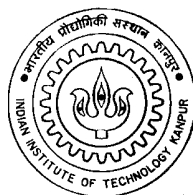


Generic Disassembler Using Processor Models

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

by

Prithvi Pal Singh Bisht



to the

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

February, 2002

Certificate

This is to certify that the work contained in the thesis entitled “ *Generic Dis-assembler Using Processor Models* ”, by *Prithvi Pal Singh Bisht*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

February, 2002

(Prof. Rajat Moona)

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

Abstract

The generic disassembler, produces symbolic relocatable disassembly of an object file in Executable and Linking Format (ELF). It uses a processor model that contains instruction set description of the processors in a language called Sim-nML. Sim-nML is simple, elegant and powerful language to express the behavior of processors at instruction level. It uses synthesized attributes to represent timing information, instruction semantics, assembly language syntax and binary representation of instructions. Generic Disassembler facilitates disassembly of programs in a GNU-compatible format. For identifying the instructions, depth first search and backtracking is used on a tree like structure of the Sim-nML instruction set description. Since the attributes of an instruction are scattered in various subtrees, syntax for the instruction is collected from the subtrees selected during traversal. Different parts of a single instruction may be matched with different subtrees. Symbolic and relocatable disassembly is achieved by using relocation and symbol information from the object file and analyzing the code to identify basic blocks.

Acknowledgments

My sincere thanks goes to my thesis supervisor Prof. Rajat Moona. His streamlined view and to-the-point approach is the backbone of this work. In addition to technical aspects, I obtained a strong sense of professionalism from him. In short, it was a rewarding experience for me to work under him.

Work done in this thesis is a part of the research done under CARES project in Cadence Research Center at IIT Kanpur. Special thanks to Dr. Sanjeev Kumar Aggarwal and Dr. Deepak Gupta for their active involvements in CARES team.

Mr Souvik Basu, senior student in CARES project, did his best in helping me begin my work in this area. I thank him for all the time he spent for me. I would like to thank Mr R. Ravindran, PhD student, for his in-time helps.

I am thankful to the MTech 2000 and 2001 batches for their significant nearness.

Special thanks goes to Pijush, Asheesh and Jayadeep for giving "in-home" and "granted" feeling back at hostel. Girish, Rajesh, Pradeep and Varada will hold separate and important places in my memory.

I am indebted to my parents, brothers and sisters for their love, care and understanding in a hard and crucial phase of life. I express my deep gratitude to my elder brother Brijesh da for his love, understanding and inspirations. Without him I would have been washed off to a different land.

Before closing this section I would like to express my deep feelings for Vinod. I will always remember him for providing the reasons to go on.

Contents

1	Introduction	1
1.1	Related Work	1
2	Sim-nML	5
2.1	Hierarchical Structure	5
2.2	Sim-nML grammar	6
2.3	Resource Usage Model	8
3	Generic Disassembler	10
3.1	Approach	10
3.2	Algorithm	10
3.2.1	Identification of Code Blocks	13
3.2.2	Generation of Symbolic Names	15
3.2.3	Disassembly	15
4	Conclusions	18
	Bibliography	19
A	User's Manual	23
A.1	IR Generation	23
A.2	Usage of Generic Disassembler	24
A.3	Configuration file	24

List of Figures

2.1	OR-Rule and AND-Rule formats	6
2.2	Addressing modes and flavor of add and branch instructions in Sim-nML	7
3.1	Approach for Tool Design with Sim-nML	11
3.2	Identifying Code Blocks	12
3.3	Algorithm for identifying object and patterns in data sections	14
3.4	Algorithm for expressing relocation information	16

Chapter 1

Introduction

With development of an Application Specific Processor (ASP), specific tools e.g. disassembler, assembler, simulator, compiler back end, etc. are also required. Such tools provide an interface for application development on this processor. Non-availability of such tools in time, may cause the whole design to fail. Problem gets complicated in case many design alternatives are to be considered. In a traditional approach, for every alternative design model, such tools are redesigned. Development of these tools is a tedious and error-prone process. CARES project group at IIT Kanpur is developing an interface that automates the process of tools development. In our approach we use processor models containing instruction set description of processors in Sim-nML. From these processor models we develop various tools using tool generators. Processor models of SPARC, Motorola 68HC11, PowerPC603, MIPS R10000, ARM, ADSP2105, 8085 and a few other processors have been developed by us and used for the generation of various tools. Generic disassembler is one of the tools in the tool chain developed at IIT Kanpur. Other developed tools are Generic Assembler Generator[4], Functional Simulator Generator[5], Retargetable Cache Simulator and Code Instrumentor[6] and Compiler back-end generator[7].

1.1 Related Work

There are various related works reported in the literature.

