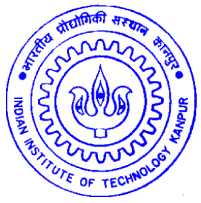
Generation Of Assemblers Using High Level Processor Models

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

by
Sarika Kumari



to the
Deapartment of Computer Science & Engineering
Indian Institute of Technology, Kanpur
Feb, 2000

Certificate

This is to certify that the work contained in the thesis entitled **"Generation Of Assemblers Using High Level Processor Models"**, by Sarika Kumari, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Date: Feb, 2000

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

Abstract

Assemblers are typically specific to the processors. Much of an assembler has to be re-implemented for each new machine. This thesis describes the design and implementation of an *assembler generator (asmg)*. The assembler generator automatically generates an assembler for a processor by taking a high level model of the processor in Sim-nML[1] intermediate form as input. The Sim-nML language provides a simple, elegant and powerful mechanism to describe the processor behavior at the instruction level.

The assembler thus generated takes an assembly language program as input and generates a relocatable binary object file in *ELF*[2] format as output. The pseudo operations supported by the generated assembler are independent of the processor and conform to the syntax and semantics of the *GNU assembler*(GAS)[3]. In addition, the generated assembler supports the link-time relocation of the program. Such relocations are typically machine specific and are handled with the help of specific configuration file which defines the machine specific types of the relocations with respect to the operations which use them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Preamble

The system designers require sophisticated and generic modeling tools for the design of high performance embedded systems. These tools help in evaluation of alternative implementations, they simplify the process of design changes and allow for the trade-offs at the early stages of development. *Hardware-software co-design* is a cost effective methodology and results in a shorter time to market as opposed to the design of software and hardware components of an embedded system separately.

Powerful modeling tools with high level of abstraction are needed with an integrated environment which allows designers to assemble, compile, simulate and analyze the performance of various alternatives of the new design. Hand-coding is not an option since it eliminates the capability of iterating over multiple hardware specifications.

In this thesis, we have designed a tool called *assembler generator* (also referred to as *asmg*) which generates an assembler for processor whose specification is available in Sim-nML[1] language. Sim-nML is simple and powerful language used to specify a complex processor architecture. The *asmg* takes an intermediate representation[4] of the Sim-nML processor specifications as its input and generates an assembler for that processor. The *assembler* so generated produces an ELF object code as output from a given assembly language program. Since Sim-nML provides a generic way of describing a processor architecture, *asmg* provides for the generation of assemblers in an architecture independent manner.

1.2 Related Work

Automation tools for performance modeling of a system is a growing area and a lot of research has been pursued in this area. These previous works have resulted in a set of performance modeling tools using different languages for processor specification.

An **automatic assembler generator**[5] has been developed by George Hadjiyian-

nis, Silvina Hanono, and Srinivas Devadas at Princeton University using the *Instruction Set Description Language ISDL*[6]. ISDL is a machine specification language similar to Sim-nML[1]. It provides constructs for specifying instruction set and other architectural features. Their assembler generator receives an ISDL description as input, and produces an assembler which assembles the compiler's output to a binary file. The assembler generator produces Lex and Yacc files, which when compiled result in an executable program for parsing the assembly language programs and generating the corresponding machine instructions.

A **meta-assembler**[7] which creates assemblers for new architectures has been written for specifications in *SLED*[8], a *Specification Language for Encoding and Decoding*. This language is used to define mappings between symbolic assembly-language and binary representations of instructions. A toolkit, (New Jersey Machine-Code (NJMC) Toolkit) is implemented to help programmers write applications that process machine code–assemblers, disassemblers, code generators, tracers, profilers, and debuggers. The *meta-assembler* is essentially a macro processor with bit-manipulation operators and special support for different integer representations.

The *Language for Instruction Set Architectures* (LISA)[9] is capable of describing the architectural details and pipeline operations of modern processors. The language is designed to describe processor architectures and to enable automated generation of software development tools, such as high-level language compiler, assembler, linker, simulator, and debugger. Furthermore, accurate and consistent documentation can be generated in LaTeX and HTML format. Currently a processor simulator SuperSim[10] has been developed.

1.3 Existing Tools

Following tools have been implemented in our environment.

Cache Simulator[4] provides a mechanism to simulate various caching policies. The designer can use the simulator to study the trade-offs between different caching policies.

Code Instrumentor[4] implements a mechanism to perform analysis and profiling of application programs through the technique of code instrumentation.

Disassembler[11] takes the IR of a processor description and a relocatable binary program in ELF format as input and produces an equivalent program in assembly language of the processor. The disassembler is generic enough to be used for all types of processors. It performs complete symbolic disassembly and is compatible to the *assembler generator*.

Instruction Set Simulator Generator[1] takes Sim-nML specification as input and generates a performance simulator, which in turn takes a binary for that processor and gives the performance based results.

Retargetable Functional Simulator[12] generates a functional simulator for a binary program targeted to run on a processor whose description is given in the Sim-nML.

The following tools are under development.

Timing Simulator to analyze a particular program for timing performance and resource usage. A compiled code simulator generator would generate a higher performance timing simulator.

Compiler Back-End Generator to generate back-end for GNU-C by automatically generating GNU machine description (.md) of a particular processor from Sim-nML.

1.4 Organization of the Report

In this thesis we have designed and implemented an **Assembler Generator** which generates an assembler for a specific processor. The generated *assembler* takes an assembly language program and optionally a configuration file to provide the information for machine specific link-time relocations as input, and generates an ELF format relocatable object file as output.

The rest of the thesis is organized as follows. In chapter 2, we provide a brief introduction to the Sim-nML language and its intermediate representation. In chapter 3, we discuss the basics of the generated assemblers and the assembler directives supported by them. We also discuss the ELF object file format and the implementation of machine independent part of the *assembler*. In chapter 4, we discuss the design of the assembler generator. We have tried to point out the information that is available in the intermediate representation of machine specification written in Sim-nML and how it contributes to the generation of the assembler. In chapter 5, we discuss generic relocation functions supported by the generated assembler, the way to define machine specific relocations. We use a machine specific configuration file for that. We finally conclude in chapter 6. In addition to these we provide a user's manual for the assembler in Appendix A.

Chapter 2

Sim-nML and Intermediate Representation

The base language for our environment is *Sim-nML*[1], a generic processor modeling language. *Sim-nML* is an extension of *nML*[13] machine description formalism. Processor models are written in Sim-nML, using which, various processor specific tools can be generated automatically. To ease the design of various tools an *intermediate representation* (IR) for *Sim-nML* has been designed. A tool called **irg**[4] takes a Sim-nML specification for a processor and converts it to IR.

2.1 The Sim-nML Language

The processor modeling language Sim-nML is used to describe the syntax and semantics of instructions in the instruction set of a processor. The instruction set is described in a hierarchical manner. The common behavior of a class of instructions is captured at the higher level of the tree and the specialized behaviors of the sub-classes are captured in the subsequent lower levels.

Sim-nML grammar has a fixed start symbol *instruction* and two kinds of productions *or-rule* and *and-rule*. The *or-rule* is written as,

$op\ n = n_0|n_1|n_2|...$

and *and-rule* is written as,

$op\ n_0(p_1 : t_1, p2 : t_2, ...)$

$a_1 = e_1\ a_2 = e_2\ ...$

where each n_i is a non-terminal, and each t_i is a token. Each a_i is an attribute name with e_i being its definition.

Sim-nML has some predefined but optional attributes named *image*, *syntax*, *action* and *uses*. The *syntax* attribute describes the textual syntax (assembly language format) of the instruction, the *image* attribute describes the binary coding of the instruction. The

action describes the semantics of the instruction while the *uses* describes the resource-usage model. The assembler generator uses the definition for *syntax* and *image* attributes only.

The Sim-nML example in Figure 2.1 and Figure 2.2 describes a simple processor with four instructions. The *add* and *sub* instructions add and subtract the contents of two general purpose registers respectively. The *jump* instruction changes the current PC value. The second byte of the *jump* instruction specifies an 8-bit branch address. PC refers to the address from which the next instruction has to be fetched. The *move* instruction moves the content of memory addressed by a general purpose register to another general purpose register. The addressing modes used by the example processor are *REG* (register direct), *MEM* (register indirect) and *IMM* (immediate).

```
type index = card(3)
reg PC[1, card(8)]
let byte_order = "big"

mem M[1024, card(8)]
reg R[8, card(8)]

resource Fetch_Unit, Exec_Unit[2], Retire_Unit

mode SHORT = MEM | REG

mode MEM (a : index) = M[R[a]]
syntax = format("(R%d)", a)
image  = format("0%3b", a)

mode REG (i : index) = R[i]
syntax = format("R%d", i)
image  = format("1%3b", i)

mode IMM (n : card(8)) = n
syntax = format("%d", n)
image  = format("%8b", n)
```

Figure 2.1: Sim-nML Specification for a Simple Processor

Addressing modes in the processor are described using *mode rule*. The basic types of Sim-nML include *card*, *int*, *bool*, *float*, *fixed* and *enum*. The resource-usage model is used to specify the micro-architecture details of the processor. A resource is an abstraction of a piece of hardware which can be acquired/released by any instruction in execution such as a register, ALU, the functional unit, ports etc. For the example processor (Fig.2.1) the resources are *Fetch_Unit*, *Exec_Unit* and *Retire_Unit*.

```
op instruction(x : instr_action)
uses = Fetch_Unit #{2}, x.uses, Retire_Unit #{2}
syntax = x.syntax
image = x.image
action = { x.action; }

op instr_action = add | sub | mov | jump

op add (src : SHORT, dst : SHORT)
uses = Exec_Unit #{2}
syntax = format("add %s,%s", src.syntax, dst.syntax)
image = format("00000001%s%s", src.image, dst.image)
action = {
           dst = src + dst;
           PC = PC + 2;
         }

op sub (src : SHORT, dst : SHORT)
uses = Exec_Unit #{2}
syntax = format("sub %s, %s", src.syntax, dst.syntax)
image = format("00000010%s%s", dst.image, src.image)
action = {
           dst = src - dst;
           PC = PC + 2;
         }

op mov (src : SHORT, dst : SHORT)
uses = Exec_Unit #{1}
syntax = format("move %s, %s", src.syntax, dst.syntax)
image = format("00000011%s%s", dst.image, src.image)
action = {
           src = dst;
           PC = PC + 2;
         }

op jump (target : IMM)
uses = Exec_Unit #{3}
syntax = format("jmp %s", target.syntax)
image = format("00000100%s", target.image)
action = {
           PC = PC + target;
         }
```

Figure 2.2: Sim-nML Specification for a Simple Processor contd ...

