of Wisconsin-Madison, Computer Science, June 1997.

[3] S. Chamberlain. *libbfd : the Binary File Descriptor Library*. Cygnus Support, Free Software Foundation, Inc., April 1991.

[4] S. Chamberlain. *Using ld : the GNU Linker*. Cygnus Support, Free Software Foundation, Inc., January 1994.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceeding of the 34th Design Automation Conference*, June 1997.

[6] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. Expression : A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the Design Automation and Test Conference in Europe*, March 1999.

[7] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1338–1354, November 2001.

[8] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proceedings of Asian-Pacific Design Automation Conference*, Hong Kong, January 1999.

[9] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[10] R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems - Tools and Applications*. Kluwer Academic Publishers, 2001.

[11] J. R. Levine. *Linkers and Loaders*. Morgan Kufmann Publishers, 2000.

[12] W. S. Mong and J. Zhu. Retargetable binary utilities - a hacker's guide. Technical Report TR-03-05-01, University of Toronto, Electrical and Computer Engineering, May 2003.

[13] R. A. Mueller, M. R. Duda, P. H. Sweany, and J. S. Walicki. Horizon:a retargetable compiler for horizontal microarchitectures. *IEEE Transactions on Software Engineering*, 14(5):575–583, May 1998.

[14] A. Nohl, G. Braun, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceeding of the 39th Design Automation Conference*, June 2001.

[15] R. H. Pesch and J. M. Osier. *The GNU Binary Utilities*. Cygnus Support, Free Software Foundation, Inc., May 1993.

[16] G. Rozenberg and F. W. Vanndrager. *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes on Computer Science*. Springer, 1998.

[17] *SimpleScalar LLC*. `http://www.simplescalar.com`.

[18] R. M. Stallman. *Using the GNU Compiler Collection*. Free Software Foundation, Inc., January 2002.

[19] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, May 1995.

[20] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Design Automation and Test Conference in Europe*, Munich, Germany, March 1999.