CBC/SIGH/SIM framework consists of a retargetable code generator CBC[8] for C compilers and the instruction set simulator SIGH/SIM[9]. This framework uses nML[3] language for processor models. nML permits concise, hierarchical processor description in a behavioral style.

Aviv[11] is a retargetable code generator which produces the optimized machine code for target processors with various instruction set architectures. It uses Instruction Set Description Language (ISDL[10]) as a high level language for describing the processor models.

Cycle accurate models of pipelined processor architectures require a pipeline-accurate behavioral description. In nML or ISDL, the semantics of the language do not provide means for describing the cycle accurate models.

EXPRESSION[12] language uses a mixed behavioral and structural representation approach for processor modeling. SIMPRESS[13], based on EXPRESSION, is a retargetable simulator generator for processor-memory architectures and EXPRESS is a retargetable compiler for embedded system-on-chip systems.

New Jersey Machine-Code toolkit[15] helps programmers write applications that process machine code e.g. assemblers, disassemblers, code generators, tracers, profilers, and debuggers. It uses processor models written in Specification Language for Encoding and Decoding (SLED[14]). SLED describes abstract, binary and assembly language representations of machine instructions. With the help of toolkit retargetable debugger, retargetable optimizing linker and disassembler for SPARC have been developed.

BUILDBONG (Building Special Computer Architectures based on Architecture and Compiler Co-Generation[16]) discusses the automatic generation of instruction set simulators and the corresponding retargeted compilers. The methodology is based on ASMs (Abstract State Machines) as the formal model for describing a processor's behavior. ASM is a mathematical model of computation based on the concept of universes, functions and updates of functions. BUILDBONG project group[31] aims at architecture synthesis and compiler generation for the architectures. XASM[18] is ASM specification language. Gem-Mex[19] tool automatically generate a debugging and simulation environment for a given XASM specification.

MIMOLA (Machine Independent Microprogramming Language[20]) is a structure level description language. RECORD[21] system based on MIMOLA aims at automatic code generation for fixed point DSPs with a fixed instruction word length.

The PlayDoh[22] architecture of HP Laboratories is based on machine descriptions specified in a high level textual language MDES[23]. It aims at developing performance oriented compilers for the VLIW and super-scalar processors.

LISA[24] processor modeling language facilitates pipeline description and explicit control specification. Lisa Processor Design Platform (LPDP[32]) tool-suite uses processor models in LISA to generate software development tools including C-compiler, assembler, linker, instruction set simulator and debugger front-end. RADL[25], another specification language is derived from LISA, focuses on detailed pipeline behavior and is used to generate instruction set simulators.

CHESS/CHECKERS[26] environment consists of a C compiler called Chess, a linker called Bridge, an instruction set simulator called Checkers and an assembler and disassembler called Darts. Processor models are described using nML[3]. Chess reads the nML description to generate binary code from a C program. Similarly, Checkers uses the nML description to accept the binary code and simulate its execution on the target processor. Chess/Checkers is a retargetable environment.

The FlexWare[27] framework consists of a retargetable code generator CODESYN and an instruction set simulator INSULIN. CODESYN takes one or more algorithms expressed in a high-level language and maps them onto a user defined instruction set to produce optimized machine code. It can be used to develop code for processors including Application Specific Processors (ASPs). INSULIN is based on a reconfigurable VHDL model of a generic instruction set processor.

CASTLE[28] is a co-design platform which provides a number of design tools for configuring application specific design flows. The design flow starts with a C/C++ program and gradually derives a register-transfer level description of a processor hardware, as well as the corresponding compiler for generating the processor opcode.

Visualization Based Micro-architecture Workbench (VMW[29]) facilitates specification of micro-architecture of processors and automatic generation of performance simulators. It defines a set of machine description files. These machine description

files can be compiled into a working performance simulator for a specific target processor. Visualization capabilities are incorporated to allow monitoring of simulation process. With the help of VMW performance simulators for PowerPC 601 and 620, DEC Alpha AXP 21064 and 21164 and IBM RS/6000 microprocessors have been generated and executed.

Chapter 2

Sim-nML

Sim-nML is an extensible formalism designed to specify generic single processor models. It is a language used to describe the instruction set architecture of a processor with the minimal knowledge about its micro-architecture. The design of Sim-nML is highly influenced by that of the nML[3].

The processor models in Sim-nML are described using attribute grammar¹ in a hierarchical manner. To facilitate this, Sim-nML defines two kind of primitive rules, namely *op-rules* and *mode-rules*. Op rules are generally used for describing the instructions while mode rules are used for describing the addressing modes.