A special symbol, $ is used to denote the start address of the current instruction. This symbol is used to specify a relative branch address for example.

2.2 Intermediate Representation

A processor specification in Sim-nML language is in a human readable text form. It is, however, wasteful for each tool to have its own parser to read the Sim-nML specification. A tool **irg**[4] was developed in an earlier work to simplify this process. It converts the Sim-nML description to an intermediate form(IR). The IR contains all useful information available in the original input without any unnecessary or redundant information. It is flexible and easy to use and facilitate the design of processor specific tools like simulator, disassembler, assembler, compiler back-end generator etc.

The IR is organized as a collection of various tables. We use the following tables for our work.

- **Meta table:** The information in this table is needed to locate other tables in the IR. This is a "table of content" that contains the information about the location and name of other tables.
- **Constant table:** This table holds all the constant declarations in the Sim-nML processor specification. The useful constants for *assembler generator* are **byte_order** and **processor_name**. They specify the endian-ness and the name of the processor respectively.
- **Attribute table:** This table holds the name of all Sim-nML attributes. A corresponding ID is given to each attribute which is used in all other tables to refer to this attribute.
- **And-Rule table:** This table holds the information about all and-rules. From this table we get an index to the *syntax table* and *image table*. This index gives the syntax and image of the and-rule.
- **Or-Rule table:** This table holds the information about children of all or-rules.
- **Syntax table:** This table holds the *syntax* attribute definitions of all and-rules. It provides the assembly language syntax and parameters for various instructions.
- **Image table:** This table holds the image-record associated with the *image* attribute definitions of all and-rules. It, therefore describes the binary coding of the instruction.

To get all informations from the above specified tables we need to look at some more tables described below.

- **Identifier table:** This table holds the name of all the identifiers (other than those specified in the *constant table* and in the *resource table*). This name is stored as an index to the *string table*.

- **String table:** This table holds strings used in the Sim-nML model. The strings are terminated by a null character. All identifier names are read from this table.
- **Integer table:** Similar to the string table, this table holds various integers in the specifications. In other tables, an index into this table is used for representing array of integers.
- **Prefix-Attribute-Definition Table:** This table holds the expressional definition of all the attributes and is referred to when some operation (e.g. arithmetic, bit manipulation) is used in the definition of the *syntax* or *image* attributes.

Chapter 3

The Assembler

Assembler translates an assembly language program for a processor into its machine language (object code). It generates its output in a relocatable form where by the program can be linked/ loaded in a position independent manner with the other programs. Besides the object code, the assembler outputs a human readable listing of the source program and its translation. It provides error messages interspersed with the code, symbol table, relocation table and section table.

The process of assembly involves *lexical analysis*, *syntactic analysis*, *object code generation*, *symbol table management* and *forward references fix-ups*. A two pass assembler processes the instructions as completely as possible during the first pass. The second pass over the code is used to fix all forward references in the instructions. A simple overview of a two pass assembler is shown in Figure 3.1.

We generate a two pass *assembler* (referred as *asm* hereinafter) in this work. In the first pass, the generated assembler parses its input, gathers all relevant information and for all undefined symbol references assumes that the definition for the symbol will appear later. It also checks for syntax and semantic errors and reports them. The second pass of the assembler begins only if the first pass of assembly completes successfully. In the second pass the values for the forward references are substituted and the output object code is generated. All undefined references at this stage are assumed external. The *asm* takes an additional *configuration file* as its input. This file contains relocation information specific to the processor and helps in the generation of relocation table in the object module. The output of the *asm* is generated in a relocatable ELF[2] object file format.

3.1 Output File Format

The format of the output file generated by the *asm* is ELF[2] (Executable and Linking Format). There are three main types of ELF object files.

- A *relocatable file* holds the code and the data. It can be linked with other similar object files to create an executable or a shared object file. The *asm* produces output in this format.

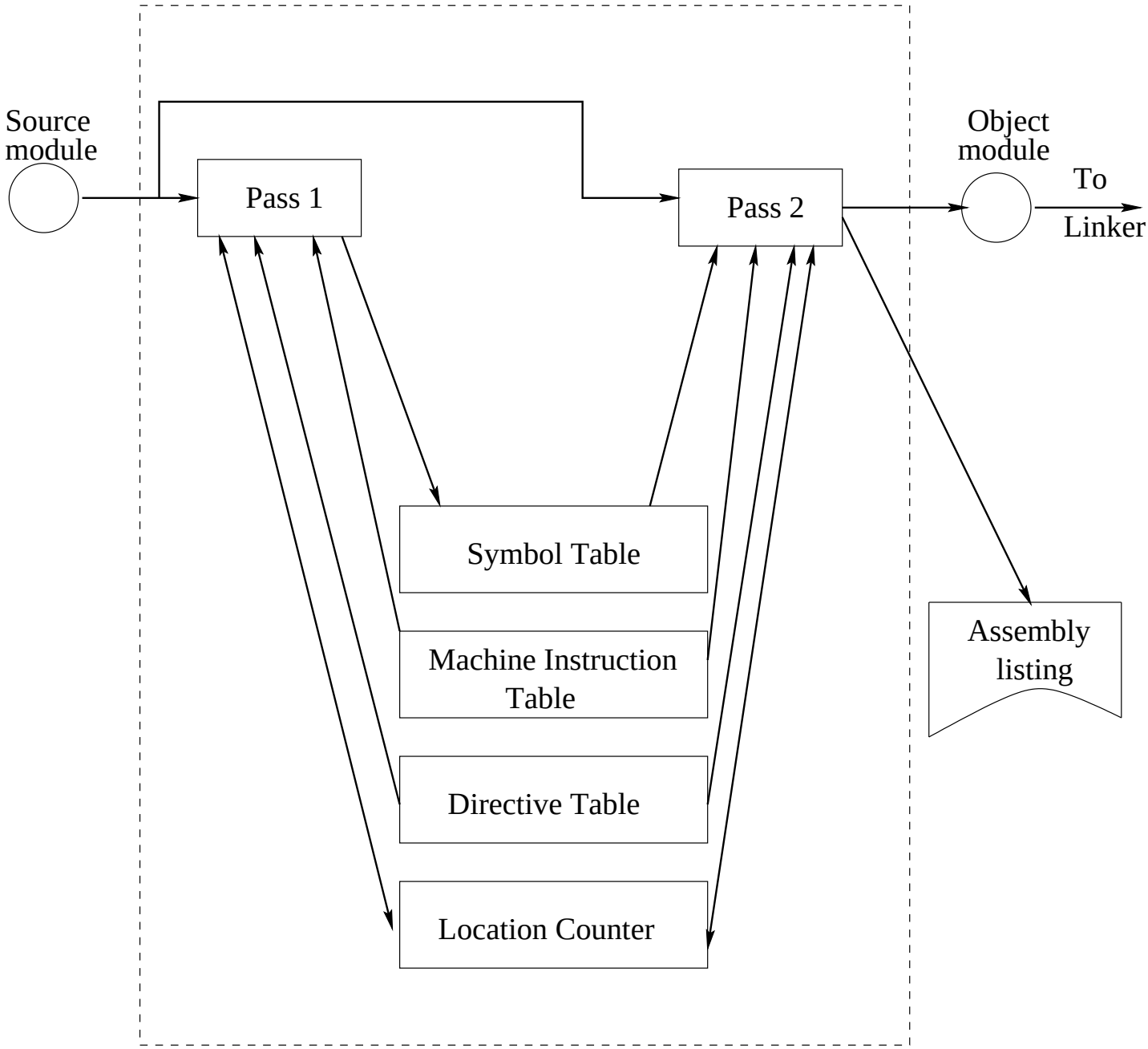


Figure 3.1: Two-pass Assembler

- An *executable file* holds a program suitable for execution.
- A *shared object file* holds code and data suitable for linking.

Figure 3.2 shows a linking view of an object file. In this file only the ELF header has a fixed position. The remaining portion is variable and depends on the actual file.

A program header table provides information on how to create a process image. A section header table contains information that describes the sections in the file. Each section has an entry in this table. Each entry gives information such as the section name,

the section size etc. Relocatable object files which are used during the linking must have a section header table.

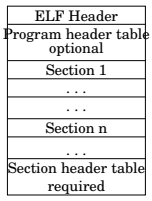


Figure 3.2: Object file format (Linking view)

In the output file generated by *asm*, first the ELF Header is written. It then has the sections and the section header table.