Appendix A

Sample Babel Processor Description

A.1 Architecture Model in Babel - `arch.bbh`

```
#ifndef ARCH_BBH
#define ARCH_BBH

typedef int ^ int           Range;
typedef field               Cell;
typedef []field             CellGroup;

enum DataTypeKind {
    KIND_DATATYPE_INT       = 0,
    KIND_DATATYPE_UNSIGNED  = 1,
    KIND_DATATYPE_FLOAT     = 2,
    KIND_DATATYPE_ADDR      = 3
    }

/* DataTypeKind ^ bit size ^ align(byte / reg number) */
typedef int ^ int ^ int     Usage;


enum CtrlRegKind {
    KIND_CTRLREG_NONE       = 0,
    KIND_CTRLREG_PC         = 1, // program counter
    KIND_CTRLREG_NPC        = 2, // next program counter
    KIND_CTRLREG_WINPTR     = 3, // current window pointer
    KIND_CTRLREG_WININVALID = 4  // window invalid check
    }

typedef Range ^ CtrlReg         CtrlRegField;
class CtrlReg {
    CtrlReg( int kind, Cell map );

    {}CtrlRegField  fields;
    }


typedef Cell ^ method   RegSetup;

class RegFile {
    RegFile( int gran, int size );
```

```
    /* window regs info */
    Range       pos;
    int         depth;
    int         overlap;
    RegSetup    ptr;                // window pointer initial setup
    RegSetup    invalid;            // invalid check initial setup
    boolean     saveDir;            // true -  move down, false - move up
    method      overflowCond;       // overflow condition
    method      underflowCond;      // underflow condition
    method      overflowUpdate;     // invalid register update
    method      underflowUpdate;    // invalid register update

    {}Usage     uses;

    CellGroup   maps;
    }


class FieldAccessor {
    FieldAccessor( string name, Range pos );
    }

typedef []FieldAccessor         opFormat;


class Field {
    /* register field */
    Field( FieldAccessor p, RegFile rfile );

    /* immediate field */
    Field( FieldAccessor p, boolean isSigned );
    }

enum InstrnKind {
    KIND_INSTRN_NONE            = 0,
    KIND_INSTRN_WINSAVE         = 1,
    KIND_INSTRN_WINRESTORE      = 2,
    KIND_INSTRN_DELAYED         = 3,
    KIND_INSTRN_TRAP            = 4,
    KIND_INSTRN_CALL            = 5,
    KIND_INSTRN_RETURN          = 6,
    KIND_INSTRN_INDIR           = 7
    }

class Instrn {
    string          asmFormat;
    opFormat        opfmt;
    []int           opcodes;
    Field           dst;
    []Field         srcs;
    InstrnKind      kind;
    method          beh;
    RegFile         win;
    }

/* assume the max field length is 32 bits */
typedef Field ^ int     FieldExpr;

class InstrnRule {
    InstrnRule( Instrn i );
    {}FieldExpr fieldValues;
    }

class InstrnConv {
    InstrnConv( int kind );
    {}InstrnRule rules;
    }
```

```
class domain.ISA {
    string          cpu;
    string          manufacturer;
    int             wordSize;
    int             addrSize;
    int             instrnSize;
    int             maxDataAlign;
    boolean         bigEndian;

    {}RegFile       rfiles;
    {}CtrlReg       ctrls;

    {}FieldAccessor accessors;
    {}opFormat      opFmts;
    {}Field         fields;
    {}Instrn        instrns;
    {}InstrnConv    convs;
    }

#define dynSectAddr         (__dyn_addr)
#define pltSectAddr         (__plt_addr)
#define gotSectAddr         (__got_addr)
#define relocValue          (__reloc_val)
#define gotRelocValue       (__got_reloc_val)
#define pltEntryOffset      (__plt_e_off_pltfill)
#define pltgotEntryOffset   (__got_e_off_pltfill)
#define jsrelEntryOffset    (__jsrel_e_off_pltfill)


#define DEFINE_RESERVED_SYMBS   \
    pointer __dyn_addr, __plt_addr, __got_addr, __reloc_val, __got_reloc_val; \
    pointer __plt_e_off_pltfill, __got_e_off_pltfill, __jsrel_e_off_pltfill;

#define SYMB_EXPR( symb )   \
    pointer symb##Expr() {  \
        return symb;        \
    }

#define DEFINE_BASIC_EXPRS  \
    SYMB_EXPR( dynSectAddr ) \
    SYMB_EXPR( pltSectAddr ) \
    SYMB_EXPR( gotSectAddr ) \
    SYMB_EXPR( relocValue ) \
    SYMB_EXPR( gotRelocValue ) \
    SYMB_EXPR( pltEntryOffset ) \
    SYMB_EXPR( pltgotEntryOffset ) \
    SYMB_EXPR( jsrelEntryOffset )


enum RelocKind {
     KIND_RELOC_NONE        = 0,
     KIND_RELOC_DATA        = 1,
     KIND_RELOC_FUNC        = 2,
     KIND_RELOC_GOT         = 3,
     KIND_RELOC_PLT         = 4,
     KIND_RELOC_COPY        = 5,
     KIND_RELOC_GLOBDAT     = 6,
     KIND_RELOC_JMPSLOT     = 7,
     KIND_RELOC_RELATIVE    = 8
     }

/* field ^ check_overflow? */
typedef string ^ boolean        RelocField;

class Reloc {
     Reloc( string name, int kind, int value );
```

```
     /* info. about calculating the relocated value */
     boolean    pcRel;
     boolean    gotRel;
     boolean    addend;         /* + addend? */
     method     operation;      /* applied on the symb. value */
     int        extSize;        /* extracted bit size */
     RelocField rfield;
     boolean    aligned;        /* restrict word alignment? */
     }

class Got {
     string     accessSymb;     /* access symb. if any */
     int        maxSize;        /* -1 if none */
     []method   startEntries;
     []method   endEntries;
     }

typedef Range ^ method      Fill;

class PltEntry {
     PltEntry( int size, []ubyte tmpl );

     {}Fill fills;
     }

class PltLayout {
     PltLayout( PltEntry normalEntry );

     PltEntry   startEntry;
     PltEntry   endEntry;
     }


class Plt {
     Plt( boolean inTextSeg );  /* resides in text seg. -> depends on GOT */

     string     accessSymb;     /* access symb. if any */
     int        maxSize;        /* -1 if none */
     int        offInGot;       /* offset value of PLT entry holding in the
                                   corresponding GOT entry */
     []PltLayout layouts;
     }


enum DynKind {
     KIND_DYN_PLTGOT        = 0
     }

/* entry in the .dynamic section
   may have more later */
class Dyn {
     Dyn( int kind );

     method val;
     }

class Stack {
    Stack( unsigned baseAddr, Cell stackPtr, int align, int maxEnviron );
    int         saveArea;
    }


class domain.ABI {
     int            e_mach;     // ELF machine number
     string         procEMachName;
     int            maxPageSize;
     int            pageSize;
     unsigned       startAddr;
```

```
    string          dynLinkerPath;
    boolean         reloca;
    string          procRelocName;

    {}string        localSymPrefixes;
    {}Reloc         relocs;
    {}Dyn           dyns;
    Got             got;

    {}PltEntries    pltEntries;
    {}PltLayout     pltLayouts;
    Plt             plt;

    CellGroup       zero;

    Stack           stack;
    {}Usage         memUses;
    }

#endif
```