The Sim-nML description given in figure 2.2 describes a simple processor with four type of instructions - add, branch, load and store. In a pipelined processor, several instructions may coexist at the same time. In such a processor, a single register such as PC cannot specify the address of an instruction in flight. Sim-nML supports a special token, \$, which is used to denote the memory address of the instruction in the definition of various attributes of an instruction.

2.1 Hierarchical Structure

In Sim-nML based descriptions, the instruction set is described in a hierarchical manner with fragments of each of the attribute being distributed over the whole

¹An attribute grammar is a context free grammar in which each non-terminal have a fixed set of attributes and for each production a set of semantic rules is given.

Sample OR Rule

op $n_0 = n_1 \mid n_2 \mid n_3 \mid n_4$

Sample AND Rule

op $n_1 (p_1 : t_1, p_2 : t_2, p_3 : t_3, \dots)$

$a_1 = e_1, a_2 = e_2, a_3 = e_3, \dots$

where each n_i is a non-terminal, each t_i a token.

Each a_i is an attribute name and e_i their respective definitions.

Figure 2.1: OR-Rule and AND-Rule formats

specification tree. The common behavior of a class of instructions is captured at the top level of the tree. The specialized behavior of the instructions is captured in the subsequent lower levels.

For example in figure 2.2, two instructions, represented by *addRtoR* and *addItoR*, have a common part of the syntax as "*ADD*". Similarly these instructions share a common part of the image as "*00000001*". This common behavior of both the instructions is captured by specification tree node *addinst*, the ancestor of *addRtoR* and *addItoR* nodes.

2.2 Sim-nML grammar

The root of the specification tree in the Sim-nML is represented by a fixed symbol called *instruction*. There are two type of constructions supported in the Sim-nML namely *OR-Rule* and *AND-Rule*, (figure 2.1). The and-rule constructions are used for the terminal symbol definition and represent the leaf nodes in the specification tree. The or-rules are non terminals which can expand to further and-rules or or-rules or both. Both primitive rules, i.e. *mode* and *op* rules can be constructed using *Or* or *And* constructions. The Sim-nML grammar predefines four attributes - *syntax*, *image*, *action* and *uses*. The syntax attribute describes the assembly language format of the instruction while the image attribute describes the binary coding of the corresponding instruction. Similarly, action attribute describes the

```

type word = card(16)
type byte = card(8)
reg R[4,word]
reg PC[1,word]
mem M[2**16,byte]
resource ifu, bu, alu[2], wb

// Addressing modes

mode immediate(x:word)= x
  syntax = format("%d",x)
  image = format("%16b",x)
mode register( i : card ( 2 ) )=R[i]
  syntax = format("R%d",i)
  image = format("0%2b",i)
mode direct(addr:word)=M[addr]:M[addr+1]
  uses = if "rand"() < 0.95 then  #{1}
        else  #{10}
        endif
  syntax = format("%d",addr)
  image = format("%16b",addr)
mode reg_indirect( i : card ( 2 ) ) =
  M[R[i]]:M[R[i]+1]
  uses = if "rand"() < 0.95 then  #{1}
        else  #{10}
        endif
  syntax = format("(R%d)",i)
  image = format("1%2b",i)
op instruction(x:inst_type)
  uses = x.uses
  syntax = x.syntax
  image = x.image
  action = { x.action ; }
op inst_type = add_inst | branch | load_store
op add_inst(x:addr_type)
  syntax = format("ADD %s",x.syntax)
  image = format("00000001%s",x.image)
  uses = x.uses
  action = { x.action ; }
op addr_type = addRToR | addIToR
op addRToR(R2:register,R3:register)
  uses = ifu#{1}, alu#{1}, wb#{1}
  syntax = format("%s,%s",R2.syntax,R3.syntax)
  image = format("00%s%s",R2.image,R3.image)
  action = {
    R3 = R3 + R2;
    PC = PC + 2;
  }
op addIToR(x:immediate,R:register)
  uses = ifu#{2}, alu#{1}, wb#{1}
  syntax = format("%s,%s",R.syntax,x.syntax)
  image = format("01%s%s",R.image,x.image)
  action = {
    R = R + x;
    PC = PC + 4;
  }
op branch(x:branchtype)
  uses = ifu#{2}, alu#{1}, bu#{1}, wb#{1}
  syntax = format("JMP %s",x.syntax)
  image = format("00000010%s",x.image)
  action = { x.action ; }