3.1.1 ELF Header

An ELF header resides at the beginning of the object file and holds a "road map" describing the file's organization. The file format is designed to be portable among machines of various sizes, without imposing the size of the largest or the smallest machine. The initial bytes mark the file as an object file and provide machine-independent data. This data provides necessary information to decode and interpret the file's contents. The ELF header contains the information regarding the type of object file, required architecture for an individual file, size of ELF header, size of section header etc.

3.1.2 Sections

The data in the object file is organized in various sections that holds program code, data and control information. Each section has a type and an associated attribute to mark the section as read only, executable, relocatable etc. An object file's section header table provides the location of all sections in the file.

Various predefined sections which are usually generated by the *asm* are listed below.

- **.bss:** This section holds uninitialized data that contribute to the program's memory image.
- **.data** and **.data1:** These sections hold initialized data that contribute to the program's memory image.
- **.rel*name*** and **.rela*name*:** These sections hold relocation informations. The *name* is supplied by the section to which the relocation applies. For example, *.rel.text* contains relocation information for symbols used in *.text* section.
- **.rodata** and **.rodata1:** These sections hold read-only data that typically contribute to a non-writable segment in the process image.

- **.shstrtab:** This section holds section names as null terminated character strings (see section "String Table").
- **.strtab:** This section holds strings, most commonly the strings that represent the names associated with symbol table entries (see section "String Table").
- **.symtab:** This section holds a symbol table (refer to the section "Symbol Table").
- **.text:** This section holds the "text" or executable instructions, of a program.

The object file generated by the *asm* consists of at least three sections named as *.bss*, *.data* and *.text* regardless of whether they exists in the input assembly program or not. These sections have a size of zero bytes if they are not defined in the input assembly language program. The generated output file also contains a *.shstrtab* section. It contains the name of all the sections in the object file.

3.1.3 String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbols and sections name. One references a string as an index into the string table section. The first byte at index zero always holds a null character. Likewise, the last byte in various string tables hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted.

3.1.4 Symbol Table

An object file's symbol table holds information required to relocate a program's symbolic definitions and references. It consists of entry for *symbol name*, *symbol value*, *symbol size*, *symbol info*, and name of the section in which the symbol is defined. A symbol table index 0 serves as an undefined symbol index.

3.1.5 Relocation Table

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address during execution. Relocatable files must have "relocation entries" which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image.

A relocation entry contains the information about the location at which the relocation action is performed, the symbol table index with respect to which the relocation must be made, and the type of machine specific information which defines the relocation

action. A relocation requires to reference two other sections - the symbol table section and the section that is modified.

3.2 The Assembler Directives

The assembler directives are not translated into the machine code. They are instructions to the assembler to perform various bookkeeping tasks, storage reservation and initialization and other control functions. The directives used by the *asm* are compatible with syntax as well as with semantics of GNU assembler GAS[3] that is available as part of the GNU binary utilities and the GNU C compiler.

All assembler directives have names that begin with a period ('.'), and are the same regardless of the target machine. Some directives however have different interpretation than the ones used in the GNU assemblers and are discussed here.

- **.align abs-expr, abs-expr, abs-expr:** This pseudo operation is used for aligning the location counter (in the current subsection) to a particular storage boundary. The GAS behavior is inconsistent for this directive and depends on the architecture for which it is configured. e.g. For some processor architecture the directive *.align 3* advances the location counter until it is a multiple of 2^3, while for some other processors the directive *.align 8* is used for the same purpose. In *asm* the directive *.align 3* is interpreted as to advance the location counter to a multiple of 2^3. The second and third *abs-expr* have same interpretation as in the GNU assembler.

- **.file string:** This pseudo operation provides the name of the file which was translated to give the assembly program. For example it is used by the C compiler to provide the name of the C file which was compiled to the assembly language program. However its handling depends on how GAS is configured. The *asm* takes the *string* and inserts it into the symbol table as a symbol of type FILE.

- **.sbttl "string", ... :** This pseudo operation is used to format the output listing. It provides a *sub-heading* which is inserted immediately after the title in the assembly listing. GAS doesn't parse the *string* and emits it as it is. The *asm* parses the string making it possible to use variables or predefined constants in the sub-heading.

- **.title "string", ... :** This pseudo operation provides the title line for the assembly listing. In GAS, this is used as the second line and the string is not parsed. In *asm*, we use this as the first line and parse the *string* as in the *.sbttl* pseudo operation. The default title is "`file listing: $__FILE Page number: $__PAGE`", where __FILE and __PAGE variables are described later.

Certain other directives like *.def*, *.dim*, *.endef*, *.ident*, *.lflags*, *.linkonce*, *.mri*, *.scl*, *.size*, *.sleb128*, *.tag*, *.type*, *.val*, and *.uleb128*. which are meaningful in GNU assembler while generating COFF[14] output. The *asm* accepts these directives and ignores them.

In addition, the *asm* provides some predefined constants which can be used anywhere in the assembly language file. They are as follows.

1. **__FILE** is the name of the current assembly language file.
2. **__LINE** is the current physical line number of the assembly language file being processed.
3. **__PAGE** is the page number of the generated listing.
4. **__DATE** is the system's current date.
5. **__TIME** is the system's current time.

3.3 Implementation Details

The compilation of the generated *assembler* requires some supporting files which are machine independent. These files contain programs for command line parsing, evaluation of assembly directive, preparation and manipulation of assembly sections, writing of object file in ELF format etc.

3.3.1 Evaluation of Assembly Directives

The *pseudoOp.c* and *pseudoOp.h* are the two files which contain all the necessary information for evaluation of assembly directives. In the first pass of assembly all directives are inserted into a hash structured symbol table for faster reference. The informations inserted are as follows.

- **Name:** This is a null terminated string that represents the name of the pseudo-op. A symbol is compared against this name while parsing the input.
- **Token:** This is the corresponding token returned to the parser when a pseudo-op is matched against the *Name*.
- **Function pointer:** This is a pointer to the corresponding function which is called to process the recognized pseudo-op.

As soon as a pseudo-op is recognized and the corresponding statement in the assembly language program is parsed, the function associated with that pseudo-op is called. This function now takes care of further processing. For example, if ".byte 2+3" is found in the assembly language program, the parser returns a token AS_BYTE. The corresponding function *as_byte* is then called with the argument 5 (the result of parsing the expression 2+3). This function then calls appropriate routine to write value 5 in the specified section. All routines for writing sections are defined in *frags.c* file, the corresponding header file is *frags.h*. All writing is carried out in the second pass only and nothing is written in any section in the first pass.

All sections are represented by contiguous blocks of memory. We have defined a *section* structure which contains the pointer to this memory block, the size of the block

and the current offset in the section where the writing takes place. For all sections the section pointer is set to NULL initially and, section size and section offset are set to zero. While generating the output in a section, the byte order is changed if the endian-ness of the target processor is not the same as that of the host processor.

The assembly is also controlled by two assembler directives *.if* and *.else*. The instructions are assembled only when the corresponding expression evaluates to *true*.

All writing routines have the following common structure.

1. if conditional assembly evaluates to false, return.
2. if the pass == first then compute the new offset and return.
3. else
   (a) if the endian-ness of the target processor differ from the host processor, swap bytes.
   (b) write data in the section at the current offset.
   (c) update offset in the section.

3.3.2 Symbol Table and Relocation Table Management

These tables are used for bookkeeping information about a symbol defined, used, or relocated in the assembly language program. All these tables are hash structured.

Symbol Table

As soon as *asm* recognizes a symbol definition it inserts it into the symbol table. The symbol is inserted only once and further definitions for the same symbol results in an error. The symbols which are not defined but used are inserted into symbol table as type *undefined*. *asm* treats all undefined symbols as external. The informations stored in *symbol table* in the memory are as follows.

- **Name:** A character pointer to the first character of the name of the symbol.
- **Value:** This holds the address or the value of the symbol depending on the context.
- **Length:** This represents the size for the symbol defined using *.comm* or *.lcomm* directives.
- **Info:** This consists of the TYPE and BIND information of the symbol. (see ELF object format[2].)
- **Section:** This contains the symbolic name of the section in which the symbol was defined or used.

- **Type:** This contains the type of the symbol which can be *defined*, *undefined*, *defined and used* etc.

The operations performed on symbol table are *look up*, *insertion* and *modification*. For each of these operations a hash index is calculated using the *name* of the symbol. If the symbol found at this index is different from the one that is stored, the symbol on the chain is examined until the symbol is found or the chain ends. The symbol is then returned or modified depending on the function call. The insertion of the symbol is done only if it doesn't exist in the table.

Relocation Table

The symbols in relocation table are inserted as many times as they are used in the input file. Thus a symbol in relocation table can have multiple entries each defining the distinct use of the symbol. The *type* of relocation is machine specific and differs even in syntax from one processor architecture to another. To provide a uniform behavior, *asm* defines some generic relocation operations. User for specific processor provides a configuration file which have a correspondence between the machine specific relocation types and the generic operations. *asm* uses a command line option `-c` to provide the name of the configuration file.

The informations stored in *Relocation Table* are as follows.

- **Name:** A character pointer pointing to the first character of the name of the symbol.
- **Value:** This holds the address where the relocation is to be applied.
- **Type:** This holds the machine specific relocation type.
- **Addend:** This member specifies a constant addend used to compute the value to be stored into the relocatable field.
- **Section:** This contains the symbolic name of the section in which the symbol was used.

The operations performed on relocation table are *look up*, and *insertion*. For each symbol reference in relocation table, a corresponding entry in symbol table for the symbol is searched. The information required for relocation entry in the ELF format is calculated and the entry is inserted in the relocation table.