A.2 Behavior Domain

A.2.1 rsparc.bbl

```
#include "arch.bbh"

typedef unsigned[32]        pointer;

facet rsparc::beh {
    DEFINE_RESERVED_SYMBS
    DEFINE_BASIC_EXPRS


    unsigned[32] g0, g1, g2, g3, g4, g5, g6, g7;
    unsigned[32] l0, l1, l2, l3, l4, l5, l6, l7;
    unsigned[32] i0, i1, i2, i3, i4, i5, fp, i7;
    unsigned[32] o0, o1, o2, o3, o4, o5, sp, o7;

    float[32] f0, f1, f2, f3, f4, f5, f6, f7;
    float[32] f8, f9, f10, f11, f12, f13, f14, f15;
    float[32] f16, f17, f18, f19, f20, f21, f22, f23;
    float[32] f24, f25, f26, f27, f28, f29, f30, f31;

    unsigned[32] psr;
    bits[5] cwp;
    bits[1] icc_c, icc_v, icc_z, icc_n;

    unsigned[32] wim, tbr, y, fsr;
    bits[2] fcc;

    unsigned[32] pc, npc;


    pointer reloc_oper1() {
        return (relocValue >> 2);
        }

    pointer reloc_oper2() {
        return (relocValue >> 10);
        }

    pointer plt_fill0() {
        return (uint)( (-(4 + pltEntryOffset)) >> 2 );
        }

    /* register window behavior */
    bits[5] win_ptr_init() {
        return 31;
        }

    unsigned win_invalid_init() {
        return 0x00000001;
        }

    boolean win_overflow_cond() {
        return (((wim >> cwp) & 1)!=0);
        }

    unsigned win_overflow_update() {
        return ((wim == 0x00000001)?0x80000000:(wim >> 1));
        }

    unsigned win_underflow_update() {
        return ((wim == 0x80000000)?0x00000001:(wim << 1));
```

```
    }

/* instruction semantics */
void call(int disp) {
    o7 = pc;
    pc = npc;
    npc = disp << 2;
    }

pointer jmpl(int src1, int src2) {
    pointer tmp;
    pc = npc;
    npc = src1+src2;
    return tmp;
    }

void bne(int disp) {
    pc = npc;
    if(!icc_z)
        npc = pc + (disp << 2);
    else
        npc = npc + 4;
    }

void bne_a(int disp) {
    pc = npc;
    if(!icc_z)
        npc = pc + (disp << 2);
    else {
        pc = npc + 4;
        npc = npc + 8;
        }
    }

uint orcc(uint src1, uint src2) {
    icc_n = ((src1 | src2) >> 31) & 0x1;
    icc_z = ((src1 | src2) == 0);
    icc_v = 0;
    icc_c = 0;
    return (src1 | src2);
    }
}
```

A.2.2 ri386.bbl

```
#include "arch.bbh"

typedef unsigned[32]        pointer;

facet ri386::beh {
    DEFINE_RESERVED_SYMBS
    DEFINE_BASIC_EXPRS

    pointer zeroVal() {
        return (uint)0;
        }

    pointer plt_fill0() {
        return (uint)( 4 + gotSectAddr );
        }

    pointer plt_fill1() {
        return (uint)( 8 + gotSectAddr );
        }

    pointer plt_fill2() {
        return (pltgotEntryOffset + gotSectAddr);
        }

    pointer plt_fill3() {
        return (uint)(-(16+pltEntryOffset));
        }
    }
```