3.3.3 The Big Numbers

The *asm* supports integer constants larger than the machine based integers. These are called *bignum*. The *bignum* have the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

Two type of bignums are available in GAS and are provided in *asm* also. They are `quad` and `octa`.

- **quad bignum:** The assembler directive *.quad* is used to refer to quad bignum. Quadnums are 8-byte integers. The operations available for quadnums are addition, subtraction, multiplication, bit-shift, bit-or, and bit-and. In *asm* these operations are implemented using bit-shift, bit-and and bit-or operations on 4-byte integers.
- **octa bignum:** The assembler directive *.octa* is used to refer 16-byte integers. The operations available for octanums are the same as those for quadnums. The octanums are viewed as a pair of quadnums and all these operations for octanum are implemented using operations on quadnums.

3.3.4 Generation of the Output Object File

After the second pass of assembly, the output object file is written in the ELF format. The formatting part of output file is implemented in file *elf.c*, the corresponding header file is *elf.h*. The *elf.h* file contains the definitions for structure of ELF header and section tables. Three sections *.bss*, *.text* and *.data* are written regardless of whether they are present in the input file or not. Each section occupies one contiguous (possibly empty) sequence of bytes within a file. They may not overlap, and no byte in a file resides in more than one section.

Generation of `.symtab` and `.rela`*name* section

The information stored in *.symtab* section comprises of, *name* which is an index to *.strtab* section, *value*, *size*, *section name* and *info*. The *info* member specifies the symbol's *type* and *binding* attributes. The algorithm used for preparation of *.symtab* section is discussed below.

1. calculate total number (no) of symbols in symbol table.
2. initialize 1st entry (index 0) for *.strtab* section. This entry (STN_UNDEF) is reserve, and serves as an undefined symbol index.
3. while $no > 0$
   (a) read symbol from symbol table, and write *.symtab* section entry by taking care of endian-ness.
   (b) write symbol name in *.strtab* section.
   (c) if symbol is relocatable, prepare relocation table entry.
   (d) decrement *no*.

After writing the output file in ELF format *asm* exits normally. The default name for output object file is *b.out*. It can be changed by using the `-o` command line option. Conventionally, the object file name ends with '.o'.

3.3.5 Error Reporting

Two kinds of errors are reported by the *asm*. These are *Warning Messages* and *Error Messages*. The *asm* writes the warnings and error messages to the standard error file (usually the terminal).

Warning messages have the following format.

Warning:file_name:NNN:Warning Message Text

Where NNN is the line number, file_name is the name of the current input file and the message text provides the warning message.

Error messages have a format similar to the warning messages as represented below.

Fatal:file_name:NNN:Error Message Text

The file name, line number and message text are derived as in the case of warning messages. The *asm* generates the output file for the assembly program even in the presence of warnings. In case of errors, the output file is not generated.

3.4 List File Generation

The generated assembly listing includes assembly program, its equivalent machine code, error messages, a section table, a relocation table and a cross-reference symbol table. An example listing is shown in Figure 3.3 and Figure 3.4.

The first column in the listing shows the line number. The second column gives the value of the location counter immediately before the corresponding statement is assembled. The third column shows the machine code that the statement is assembled into, and the remainder of each line is the source code just as it is presented to the assembler. If an error is found, the error message is output on the line following the line containing the error. The "R" in the *line 16, 17* and *20* shows that these instructions use symbol values which should be relocated by the linker.

The cross-reference symbol table summarizes the information regarding the identifiers in the program. The *value* is the value of location counter where the symbol is defined. Since *printf* is undefined it contains a value **UND**. The *section* specifies the name of the section in which the symbol is defined and *info* provides the binding and type information for the symbol as defined for ELF[2] file format. The relocation table contains the name of the symbol, address where relocation should be applied, the section name and the machine specific relocation type. The section table summarizes the information regarding generated sections. It gives the start address, end address and the size of all generated sections.

```
file listing: ppc_test.s  Page number: 1

   1 00000000                .file              "test.c"
   2 00000000                gcc2_compiled.:
   3
   4 00000000                .section           ".rodata"
   5 00000000                .align  2
   6 00000000                .LC0:
   7
   8 00000000 53554d3a       .string            "SUM:%d\n"
   9          25640a00
  10 00000000                .section  ".text"
  11 00000000                .globl  main
  12 00000000                .type               main,@function
  13 00000000                main:
  14
  15 00000000 901f0010       stw 0,16(31)
  16 00000004 3d200000 R     addis 9,0,.LC0@16:16
  17 00000008 38690000 R     addi 3,9,.LC0@16:0
  18 0000000c 809f0010       lwz 4,16(31)
  19 00000010 4cc63182       crxor 6,6,6
  20 00000014 48000001 R     bl printf


file listing: ppc_test.s  Page number: 2

                --------------------------------
                    Symbol Table Information
                --------------------------------

Symbol          Value           Section     Info(bind - type)

.LC0            00000000        .rodata     LOCAL-OBJECT
gcc2_compiled.  00000000        .text       LOCAL-OBJECT
main            00000000        .text       GLOBAL-OBJECT
printf          *UND*           .text
test.c          00000000        *ABS*       LOCAL-FILE
```

Figure 3.3: A Sample Assembly Listing for PowerPC Processor

```
file listing: ppc_test.s  Page number: 3


                ---------------------------------
                   Relocation Table Information
                ---------------------------------


Symbol              Address     Section    Relocation Type

printf              00000014    .text       8
.LC0                0000000a    .text       4
.LC0                00000006    .text       6



file listing: ppc_test.s  Page number: 4


                ---------------------------------
                    Section Table Information
                ---------------------------------


Section             Start       End             Size

.bss                00000034    00000034        00000000
.data               00000034    00000034        00000000
.rodata             00000034    0000003c        00000008
.shstrtab           0000003c    00000080        00000044
.strtab             00000080    000000b2        00000032
.symtab             000000b2    00000132        00000080
.text               00000132    0000018e        0000005c
.rel.text           0000018e    000001a6        00000018
```

Figure 3.4: A Sample Assembly Listing Contd ...

Chapter 4

The Assembler Generator

In this chapter we discuss the design and implementation of the *assembler generator (asmg)*. The *asmg* takes a processor model in its IR form, and generates a two pass assembler specific to that processor. The generated assembler consists of a file containing specification for the *Lex program*[15], a file containing specification for the *Yacc program*[16], and a keyword file used for the token generation. These specification files are used for the lexical and syntactic analysis of the assembly language program. These files are generated from the IR description of the processor model.

The generated files consist of all the information about the processor e.g. lengths of the instructions, parameters for the instructions, binary representations of the instructions, endian-ness of the processor, which are all relevant to the assembler. In addition to these, the assembler uses a C module to handle symbol table operations, a parser and analyzer module for the pseudo operations, and a C module to write output file in ELF format. These files are independent of the processor model and are the same for all assemblers that can be generated. We have already discussed the processor independent files in Chapter 3. In this chapter, we describe the algorithms used in the generation of specification files from the IR representation of the processor model.

4.1 Overview

The basic structure of the assembler generator is shown in Figure 4.1. The generation of assembler is done in two passes over the IR. In order to generate the assembler following steps are followed.

1. PASS ONE
   (a) Initialization of *asmg*
      i. identification of the data encoding (endian-ness) of the host processor on which *asmg* is running.
      ii. checking the integrity of the IR file by reading the magic number in IR header.