A.3 ISA Domain

A.3.1 rsparc.isa.bbl

```
#include "arch.bbh"

facet rsparc::isa = new domain.ISA {
    cpu = "rsparc";
    manufacturer = "sun";
    wordSize = 32;
    addrSize = 32;
    instrnSize = 32;
    maxDataAlign = 8;
    bigEndian = true;

    rfiles = {
        gpr = new RegFile( 32, 32 ) {
            pos = <8,31>;
            depth = 32;
            overlap = 8;
            ptr = <cwp, rsparc.win_ptr_init(void)>;
            invalid = <wim, rsparc.win_invalid_init(void)>;
            saveDir = false;        // move up one window to save
            overflowCond = rsparc.win_overflow_cond(void);
            underflowCond = rsparc.win_overflow_cond(void);
            overflowUpdate = rsparc.win_overflow_update(void);
            underflowUpdate = rsparc.win_underflow_update(void);
            uses = {
                <KIND_DATATYPE_UNSIGNED, 8, 1>,
                <KIND_DATATYPE_INT, 8, 1>,
                <KIND_DATATYPE_UNSIGNED, 16, 1>,
                <KIND_DATATYPE_INT, 16, 1>,
                <KIND_DATATYPE_UNSIGNED, 32, 1>,
                <KIND_DATATYPE_INT, 32, 1>,
                <KIND_DATATYPE_UNSIGNED, 64, 2>,
                <KIND_DATATYPE_INT, 64, 2>,
                <KIND_DATATYPE_ADDR, 32, 1>
                }
            maps = [g0, g1, g2, g3, g4, g5, g6, g7,
                    l0, l1, l2, l3, l4, l5, l6, l7,
                    i0, i1, i2, i3, i4, i5, fp, i7,
                    o0, o1, o2, o3, o4, o5, sp, o7];
            }
        fpr = new RegFile( 32, 32 ) {
            uses = {
                <KIND_DATATYPE_FLOAT, 32, 1>,
                <KIND_DATATYPE_FLOAT, 64, 2>
                }
            maps = [f0, f1, f2, f3, f4, f5, f6, f7,
                    f8, f9, f10, f11, f12, f13, f14, f15,
                    f16, f17, f18, f19, f20, f21, f22, f23,
                    f24, f25, f26, f27, f28, f29, f30, f31];
            }
        }

    ctrls = {
        cPSR = new CtrlReg(KIND_CTRLREG_NONE, psr) {
            fields = {
                < <4, 0>, new CtrlReg(KIND_CTRLREG_WINPTR, cwp) >,
                < <20, 20>, new CtrlReg(KIND_CTRLREG_NONE, icc_c) >,
                < <21, 21>, new CtrlReg(KIND_CTRLREG_NONE, icc_v) >,
                < <22, 22>, new CtrlReg(KIND_CTRLREG_NONE, icc_z) >,
                < <23, 23>, new CtrlReg(KIND_CTRLREG_NONE, icc_n) >
                }
```

```
        }
    cWIM = new CtrlReg(KIND_CTRLREG_WININVALID, wim);
    cTBR = new CtrlReg(KIND_CTRLREG_NONE, tbr);
    cY   = new CtrlReg(KIND_CTRLREG_NONE, y);
    cFSR = new CtrlReg(KIND_CTRLREG_NONE, fsr) {
        fields = {
            < <11, 10>, new CtrlReg(KIND_CTRLREG_NONE, fcc)>;
            }
        }
    cPC  = new CtrlReg(KIND_CTRLREG_PC, pc);
    cNPC = new CtrlReg(KIND_CTRLREG_NPC, npc);
    }

accessors = {
    op     = new FieldAccessor("op", <31, 30>),
    disp30 = new FieldAccessor("disp30", <29, 0>),
    rd     = new FieldAccessor("rd", <29, 25>),
    op2    = new FieldAccessor("op2", <24, 22>),
    imm22  = new FieldAccessor("imm22", <21, 0>),
    a      = new FieldAccessor("a", <29, 29>),
    cond   = new FieldAccessor("cond", <28, 25>),
    disp22 = new FieldAccessor("disp22", <21, 0>),
    op3    = new FieldAccessor("op3", <24, 19>),
    rs1    = new FieldAccessor("rs1", <18, 14>),
    i      = new FieldAccessor("i", <13, 13>),
    asi    = new FieldAccessor("asi", <12, 5>),
    rs2    = new FieldAccessor("rs2", <4, 0>),
    simm13 = new FieldAccessor("simm13", <12, 0>),
    opf    = new FieldAccessor("opf", <13, 5>),
    byte8  = new FieldAccessor("byte8", <7, 0>),
    half16 = new FieldAccessor("half16", <15, 0>),
    word32  = new FieldAccessor("word32", <31, 0>)
    }

opFmts = {
    opf1A = [op],
    opf2A = [op, op2],
    opf2B = [op, a, cond, op2],
    opf3A = [op, op3, i],
    opf3C = [op, op3, opf],
    opf3D = [op, cond, op3, i]
    }

fields = {
    f_byte8    = new Field( byte8, false ),
    f_byte8_s  = new Field( byte8, true ),
    f_half16   = new Field( half16, false ),
    f_half16_s = new Field( half16, true ),
    f_word32   = new Field( word32, false ),
    f_disp32_s = new Field( word32, true ),
    f_disp30_s = new Field( disp30, true ),
    f_disp22_s = new Field( disp22, true ),
    f_imm22    = new Field( imm22, false ),
    f_simm13   = new Field( simm13, false ),
    f_simm13_s = new Field( simm13, true ),
    f_rs1_g    = new Field( rs1, gpr ),
    f_rs2_g    = new Field( rs2, gpr ),
    f_rs1_f    = new Field( rs1, fpr ),
    f_rs2_f    = new Field( rs2, fpr ),
    f_rd_g     = new Field( rd, gpr ),
    f_rd_f     = new Field( rd, fpr )
    }

instrns = {
    ldsb = new Instrn {
        asmFormat = "ldsb [%1+%2], %0";
        opfmt = opf3A;
        opcodes = [0x3, 0x9, 0x0];
```

```
    dst = f_rd_g;
    srcs = [f_rs1_g, f_rs2_g];
    beh = byte.OP_LOD( pointer, int );
    }

ldsbi = new Instrn {
    asmFormat = "ldsb [%1+#2], %0";
    opfmt = opf3A;
    opcodes = [0x3, 0x9, 0x1];
    dst = f_rd_g;
    srcs = [f_rs1_g, f_simm13_s];
    beh = byte.OP_LOD( pointer, int );
    }

ldsh = new Instrn {
    asmFormat = "ldsh [%1+%2], %0";
    opfmt = opf3A;
    opcodes = [0x3, 0xa, 0x0];
    dst = f_rd_g;
    srcs = [f_rs1_g, f_rs2_g];
    beh = short.OP_LOD( pointer, int );
    }

ldshi = new Instrn {
    asmFormat = "ldsh [%1+#2], %0";
    opfmt = opf3A;
    opcodes = [0x3, 0xa, 0x1];
    dst = f_rd_g;
    srcs = [f_rs1_g, f_simm13_s];
    beh = short.OP_LOD( pointer, int );
    }

call = new Instrn {
    asmFormat = "call #1";
    opfmt = opf1A;
    opcodes = [0x1];
    srcs = [f_disp30];
    kind = KIND_INSTRN_DELAYED;
    beh = rsparc.call(int);
    }

jmpl = new Instrn {
    asmFormat = "jmpl [%1+%2],%0";
    optfmt = opf3A;
    opcodes = [0x2, 0x38, 0x0];
    dst = f_rd_g;
    srcs = [f_rs1_g, f_rs2_g];
    kind = KIND_INSTRN_DELAYED;
    beh = rsparc.jmpl(int, int);
    }

jmpli = new Instrn {
    asmFormat = "jmpl [%1+#2],%0";
    optfmt = opf3A;
    opcodes = [0x2, 0x38, 0x1];
    dst = f_rd_g;
    srcs = [f_rs1_g, f_simm13_s];
    kind = KIND_INSTRN_DELAYED;
    beh = rsparc.jmpl(int, int);
    }

bne = new Instrn {
    asmFormat = "bne #1";
    opfmt = opf2B;
    opcodes = [0x0, 0x0, 0x9, 0x2];
    srcs = [f_disp22_s];
    kind = KIND_INSTRN_DELAYED;
    beh = rsparc.bne(int);
```

```
        }

    bne_a = new Instrn {
        asmFormat = "bne_a #1";
        opfmt = opf2B;
        opcodes = [0x0, 0x1, 0x9, 0x2];
        srcs = [f_disp22_s];
        kind = KIND_INSTRN_DELAYED;
        beh = rsparc.bne_a(int);
        }

    orcc = new Instrn {
        asmFormat = "orcc %1,%2,%0";
        opfmt = opf3A;
        opcodes = [0x2, 0x12, 0x0];
        dst = f_rd_g;
        srcs = [f_rs1_g, f_rs2_g];
        kind = KIND_INSTRN_NONE;
        beh = rsparc.orcc( uint, uint );
        }

    save = new Instrn {
        asmFormat = "save %1,%2,%0";
        opfmt = opf3A;
        opcodes = [0x2, 0x3c, 0x0];
        dst = f_rd_g;
        srcs = [f_rs1_g, f_rs2_g];
        kind = KIND_INSTRN_WINSAVE;
        win = gpr;
        beh = int.OP_ADD( int, int );
        }

    restore = new Instrn {
        asmFormat = "restore %1,%2,%0";
        opfmt = opf3A;
        opcodes = [0x2, 0x3d, 0x0];
        dst = f_rd_g;
        srcs = [f_rs1_g, f_rs2_g];
        kind = KIND_INSTRN_WINRESTORE;
        win = gpr;
        beh = int.OP_ADD( int, int );
        }

    ta = new Instrn {
        asmFormat = "ta";
        opfmt = opf3D;
        opcodes = [0x2, 0x8, 0x3a, 0];
        kind = KIND_INSTRN_TRAP;
        }

    /*

        TOO MANY INSTRUCTIONS TO PUT HERE!

    */

    }

convs = {
    d_call = new InstrnConv( KIND_INSTRN_CALL ) {
        rules = {
            new InstrnRule( call );
            }
        }

    d_return = new InstrnConv( KIND_INSTRN_RETURN ) {
        rules = {
            new InstrnRule( jmpli ) {
```

```
                fieldValues = {<f_rd_g, 0>, <f_rs1_g, 31>, <f_simm13_s, 8>};
                }
            }
        }

    d_indir = new InstrnConv( KIND_INSTRN_INDIR ) {
        rules = {
            new InstrnRule( jmpl ) {
                fieldValues = {<f_rd_g, 15>};
                }
            new InstrnRule( jmpli ) {
                fieldValues = {<f_rd_g, 15>};
                }
            }
        }
    }

}
```