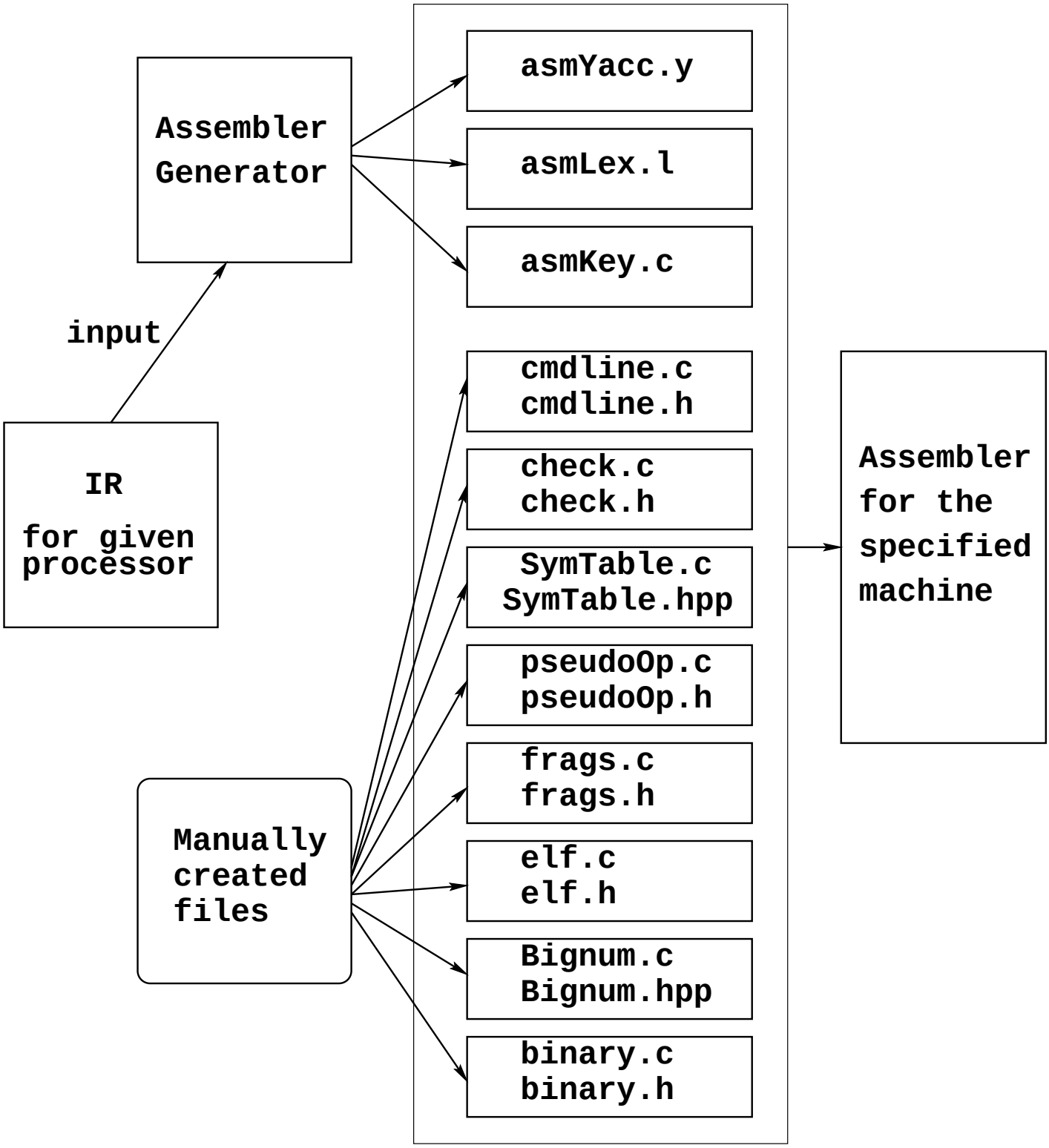


Figure 4.1: Structure of the Assembler Generator *asmg*

iii. from IR header determine the data encoding used in the IR. If data encoding of the host processor is different from that in the IR, then a flag is set to indicate that the data read from the IR file must be converted to the host byte order before its use.

(b) for each and-rule R in Sim-nML description; do,

i. read syntax string for R using *integer table*, and *syntax table*. Parse the syntax string using algorithm described in section 4.2.2.

ii. if the processor description contains instructions with same syntax, remember the corresponding Sim-nML rules. Only one Yacc rule is generated for all these instructions.

2. PASS TWO

(a) Initialization of Yacc specification, and keyword file. The constant starting part of these files consisting of variable declarations, and some part of Yacc definition section is written.

(b) For each rule in IR with distinct syntax string a corresponding Yacc rule is generated. The *and-rule* and the *or-rule* are considered differently for generation of these rules.

(c) Keywords are generated corresponding to each assembly mnemonic. They are used in token generation.

The *asmg* generates the following files.

- `asmgYacc.y`, which contains grammar rules.
- `asmgKey.c`, which contains assembly language mnemonics and the corresponding tokens.
- `asmgLex.l`, which contains the scanner rules.

The description of the *asmg* is best described using an example Sim-nML processor specification. Since only the *syntax* and *image* attributes are relevant for *asmg*, other attributes have been dropped from the Sim-nML model shown in Figure 4.2 and Figure 4.3.

4.2 Implementation Details

The relevant information from IR is extracted and machine dependent files are generated using this information. These files after compilation produces the assembler for the specified processor. A shell script is provide to simplify the generation of the assembler. It first generates the assembler files and then compiles them to produce the assembler.

```
type index = card(3)
reg PC[1, card(8)]

let byte_order = "big"
let processor_name = "none"

mem M[1024, card(8)]
reg R[8, card(8)]

mode SHORT = MEM
       | REG

mode MEM (a : index) = M[R[a]]
syntax = format("(R%d)", a)
image  = format("0%3b", a)

mode REG (i : index) = R[i]
syntax = format("R%d", i)
image  = format("1%3b", i)

mode IMM (n : card(8)) = n
syntax = format("%d", n)
image  = format("%8b", n)

op instruction(x : instr_action)
syntax = x.syntax
image = x.image

op instr_action = alu_op
       | jump
       | test_op

op alu_op(src:SHORT, dst:SHORT, aa:alu_action)
syntax = format("%s %s,%s", aa.syntax, src.syntax, dst.syntax)
image = format("0000%s%s%s", aa.image, src.image, dst.image)

op alu_action = a_add
       | a_sub

op a_add()
syntax = "add"
image = "0001"
```

Figure 4.2: Sim-nML specification for an example processor

```
op a_sub()
syntax = "sub"
image = "0010"

op jump (target : IMM)
syntax = format("jmp %s", target.syntax)
image = format("10000000%s", target.image)

op test_op(in:intype)
syntax = format("test %s", in.syntax)
image = format("1111%s%s", in.image<2..3>, in.image)

op intype(src:REG, src1:REG)
syntax = format("%s %s", src.syntax, src1.syntax)
image = format("11%s%s", src1.image, src.image)
```

Figure 4.3: Sim-nML specification for an example processor contd ...

4.2.1 Generation of the Yacc Specification File

The format of the generated Yacc specification file is as follows.

```
[ definitions ]
%%
[ rules ]
[ %%
  [ user functions ]
]
```

Here **Definitions** is the section where the variables are defined that are used later in the grammar. It also contains *#include* directives. **Rules** is the section that contains grammar rules for the parser. These rules are generated according to the processor specifications in the IR. **User functions** is the section that contains the definition of the functions used in the rules section.

The hierarchical structure of processor description in Sim-nML is preserved in generation of Yacc rules. For each rule in processor description we get a corresponding Yacc-rule (except in the case when two rules have the same syntax. This case is described later). For the purpose of parsing and grammar rules generation, *mode-rule* and *op-rule* in Sim-nML specification are not differentiated. Only the attributes *syntax* and *image* in op/mode rules are used by the *assembler generator*.

The *syntax* attribute is used to generate the grammar rule where the name of the non terminal (on the left side of the production) is same as the name of the op/mode rule.

The format string in the *syntax* attribute is used for the token generation. The generated keywords for the tokens are all in capital letters and are prefixed by the string `AS_` to avoid conflicts with predefined constants. For each non-terminal, a definition %type is generated to specify the value returned by the rule. Similarly, for each terminal symbol, a definition %token is generated to specify the value associated with the token. The *mode and-rule* and *op and-rule* may differs in %type definition. The *op rule* returns a pointer to a character array. The *mode-rule* returns an expression structure consists of the expression tree created while parsing an expression or a pointer to a character array depending on the context. The *image* attribute is used for the generation of the action-part in the Yacc rule. The Yacc rules used for parsing of arithmetic expressions and assembly directives are fixed and are independent of the processor under consideration.

The processor specific part of generated Yacc specification file for example processor (Figure 4.2) is shown in Figure 4.4.

```
instruction  :  instr_action  { $$ = $1; }
        ;
instr_action  :  alu_op       { $$ = $1; }
        | jump                { $$ = $1; }
        | test_op             { $$ = $1; }
        ;

test_op  :  AS_TEST intype {
                string str;
                str = "1111" + bitselect($2,2,3) + (string)$2;
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
                delete [] $2;
        }
        ;

intype  :  REG REG {
                string str;
                str = "11" + (string)$2 + (string)$1;
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
                delete [] $2;
                delete [] $1;
        }
        ;
```

Figure 4.4: Generated Yacc specification file for the example processor

```
REG  :  AS_R {
                string str;
                str = "1" + setsize(itosul($1.val[0] ,3), 3);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
        }
       ;
jump  :  AS_JMP IMM {
                string str;
                str = "10000000" + (string)$2;
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
                delete [] $2;
        }
       ;
IMM  :   expr {
           if ($1.val[0] < 0 || $1.val[0] > 255){
                yyerror("Value: %d Is out of range", $1.val[0]);
                errorNo++;
                $$ = new char[1];
                $$[0]='\0';
           }
           else{
                //calculate image
                string str;
                sizeInstr = 8;
                relocateSymbol(&$1);
                str = setsize(itosul($1.val[0] ,8), 8);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
           }
         }
       ;
alu_op  :  alu_action SHORT ',' SHORT {
                string str;
                str = "0000" + (string)$1 + (string)$2 +
                      (string)$4;
                $$ = new char[str.size()+1];
```

Figure 4.5: Generated Yacc specification file for the example processor contd ...

```
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
                delete [] $1;
                delete [] $2;
                delete [] $4;
        }
      ;
SHORT  :  MEM
            { $$ = $1; }
        | REG
            { $$ = $1; }
      ;
MEM  :  '(' AS_R ')' {
                string str;
                str = "0" + setsize(itosul($2.val[0] ,3), 3);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
        }
      ;
alu_action  :  a_add
            { $$ = $1; }
        | a_sub
            { $$ = $1; }
      ;
a_sub  :  AS_SUB {
                string str;
                str = "0010";
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
        }
        ;
a_add  :  AS_ADD {
                string str;
                str = "0001";
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
        }
        ;
%%
```

Figure 4.6: Generated Yacc specification file for the example processor contd ...

The algorithm for generation of Yacc-rules can be summarized as follows.

1. start from *instruction* node.
2. for each rule R in Sim-nML description; do,
   (a) if R is an *and-rule*,
      i. read *number of parameters* from *and-rule table*.
      ii. read syntax string and the image string corresponding to R using *integer table*, *syntax table* and *image table*.
      iii. parse the syntax string using algorithm described in section 4.2.2 and write the rule in Yacc file.
      iv. parse the image string and write the action using the algorithm in section 4.2.2.
   (b) if R is an *or-rule*,
      i. Read *number of children* from *or-rule table*.
      ii. for all children write a rule in Yacc file. The corresponding action is *$$ = $1*.