A.3.2 ri386.isa.bbl

```
#include "arch.bbh"

facet ri386::isa = new domain.ISA {
    cpu = "ri386";
    manufacturer = "pc";
    wordSize = 32;
    addrSize = 32;
    maxDataAlign = 4;
    bigEndian = false;

    accessors = {
        word32  = new FieldAccessor("word32", <31, 0>)
        }

    fields = {
        f_word32 = new Field( word32, false );
        }
    }
```

A.4 ABI Domain

A.4.1 rsparc.abi.bbl

```
#include "arch.bbh"

facet rsparc::abi = new domain.ABI {
    e_mach = 2;
    procEMachName = "EM_RSPARC";
    maxPageSize = 0x10000;
    pageSize = 4096;
    startAddr = 0x10000;
    dynLinkerPath = "/usr/lib/ld.so.1";
    reloca = true;
    procRelocName = "RSPARC";

    localSymPrefixes = { ".L", "..", "_.L_" };

    relocs = {
        r_rsparc_none = new Reloc( "R_RSPARC_NONE", KIND_RELOC_NONE, 0 ) {
            aligned = true;
            }

        r_rsparc_8 = new Reloc( "R_RSPARC_8", KIND_RELOC_DATA, 1 ) {
            addend = true;
            operation = rsparc.relocValueExpr(void);
            extSize = 8;
            rfield = <"f_byte8", true>;
            aligned = true;
            }

        r_rsparc_16 = new Reloc( "R_RSPARC_16", KIND_RELOC_DATA, 2 ) {
            addend = true;
            operation = rsparc.relocValueExpr(void);
            extSize = 16;
            rfield = <"f_half16", true>;
            aligned = true;
            }

        r_rsparc_32 = new Reloc( "R_RSPARC_32", KIND_RELOC_DATA, 3 ) {
            addend = true;
            operation = rsparc.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }

        r_rsparc_disp8 = new Reloc( "R_RSPARC_DISP8", KIND_RELOC_FUNC, 4 ) {
            pcRel = true;
            addend = true;
            operation = rsparc.relocValueExpr(void);
            extSize = 8;
            rfield = <"f_byte8_s", true>;
            aligned = true;
            }

        r_rsparc_disp16 = new Reloc( "R_RSPARC_DISP16", KIND_RELOC_FUNC, 5 ) {
            pcRel = true;
            addend = true;
            operation = rsparc.relocValueExpr(void);
            extSize = 16;
            rfield = <"f_half16_s", true>;
            aligned = true;
            }
```

```
r_rsparc_disp32 = new Reloc( "R_RSPARC_DISP32", KIND_RELOC_FUNC, 6 ) {
    pcRel = true;
    addend = true;
    operation = rsparc.relocValueExpr(void);
    extSize = 32;
    rfield = <"f_disp32_s", true>;
    aligned = true;
    }

r_rsparc_wdisp30 = new Reloc( "R_RSPARC_WDISP30", KIND_RELOC_FUNC, 7 ) {
    pcRel = true;
    addend = true;
    operation = rsparc.reloc_oper1(void);
    extSize = 30;
    rfield = <"f_disp30_s", true>;
    aligned = true;
    }

r_rsparc_wdisp22 = new Reloc( "R_RSPARC_WDISP22", KIND_RELOC_FUNC, 8 ) {
    pcRel = true;
    addend = true;
    operation = rsparc.reloc_oper1(void);
    extSize = 22;
    rfield = <"f_disp22_s", true>;
    aligned = true;
    }

r_rsparc_hi22 = new Reloc( "R_RSPARC_HI22", KIND_RELOC_DATA, 9 ) {
    addend = true;
    operation = rsparc.reloc_oper2(void);
    extSize = 22;
    rfield = <"f_imm22", false>;
    aligned = true;
    }

r_rsparc_22 = new Reloc( "R_RSPARC_22", KIND_RELOC_DATA, 10 ) {
    addend = true;
    operation = rsparc.relocValueExpr(void);
    extSize = 22;
    rfield = <"f_imm22", true>;
    aligned = true;
    }

r_rsparc_13 = new Reloc( "R_RSPARC_13", KIND_RELOC_DATA, 11 ) {
    addend = true;
    operation = rsparc.relocValueExpr(void);
    extSize = 13;
    rfield = <"f_simm13", true>;
    aligned = true;
    }

r_rsparc_lo10 = new Reloc( "R_RSPARC_LO10", KIND_RELOC_DATA, 12 ) {
    addend = true;
    operation = rsparc.relocValueExpr(void);
    extSize = 10;
    rfield = <"f_simm13", false>;
    aligned = true;
    }

r_rsparc_got10 = new Reloc( "R_RSPARC_GOT10", KIND_RELOC_GOT, 13 ) {
    operation = rsparc.relocValueExpr(void);
    extSize = 10;
    rfield = <"f_simm13", false>;
    aligned = true;
    }