The user defined function part of the Yacc specification is constant and appended at the end of the generated file. This part consists of the functions to output the assembly listing in a predefined format.

Merging of Yacc Rules

Only one Yacc rule is generated for all the instructions that have the same assembly language syntax but generate different images depending upon the arguments. For example, a processor can have an instruction *JUMP target* where, the *target* can take any integer value. If the value of *$ - target* is less than 2^{15} the *jump* can be coded as a relative jump. For target addresses larger than that, the assembler generates an absolute jump (if the target can fit in 16 bits). The machine codes in two different cases are different. The generated Yacc rule for each of these instructions is `AS_JUMP` ***expr***. If we write two such rules in the Yacc specification file the parser generated by this file contains a *reduce-reduce* conflict. In the presence of this conflict the generated parser always recognizes only one rule (the rule which comes first in the specification file) and ignores all other rules. To avoid this we merge all these rule in one Yacc specification rule. The algorithm used is given below.

1. Create a list of rules with the same syntax (the list is created for each rules in the first pass of generation of specification file).
2. Sort this list on the size of the parameters.
3. Write the rule in Yacc specification file.

```
branch  :  AS_JUMP expr {
           if (!relocatableSymbol && $2.val[0] >= -32768
               && $2.val[0] <= 32767){
                sizeInstr = 15;
                relocateSymbol(&$2);
                string str;
                str = "000000001"
                       + setsize(itosul($2.val[0], 15), 15);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
           }
           else if (!relocatableSymbol && $$2.val[0] >= -65536
               && $2.val[0] <= 65535){
                sizeInstr = 16;
                relocateSymbol(&$2);
                string str;
                str = "10000001"
                       + setsize(itosul($2.val[0], 16), 16);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
           }
           else if(!relocatableSymbol){
                yyerror("Value: %d Is out of range" ,$2);
                errorNo++;
                $$ = new char[1];
                $$[0]='\0';
           }
           else if(relocatableSymbol){
                sizeInstr = 16;
                relocateSymbol(&$2);
                string str;
                str = "10000001"
                       + setsize(itosul($2.val[0], 16), 16);
                $$ = new char[str.size()+1];
                strncpy($$,(char *)str.c_str(),str.size());
                $$[str.size()]='\0';
           }
         }
         ;
```

Figure 4.7: An example merged Yacc Rule

4. For each rule in the list,

   (a) Write a Yacc action corresponding to the rule with a condition on the parameter.

The generated rule for the example *JUMP* instruction is shown in Figure.4.7. The *relocatableSymbol* is a boolean variable used by the Yacc specification to mark the variable as relocatable. The variables which are undefined or whose value depends on where the program is loaded in the memory are referred as relocatables. In case of relocatable symbols the instruction with the largest parameter size is generated to give linker the maximum flexibility.

4.2.2 Extraction of Syntax and Image of Instructions

The IR of the processor specification contains *syntax* and *image* records for all the instructions. We extract this information for each *op-rule* starting from *instruction* node. These records encode the syntax of an assembly language instruction, corresponding binary image and information about the arguments. The information about the arguments is found with the help of the *and-rule table* and *integer table*.

The expression corresponding to *syntax* record of instructions does not contain the verbatim syntax of the instructions. For example, in IR the `jump` instruction described in Figure 4.3 has the syntax string as "jump %s{0.3}". The instruction takes one argument specified by the %s. The first integer value enclosed in {} specifies the parameter number, and the second value specifies the attribute name. For instruction "jump" the argument is described by a *mode-and rule* named as `IMM`. The algorithm used to parse **syntax string** is described below.

1. read syntax string corresponding to the current *op-and-rule* from syntax table.
2. for all characters in the syntax string; do,
3. if the character is a '%',
   (a) read the next character, it should be one of 'd', or 's' or 'x'. Remember this character.
   (b) read two integer values enclosed in between { and }. The first integer gives the parameter number and the second integer gives the index for corresponding attribute. Read the name of the parameter in the variable *name* using integer table and identifier table.
   (c) if the remembered character was 's' write this *name* in Yacc specification file, for the above `jump` instruction the *name* corresponds to `IMM`.
   (d) If the remembered character was 'd' or 'x', the parameter index to the integer table gives the type and value for the parameter. Write these values as *expr* in Yacc specification file. *expr* is a predefined Yacc rule for evaluation of the integer expressions.

4. else if the scanned character is an alphanumeric character append it into a string variable *nameStr*.

5. else if the scanned character is a *blank*, write the value of *nameStr* in the Yacc specification file after prefixing it with `AS_` string. This represents a TOKEN. The generated tokens are changed into upper case if the generated assembler is non case-sensitive otherwise they are retained as extracted from the syntax expression. Corresponding entry in keyword file is also made for this TOKEN.

6. else if the scanned character is any punctuation mark e.g '(', '@', ')' etc., it is written in Yacc file as it is.

Similarly, the string corresponding to the *image* record of instructions also does not contain the binary information for instructions verbatim. For the same `jump` instruction the image expression is "10000000%s{0.4}". If instruction "jump 20" is assembled, then the corresponding generated binary image should be "1000000000010100" The algorithm used to parse **image string** is as follows.

1. read image string corresponding to current *op-and-rule* from *image table*.

2. for all characters in the image string; do,

3. if the character is a '0' or '1', write it in the Yacc specification file.

4. else if the character is a '%',

   (a) read the next character, if it is 'd' or 's' or 'x' remember the character. Otherwise the image is of type %*integer*b e.g. %2b. The associated *integer* value is used to check the size of the operand at the time of assembly.

   (b) read two integer values enclosed in between { and }.

      i. if the first value read is negative then it is an index to *prefix attribute table* and second value is the length of corresponding expression. Read the expression from *prefix attribute table* and write it into the Yacc specification file.

      ii. else write the Yacc parameter number.

4.2.3 Generation of the Lex Specification File

The generated Lex specification file works as a scanner. The rules are written in the form of regular expression. The format of generated Lex specification file is,

```
definitions
%%
rules
%%
user_subroutines
```

Definitions is the section which contains definition for all the variables used in *rules* section. The definitions section in the generated Lex specification file is shown below. Here, the left hand side shows the variable name and the right hand side provides its definition.

```
L         [a-zA-Z_]
D         [0-9]
OP        [!-&(-/:-@[-^{-}]
HEXDIGIT  [0-9a-fA-F]
OCTAL     [0-7]
DEC       "."
DOLLAR    "$"
EXP       [eE]
PLUS      "+"
MINUS     "-"
```

In the rules section, these variable names are used within braces {}.

Rules is the section which provides the way of handling tokens. In this section, the left hand side contains the pattern to be recognized and the right hand side contains the C program fragment executed when that pattern is recognized. Some example generated rules are shown in Figure.4.8 and Figure.4.9.

```
{L}({L}|{D}|{DEC}|{DOLLAR})*/":" {  /* a label */
            copy(yytext, textptr);
            copy(yytext);
            strncpy(textptr, ":", 1);
            textptr++;
            return LABELID;
            delete [] yylval.sym;
         }
"//"[^\n]* {    /* one-line comment  */
            copy(yytext);
            return COMMENT;
            delete [] yylval.sym;
         }
{D}+     {       /* numbers having digits 0-9 */
            copy(yytext, textptr);
            copy(yytext);
            return INTEGER;
            delete [] yylval.sym;
         }
```

Figure 4.8: Generated Lex rules

```
          /* The operators + - * / | & << >> @ # $ % , etc */
{OP}    {
           *textptr++ = yytext[0];
           if(yytext[0]=='<'){
             SBYTE c= yyinput();
             if(c=='<')
               return SHL;
             else
               unput(c);
           }
           if(yytext[0]=='>'){
             SBYTE c= yyinput();
             if(c=='>')
               return SHR;
             else
               unput(c);
           }
           return yytext[0] ;
         }
          /* a keyword a macro name or a variable */
{L}({L}|{D}|{DEC}|{DOLLAR})*  {
          /* check for macro name */
          mac = macTbl.look(yylval.sym);
          if(mac){
            /* function to parse macro call */
            return MACRO;
          }

          /* search the variable into the keyword table */
          name = keywordTbl.look(yylval.sym, !sensitiveCase);
          if(name){
            /* return token corresponding to this keyword */
          }

          /* check for a defined variable */
          var = symbolTbl.look(yylval.sym);
          if(var){
            /* return defined variable token */
            }
          /* undefined symbol */
          else{
            return UNDEFINED;
            }
          }
```

Figure 4.9: Generated Lex rules contd ...

User_Subroutines is the section that contains supporting functions, called in the rules section. For example, the routines *copy* to copy the content of `yytext` into `yylval`, *createBuffer* to create a new buffer for `yyin`, *macParse* to parse the macro use etc. are used in the rules section and defined here.

4.2.4 Generation of the Keyword File

Keywords are assembly mnemonics which are extracted from IR while parsing *syntax* expression in Sim-nML description of the processor. We use the terms keywords and assembly mnemonics both to refer to the string obtained by parsing a *syntax* expression interchangeably. As described earlier syntax strings are parsed and corresponding tokens are generated. Characters such as space, comma etc. work as delimiter while parsing the syntax strings.

```
static struct {          // keywords
         char * name;
         int token;
} keywords[] = {
        "TEST", AS_TEST,
        "R",    AS_R,
        "JMP",  AS_JMP,
        "SUB",  AS_SUB,
        "ADD",  AS_ADD,
         0,      0,
};

initialize_keywords(){
  for(int i=0; keywords[i].name;++i)
    keywordTbl.insert(keywords[i].name, keywords[i].token);
}
```

Figure 4.10: Generated Keywords for the example processor

If the *assembler* is generated to be a case-sensitive one, the generated keywords are same as obtained from *syntax* expression. Otherwise the keywords are first converted into upper-case and then written in keyword file. e.g. if `add` is embedded in the syntax-string in the IR, the generated keyword for a case-sensitive assembler will be `add` and that for a case-insensitive one will be `ADD`.

The token names are also generated corresponding to each assembly mnemonics. A token name is generated by prefixing `AS_` to each generated keyword. This way conflicts

in the predefined names are avoided.

All these generated keywords and the corresponding tokens are installed in a hash structure *keyword table*. The hash key is calculated using the name of the keyword, and the conflicts in the hashing are resolved using chains. While parsing an assembly language file whenever Yacc recognizes a string it hashes it into *keyword table*, if it gets matched the corresponding token value is returned to the parser. The generated keywords file for the example processor description (Figure 4.2) is shown in Figure 4.10.

Chapter 5

Relocation Handling

Consider the following example of assembly language program segment for PowerPC processor.

```
.LC0: addis  9, 0, .LC0@16:16
      addis  3, 9, .LC0@16:1
      bl     printf
      .
      .
```

The following points are important. The definition of the printf routine is typically provided in the library and therefore the address of printf is not known at the time of assembly. Further, the program segment may be put at any address and therefore .LC0 value is determined only at the time of linking. Thus these addresses should be assigned typically at the time of linking or loading the program into memory. Accordingly wherever these symbols are used in the instructions, instruction operand should be adjusted at the same time.

The process of adjusting instruction operands prior to running the program is called **relocation**. Relocation requires to locate the usage of symbolic addresses within the section and adjust them so that they refer to the proper run-time addresses. The adjustment of the addresses is typically processor specific. For example, a program instruction may require 24 bits of an address to be loaded in a register (such as in Dec Alpha) where a 32 bit address is loaded in a register using two instructions - first to load 24 bits and then to load 12 bits. Similarly, in the example of the code given earlier, 16 bits of .LC0 are loaded in two separate instructions for Power PC. Since the address is not known at the time of assembly, the corresponding load instruction necessarily require adjustment (referred to as relocation now onwards) that is processor specific.

The relocation method is typically denoted by an integer called **relocation type**. Therefore the relocation types are also processor-specific (as per the ELF documentation) and vary from one processor to another. Since the values of the symbols are not known at the time of assembly, the values of the expressions involving them can not be computed. The assembler prepares a relocation table entry for each of these usage of symbolic ad-

dresses in expressions using a relocation type. The value of this expression is calculated by the linker depending on the associated relocation type.

We need a generic way to describe the *relocation type* to facilitate the automatic generation of assembler for all the processors. Keeping this in mind in our method we have designed a generic way to describe all kinds of relocations.

5.1 Generic Expressions for Assembly Program

We have defined generic expressions which involve symbolic addresses. The only operators permitted in these generic expressions are *extract*, *signed extension*, *unsigned extension*, *shift left*, *shift right*, *addition of a constant* and *subtraction of a constant*. The generic expressions are not evaluated by the *asm*. Instead the machine specific relocation types are found for these expressions as described in the next section. The syntax for operators on symbolic addresses is given below.

Extend(*symbol*@abs_exp1:abs_exp2, abs_exp3) **op1** abs_exp4 **op2** abs_exp5

Two types of *Extend* operators are supported - signed-extension and unsigned-extension. It is specified using string **SN** (for signed-extension) and **UN** (for unsigned-extension). *abs_exp3* gives the size for extension.

The *extract* operator is used to specify the extraction of a number of bits from the value of the symbol starting from some bit position. "@" is used to represent an *extract* operator. *abs_exp1* gives the size and *abs_exp2* provides the start bit position for the *extract*. In the example given above .LC0@16:16 is an expression used to extract 16 bits from the value of .LC0 starting from bit position 16.

Four arithmetic operators are supported. *Shift Left n* specifies a left shift by n bits. The operator is represented by "<<" in the assembly language program. *Shift Right n* specifies a right shift by n bits. This operator is represented by ">>" in the assembly language program. The *op1* specifies a *Shift Left* or *Shift Right* and *abs_exp4* gives the number of bits. The *Add n* specifies an addition of a constant n to the expression. It is represented by the use of "+" in the assembly program. Similarly the *Subtract n* represents a subtraction of a constant from the value of the expression and is represented using "−" in the assembly program. In the syntax string given above, the *op2* represents an *Add* or *Subtract* operator with a value *abs_exp5*.

The grammar used to parse the expression in the assembly language program which use relocatable symbols is given in Figure.5.1.

The tokens `SIGN, UNSIGN` represent the signed-extension and the unsigned-extension operators and `INTEGER` is a token that represents an integer. The tokens `SHL, SHR` represent the shift-left and shift-right operators respectively. The `expr` is the integer expression involving any arithmetic and boolean operators supported by the assembler.

Examples of a few expressions involving a symbolic address are .LC0, .LC0@16:16,

```
extract: symbol
        | symbol '@' expr ':' expr

extend: SIGN '(' extract ',' INTEGER ')'
         | UNSIGN '(' extract ',' INTEGER ')'
         | extract

opfirst: extend SHL INTEGER
        | extend SHR INTEGER
        | '(' extend SHL INTEGER ')' SHR INTEGER
        | '(' extend SHR INTEGER ')' SHL INTEGER
        | '(' extend SHL INTEGER ')' SHL INTEGER
        | '(' extend SHR INTEGER ')' SHR INTEGER
        | extend

opfinal: opfirst '+' INTEGER
        | opfirst '-' INTEGER
        | opfirst '+' '(' expr ')'
        | opfirst '-' '(' expr ')'
        | opfirst
```

Figure 5.1: Grammar used for parsing relocatable symbols

SN(.LC0@16:16, 32), .LC0 + 10 etc.

In case an operator is missed out, corresponding identity operation is assumed (for example << 0 or >> 0). If no *extract* operator is specified a default *extract* is used with the size of the expression as specified in the Sim-nML processor description for the corresponding instruction.

In addition the *asm* also supports two operators *Direct* and *Relative* to represent the direct and relative relocation types. The *asm* identifies these operations using the Sim-nML description and depending on whether the corresponding image for the binary of the instruction uses relative or direct addressing.

Since generic expressions are not evaluated we need to know what value to write in place of these expressions in the output binary generated by the *asm*. Also the address where the relocation should be applied is not known. For example, the instruction *bl* takes a 24 bit parameter. The instruction is assembled assuming the value of the expression (printf) as zero. The address in the relocation table entry where the relocation is to be performed is the start address for this instruction. While in *addis* instruction the expression is assembled with its value taken as zero and the relocation address is calculated after adding two to the start address of the instruction. The start address for these instructions corresponds to the start address at the time of assembly. The configuration file provides, both, the value to be substituted for the expression and the address to be

written in the relocation table in addition to the relocation type to be generated.

5.1.1 Configuration File

The configuration file helps in mapping generic relocation operations to machine relocation types. The configuration file also provides information like what value to substitute in place of the expression involving the symbol and what address to put in the relocation table. The format of the configuration file is fixed. All the fields in the configuration file must be separated by blanks or tabs and must be written in a specified order. One line is provided for each type of machine specific relocation. All the four field in the configuration file for a relocation type must be present. Comments can also be written in the configuration file using C or C++ type comment style. The *asm* accepts the name of this file using `-c` command-line option.

```
genericOp: opr '(' extractConfig  change  shift  arith ')'

// relocation type
opr: DIRECT
     | RELATIVE

//  extraction operator
extractConfig:  '<' INTEGER ':' INTEGER '>'

// sign and un-signed extension operator
change: // null statement
         | SIGN
         | UNSIGN

// shift operator
shift: // null statement
         | SHIFTLEFT
         | SHIFTRIGHT

// addition and subtraction operator
arith: // null statement
         | '+' INTEGER
         | '-' INTEGER
```

Figure 5.2: The grammar used to parse Generic Operations

In the configuration file, the *extend* operator is specified using one of the two strings **SN*const_int*** or **UN*const_int***. The *const_int* gives the size for extension and it represents a constant integer. SN specifies a signed-extension and UN specifies an unsigned-extension.

The *extract* operator is represented by a pattern like $< size : start\_bit >$. Here size and start_bit provide the parameters for the extraction. Both of these are represented using constant integer values.

The arithmetic operations *shift left* and *shift right* are represented by **SHL*const_int*** and **SHR*const_int*** respectively. The *add* and *subtract* operations are represented using $+const\_int$ and $-const\_int$ respectively. The grammar used to parse the generic expressions in the configuration file is shown in Figure 5.2.

The token `SHIFTLEFT, SHIFTRIGHT` represent the shift-left and shift-right operators respectively. For example, DIRECT($< 32 : 0 >$) SN32 SHR16 + 16 and RELATIVE($< 32 : 0 >$) SN32 - 32 are valid generic operations.

Any of the generic relocation operators except *extract* can be omitted while describing a relocation expression in the configuration file. The example configuration file for PowerPC 603 is shown in Figure 5.3.

```
/* generic       m/c specific  generated       address in the   */
/* expr          relocation    code in place   relocation table */
/*                             of symbol                        */

DIRECT(<32:0>)         1            0              $
DIRECT(<24:0>)         2            0              $ + 1
DIRECT(<16:0>)         4            0              $ + 2
DIRECT(<16:16>)        6            0              $ + 2
DIRECT(<14:0>)         7            0              $
RELATIVE(<24:0>)       10           0              $
RELATIVE(<14:0>)       11           0              $
RELATIVE(<32:0>)       26           0              $
```

Figure 5.3: An example configuration file for the PowerPC Processor

The symbol $ is used in the configuration file to indicate the value of current location counter (i.e. the address of the instruction). @ is used to represent the value of the symbol as known at the time of assembly of the program. For example, the first line in the example configuration file represents that whenever an expression involving a symbol is found in the assembly program which uses 32 bits of the symbol for direct addressing, relocation type is 1 and 0 is substituted for the expression while assembly. The address to emit is the address of the current instruction.