r_rsparc_got13 = new Reloc( "R_RSPARC_GOT13", KIND_RELOC_GOT, 14 ) {
```

```
        operation = rsparc.relocValueExpr(void);
        extSize = 13;
        rfield = <"f_simm13_s", true>;
        aligned = true;
        }

    r_rsparc_got22 = new Reloc( "R_RSPARC_GOT22", KIND_RELOC_GOT, 15 ) {
        operation = rsparc.reloc_oper2(void);
        extSize = 22;
        rfield = <"f_imm22", false>;
        aligned = true;
        }

    r_rsparc_pc10 = new Reloc( "R_RSPARC_PC10", KIND_RELOC_DATA, 16 ) {
        pcRel = true;
        addend = true;
        operation = rsparc.relocValueExpr(void);
        extSize = 10;
        rfield = <"f_simm13", false>;
        aligned = true;
        }

    r_rsparc_pc22 = new Reloc( "R_RSPARC_PC22", KIND_RELOC_DATA, 17 ) {
        pcRel = true;
        addend = true;
        operation = rsparc.reloc_oper2(void);
        extSize = 22;
        rfield = <"f_imm22", true>;
        aligned = true;
        }

    r_rsparc_wplt30 = new Reloc( "R_RSPARC_WPLT30", KIND_RELOC_PLT, 18 ) {
        pcRel = true;
        addend = true;
        operation = rsparc.reloc_oper1(void);
        extSize = 30;
        rfield = <"f_disp30_s", true>;
        aligned = true;
        }

    r_rsparc_copy = new Reloc( "R_RSPARC_COPY", KIND_RELOC_COPY, 19 ) {
        aligned = true;
        }

    r_rsparc_globdat = new Reloc( "R_RSPARC_GLOB_DAT", KIND_RELOC_GLOBDAT, 20 ) {
        addend = true;
        operation = rsparc.relocValueExpr(void);
        extSize = 32;
        rfield = <"f_word32", false>;
        aligned = true;
        }

    r_rsparc_jmpslot = new Reloc( "R_RSPARC_JMP_SLOT", KIND_RELOC_JMPSLOT, 21 ) {
        aligned = true;
        }

    r_rsparc_relative = new Reloc( "R_RSPARC_RELATIVE", KIND_RELOC_RELATIVE, 22 ) {
        addend = true;
        operation = rsparc.relocValueExpr(void);
        extSize = 32;
        rfield = <"f_word32", false>;
        aligned = true;
        }

    r_rsparc_ua32 = new Reloc( "R_RSPARC_UA32", KIND_RELOC_DATA, 23 ) {
        addend = true;
        operation = rsparc.relocValueExpr(void);
        extSize = 32;
```

```
            rfield = <"f_word32", true>;
            aligned = false;
            }
        }

    dyns = {
        dynpltgot = new Dyn( KIND_DYN_PLTGOT ) {
            val = rsparc.pltSectAddrExpr(void);
            }
        }

    got = new Got() {
        accessSymb = "_GLOBAL_OFFSET_TABLE_";
        maxSize = 8192;
        startEntries = [ rsparc.dynSectAddrExpr(void) ];
        }

    pltEntries = {
        pe1 = new PltEntry(
                48,
                [ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ]
                );

        pe2 = new PltEntry(
                12,
                [ 0x03, 0x00, 0x00, 0x00, 0x30, 0x80, 0x00, 0x00,
                  0x01, 0x00, 0x00, 0x00 ]
                ) {
            fills = {
                < <10, 31>, rsparc.pltEntryOffsetExpr(void) >;
                < <42, 63>, rsparc.plt_fill0(void) >;
                }
            }

        pe3 = new PltEntry(
                4,
                [ 0x01, 0x00, 0x00, 0x00 ]
                )
        }

    pltLayouts = {
        pl1 = new PltLayout( pe2 ) {
            startEntry = pe1;
            endEntry = pe3;
            }
        }

    plt = new Plt( false ) {
        accessSymb = "_PROCEDURE_LINKAGE_TABLE_";

        layouts = [pl1];
        }

    zero = [g0];

    stack = new Stack( 0xffbf0000, sp, 64, 16384 ) {
        saveArea = 64;
        }

    memUses = {
        <KIND_DATATYPE_UNSIGNED, 8, 1>;
        <KIND_DATATYPE_INT, 8, 1>;
        <KIND_DATATYPE_UNSIGNED, 16, 2>;
```

```
        <KIND_DATATYPE_INT, 16, 2>;
        <KIND_DATATYPE_UNSIGNED, 32, 4>;
        <KIND_DATATYPE_INT, 32, 4>;
        <KIND_DATATYPE_UNSIGNED, 64, 8>;
        <KIND_DATATYPE_INT, 64, 8>;
        <KIND_DATATYPE_FLOAT, 32, 4>;
        <KIND_DATATYPE_FLOAT, 64, 8>;
        <KIND_DATATYPE_ADDR, 32, 4>;
        }
    }
```