Whenever *asm* finds a symbolic address reference in assembly language program it searches for a corresponding entry in the configuration file depending on the operators applied on the symbol. If an entry gets matched it generates the relocation table using the information specified in the configuration file.

5.1.2 An example Relocation for PowerPC Processor

An Example assembly language program fragment with relocations for Power PC processor is shown below.

```
    .LC0:

    addis     9, 0, .LC0@16:16
    addis     3, 9, .LC0@16:1
    addi      3, 9, .LC0
    bl        printf
```

Figure 5.4: An example file with relocations

The generated relocation table entries are as follows.

- **.LC0@16:16**: This expression gets matched with the fourth entry (i.e. DIRECT (16:16)) and therefore the generated relocation type is 6, the instruction is assembled as addis 9,0,0 and the address in the relocation entry is 2 + the address of the current instruction.
- **.LC0@16:1**: The assembler tries to match it but the configuration file doesn't have an entry DIRECT($< 16 : 1 >$) so an error is flagged.
- **.LC0**: Since nothing is specified assembler gets its size using the information available in Sim-nML specification of the processor. The Sim-nML specification for PowerPC 603 shows that the size of argument for instruction *addi* should be 16 bits. Hence the size for symbol .LC0 is 16. The third entry of configuration file corresponds to this relocation operation and the generated relocation type is 4.
- **printf**: The symbol value is undefined. The Sim-nML specification file shows that the instruction *bl* is a relative branch instruction and it uses a 24 bit operand. In the configuration it matches with the sixth entry and therefore the generated relocation type is 10.

Let us assume that the location counter for the first instruction is 0x00000034. After removal of error line the generated assembly listing is shown in the Figure 5.5.

```
    00000034 3d200000 R     addis 9,0,.LC0@16:16
    00000038 38690000 R     addi 3,9,.LC0
    0000003c 48000001 R     bl printf
```

Figure 5.5: The generated assembly listing

In this example for the first instruction the value assembled in place of .LC0@16:16 is zero as specified in the configuration file and the address for relocation is $+2 which evaluates to 0x36. Similarly the value assembled in place of .LC0 in the second instruction is also zero and the address is $+2 (i.e. 0x3a) as specified by the third entry in the configuration file. For the last instruction the value substituted is zero while the address for relocation is $ (i.e. 3c).

```
Symbol              Address     Section    Reloc-Type

printf              0000003c    .text       10
.LC0                0000003a    .text       4
.LC0                00000036    .text       6
```

Figure 5.6: The generated relocation information

The generated relocation table is shown in Figure 5.6. The section in relocation table specifies the section in which the symbol was found.

Chapter 6

Results and Conclusion

6.1 Results

The *assembler generator(asmg)* is tested for PowerPC 603, 68HC11, 8085 and Hitachi H/8 processor models in Sim-nML. For each of these processor description an assembler is generated and verified for different assembly language programs. It is also verified that the *asmg* takes care of processor endian-ness conversion by running it on Pentium (little-endian) based Linux machines as well as on Sparc (big-endian) based machines. The assembly listing and ELF generated by the assembler have been successfully matched with those generated by the GNU assembler. In case of PowerPC 603 assembler, the output ELF file was also verified with GNU objdump.

The complete procedure to generate an assembler from Sim-nML specification is encapsulated into a shell script. When this script is run with IR of a processor as an argument, the first phase of generation of machine dependent files is executed. This phase creates some intermediate files specific to that processor. In the next phase, the assembler is generated using the machine specific generated files and the machine independent files.

For PowerPC 603 processor the input assembly language programs are generated using GNU C cross-compiler running on Pentium based Linux machines. The generated assembly language file consists of some instructions which are not part of PowerPC 603 Sim-nML specification since those instructions for PowerPC 603 are alternative names of some other instructions. For example, a generated instruction can be *blt target* which is equivalent to the instruction *bc 12, 0, target*. Similarly the instruction *mr 31, t* is equivalent to *or 31, 1, t* and the instruction *li s, d* is equivalent to *addi s, d*.

In the gcc generated assembly programs, these instructions can be edited to their equivalent instructions. As an alternative these instructions can be added in the Sim-nML model. We tried both approaches and both seems to work fine.

In case of PowerPC 603, the compiler generates a few machine specific operations in the expressions. For example *.LC*0@*ha* which is equivalent to *.LC*0@16 : 16 in our assembler. Prior to the assembly, we edit the gcc generated programs and replace all such machine specific operations to their counterparts.

6.2 Conclusion

In this thesis we have developed an *assembler generator* which takes an IR of Sim-nML basic model for a processor and generates an assembler. The generated assembler takes an assembly language program specific to that processor and generates a relocatable ELF binary object file. The assembler is generated and tested for PowerPC 603, 68HC11 and 8085 processors.

The availability of an assembler allows assembly programs to be written and tested on the functional simulator[12], even when no compiler is available.

6.3 Future Work and Extensions

Using Lex and Yacc for the assembler implies that the assembly syntax must be single token lookahead since the parsers generated by the Yacc can only look one token ahead. Also the assembler can produce relocatable object file in ELF format only. It could be useful if extended to generate other format such as COFF, a.out etc.

Appendix A

User's Manual

A.1 Assembler Generator

The generation and compilation of assembler files to get the final executable assembler(asm) is done with the help of a shell script which is available as a command name **asmg**. The assembler generator requires a processor specification in the IR form, optionally we can name the generated intermediate files. A string **CS** in command line is used to make the generated assembler case-sensitive. By default it is none case-sensitive.

A.1.1 Usage

Usage: asmg [ir-file] {yacc-file} {key-file} {lex-file} {CS}

ir-file: This is the name of the input IR file.

yacc-file: Generate the intermediate Yacc specification file in a file name *yacc-file*. The default name is *asmgYacc.y*.

key-file: Generate the intermediate keywords file in a file name *key-file*. The default name is *asmgKey.c*.

lex-file: Generate the intermediate Lex specification file in a file name *lex-file*. The default name is *asmgLex.l*.

A.2 Assembler

The assembler is used to translate an assembly language program to its relocatable binary counterpart in ELF format. The generated assembler has a command line interface that is conventional for the utilities/commands in a Unix system. If the *assembler* is run without any arguments, it displays a small help giving all the options.

A.2.1 Usage

Usage: asm {-h} {-p} {-m machine_name} {-o output_file} {-c config_file} {-l list_file} [files]

-h: This is an optional argument to print the *usage* message. If this option is specified, all other arguments are ignored.

-p: This is an optional argument. It prints the name of all supporting processors as described in the ELF documentation.

-o output_file: Generate the ELF output in a file name *output_file*. Default name is *b.out*.

-m machine_name: Specify the target machine name. This name is used in the ELF header.

-l list_file: Generate the assembly listing in *list_file*. Default is *stdout* if listing is on.

-c config_file: Use configuration file *config_file* for machine specific relocations.

files: These are the name of input assembly language files. The names are separated by spaces. These files specify exactly one source program. The source program is a concatenation of all the files in the order specified from left file name to right.

Bibliography

[1] V.Rajesh. A Generic Approach to Performance Modeling and its Application to Simulator Generator. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, July 1998. http://kshitiz.cse.iitk.ac.in/~cares/projects/simnml/.

[2] ELF Object File Format. http://www.sco.com/developer/gabi/contents.html.

[3] GNU Assembler Manual. http://www.gnu.org/manual/binutils-2.9.1/binutils.html.

[4] Rajiv A. R. Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, Dec 1999. http://kshitiz.cse.iitk.ac.in/~cares/projects/simnml/.

[5] ISDL - An Instruction Set Description Language for Retargetability. http://www.ee.princeton.edu/spam/pubs/ISDL-TR.html.

[6] ISDL - An Instruction Set Description Language. http://caa.lcs.mit.edu/caa/.

[7] D. E. Ferguson. The evolution of the meta-assembly program. *Communications of the ACM*, 9(3):190–193, 1966.

[8] SLED - Specification Language for Encoding and Decoding. http://www.eecs.harvard.edu/~nr/toolkit/.

[9] LISA - Language for Instruction Set Architectures. http://www.ert.rwth-aachen.de/lisa/lisa.html.

[10] SuperSim Processor Simulators. http://www.ert.rwth-aachen.de/lisa/supersim.html.

[11] Nihal Chand Jain. Disassembler using High Level Processor Models. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, Jan 1999. http://kshitiz.cse.iitk.ac.in/~cares/projects/simnml/.

[12] Y. Subhash Chandra. Retargetable Functional Simulator. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, June 1999. http://kshitiz.cse.iitk.ac.in/~cares/projects/simnml/.

[13] Markus Freerick. The nML Machine Description Formalism, July 1993. http://www.cs.tu-berlin.de/~mfx/dvi_docs/nml_2.dvi.gz.

[14] Common Object File Format. http://msdn.microsoft.com/library/specs/msdn_pecoff.htm.

[15] A Lexical Analyzer. http://www.gnu.org/manual/flex-2.5.4/html_chapter/flex_toc.html.

[16] Yet Another Compiler Compiler. http://www.ma.adfa.oz.au/Local/Info/bison/bison_toc.html.