A.4.2 ri386.abi.bbl

```
#include "arch.bbh"

facet ri386::abi = new domain.ABI {
    e_mach = 3;
    procEMachName = "EM_R386";
    maxPageSize = 0x1000;
    startAddr = 0x8048000;
    dynLinkerPath = "/lib/ld-linux.so.2";
    reloca = false;
    procRelocName = "R386";

    localSymPrefixes = { ".L", "..", "_.L_", ".X" };

    relocs = {
        r_r386_none = new Reloc( "R_R386_NONE", KIND_RELOC_NONE, 0 ) {
            aligned = true;
            }
        r_r386_32 = new Reloc( "R_R386_32", KIND_RELOC_DATA, 1 ) {
            addend = true;
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_pc32 = new Reloc( "R_R386_PC32", KIND_RELOC_FUNC, 2 ) {
            pcRel = true;
            addend = true;
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_got32 = new Reloc( "R_R386_GOT32", KIND_RELOC_GOT, 3 ) {
            addend = true;
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_plt32 = new Reloc( "R_R386_PLT32", KIND_RELOC_PLT, 4 ) {
            pcRel = true;
            addend = true;
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_copy = new Reloc( "R_R386_COPY", KIND_RELOC_COPY, 5 ) {
            aligned = true;
            }
        r_r386_globdat = new Reloc( "R_R386_GLOB_DAT", KIND_RELOC_GLOBDAT, 6 ) {
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_jumpalot = new Reloc( "R_R386_JUMP_SLOT", KIND_RELOC_JMPSLOT, 7 ) {
            operation = ri386.relocValueExpr(void);
            extSize = 32;
            rfield = <"f_word32", true>;
            aligned = true;
            }
        r_r386_relative = new Reloc( "R_R386_RELATIVE", KIND_RELOC_RELATIVE, 8 ) {
            addend = true;
            operation = ri386.relocValueExpr(void);
```

```
        extSize = 32;
        rfield = <"f_word32", true>;
        aligned = true;
        }
    r_r386_gotoff = new Reloc( "R_R386_GOTOFF", KIND_RELOC_DATA, 9 ) {
        gotRel = true;
        addend = true;
        operation = ri386.relocValueExpr(void);
        extSize = 32;
        rfield = <"f_word32", true>;
        aligned = true;
        }
    r_r386_gotpc = new Reloc( "R_R386_GOTPC", KIND_RELOC_DATA, 10 ) {
        pcRel = true;
        addend = true;
        operation = ri386.gotRelocValueExpr(void);
        extSize = 32;
        rfield = <"f_word32", true>;
        aligned = true;
        }
    }

dyns = {
    dynplt = new Dyn( KIND_DYN_PLTGOT ) {
        val = ri386.gotSectAddrExpr(void);
        }
    }

got = new Got() {
    accessSymb = "_GLOBAL_OFFSET_TABLE_";
    startEntries = [ ri386.dynSectAddrExpr(void),
                     ri386.zeroVal(void),
                     ri386.zeroVal(void) ];
    }

pltEntries = {
    pe1 = new PltEntry (
        16,
        [ 0xff, 0x35, 0x00, 0x00, 0x00, 0x00, 0xff, 0x25, 0x00,
          0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ]
        ) {
        fills = {
            < <16, 47>, ri386.plt_fill0(void) >;
            < <64, 95>, ri386.plt_fill1(void) >;
            }
        }

    pe2 = new PltEntry (
        16,
        [ 0xff, 0x25, 0x00, 0x00, 0x00, 0x00, 0x68, 0x00,
          0x00, 0x00, 0x00, 0xe9, 0x00, 0x00, 0x00, 0x00 ]
        ) {
        fills = {
            < <16, 47>, ri386.plt_fill2(void) >;
            < <56, 87>, ri386.jsrelEntryOffsetExpr(void) >;
            < <96, 127>, ri386.plt_fill3(void) >;
            }
        }

    pe3 = new PltEntry (
        16,
        [ 0xff, 0xb3, 0x04, 0x00, 0x00, 0x00, 0xff, 0xa3,
          0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ]
        );

    pe4 = new PltEntry (
        16,
        [ 0xff, 0xa3, 0x00, 0x00, 0x00, 0x00, 0x68, 0x00,
```

```
          0x00, 0x00, 0x00, 0xe9, 0x00, 0x00, 0x00, 0x00  ]
        ) {
        fills = {
            < <16, 47>, ri386.pltgotEntryOffsetExpr(void) >;
            < <56, 87>, ri386.jsrelEntryOffsetExpr(void) >;
            < <96, 127>, ri386.plt_fill3(void) >;
            }
        }
    }

pltLayouts = {
    pl1 = new PltLayout( pe2 ) {
        startEntry = pe1;
        }
    pl2 = new PltLayout( pe4 ) {
        startEntry = pe3;
        }
    }

plt = new Plt( true ) {
    offInGot = 6;

    layouts = [pl1, pl2];
    }

}
```