```

```

op branchtype = branchrelative | branchabsolute
op branchrelative( target : card ( 16 ) )
  syntax = format("%d", target)
  image = format("00000001%16b", $+target)
  action = {
    PC = target;
  }
op branchabsolute( target : card ( 16 ) )
  syntax = format("%d",target)
  image = format("00000010%16b",target)
  action={
    PC = target;
  }
op load_store = load | storevar tmp[1,word]
op load(r:register,l:loadmode)
  uses = l.uses
  syntax = format("LOAD %s,%s",r.syntax,l.syntax)
  image = format("00000011%s%s",r.image,l.image)
  action= {
    l.action;
    r = tmp;
  }
op loadmode = load_direct | load_indirect
op load_direct(m:direct)
  uses = ifu#{2}, m.uses, wb#{1}
  syntax = format("%s",m.syntax)
  image = format("01000%16b",m.image)
  action = {
    PC = PC + 4;
    tmp = m;
  }
op load_indirect(r:reg_indirect)
  uses = ifu#{1}, r.uses, wb#{1}
  syntax = format("%s",r.syntax)
  image = format("00%s",r.image)
  action = {
    PC = PC + 2;
    tmp = r;
  }
op store(r:register,s:storemode)
  uses = s.uses, wb#{2}
  syntax = format("STORE %s,%s",r.syntax,s.syntax)
  image = format("00000100",r.image,s.image)
  action = {
    s.action;
    M[tmp] = r<7..0>;
    M[tmp+1] = r<15..8>;
  }
op storemode = store_direct | store_indirect
op store_direct(m:immediate)
  uses = ifu#{2}, wb#{1}
  syntax = format("%s",m.syntax)
  image = format("00001%s",m.image)
  action = {
    PC = PC + 4;
    tmp = m;
  }
op store_indirect(r:reg_indirect)
  uses = ifu#{1}, r.uses, wb#{1}
  syntax = format("%s",r.syntax)
  image = format("00010%s",r.image)
  action = {
    PC = PC + 2;
    tmp = r;
  }

```

Figure 2.2: Addressing modes and flavor of add and branch instructions in Sim-nML

semantics of the action and the *uses* attribute describes the resource usage model.

2.3 Resource Usage Model

The micro-architecture details of the processor can be specified using the resource usage model. In Sim-nML specification, the entities within the processor such as functional units, ALUs, pipeline stages, registers, ports etc., constitute a set of resources. Every instruction utilizes several resources each for an amount of time which may be constant or dependent upon only the instruction. The resource usage model is based on the assumption that at any instant, an instruction in execution holds a set of resources and performs certain action. The next set of resources must be acquired before performing the corresponding action. The resources held by the instruction and the action taken change progressively.

In this model instructions contend for resources and wait, if the resources are not available. Thus the flow of the instructions in the micro-architecture is essentially a way of resolving resource conflicts. When two instructions wait simultaneously for a single resource, the conflict is resolved in FIFO order of the instructions. The *uses* attribute in Sim-nML descriptions describes the resource usage model for an instruction.

An example model for a super-scalar processor is shown in figure 2.2. All instructions are read by a resource IFU (Instruction Fetch Unit). The instructions have variable length (16 bits or 32 bits) and accordingly take one or two units of time at the IFU. In addition there is a BU (Branch Unit), a resource that is used by the branch instructions only. There are two ALUs, both alike, and are used by the ADD instruction (and other ALU instructions) and by the branch instructions. Finally there is a resource called WB (Write Back) unit. It is used by the instructions that need to write their results back into the registers.

In some cases, the resource uses depends on a condition. For example, time for memory access depends on whether it was a hit in the cache or not. The conditional use can be specified by the *if* construction of *uses* attribute. For example, construction *if "rand"() < 0.95 #\{1\} else #\{10\}* represents that in case of a cache hit (hit

ratio is 0.95), delay of 1 time unit occurs while delay is of 10 time units in case of a miss. This conditional construction also shows that in Sim-nML models, it is possible to specify components of uses attribute without any resource to represent just the time delay.

In Sim-nML models, the architectural changes are relatively easy to incorporate. For example a three way super-scalar processor with 3 ALU units can be obtained by appropriately changing the resource declaration in figure 2.2.

Chapter 3

Generic Disassembler

3.1 Approach

The approach for the generic disassembler is similar to the other tools based on Sim-nML processor models (figure 3.1). Sim-nML processor models are first converted to a compact *IR* (*Intermediate Representation*) by a tool *IRG* (*IR-Generator*). This IR along with the other required inputs is used by different tools.

IR captures all the information in the Sim-nML description and makes it easy to access the information by various tools. In our approach, IR is a collection of tables each of fixed size records. A ‘table of index’ in the beginning of the IR file provides the size and locations of other tables in the file. The generic disassembler uses a simple configuration file in addition to the IR to convert an ELF binary to corresponding assembly language program.

3.2 Algorithm

While disassembling an object file, the object file is first read in the memory. The process of disassembly for relocatable and executable object file differs only in the aspect that executable file contains absolute addresses while the relocatable file contains the relocatable addresses. This difference is handled while considering the start addresses of the various sections in the object file.

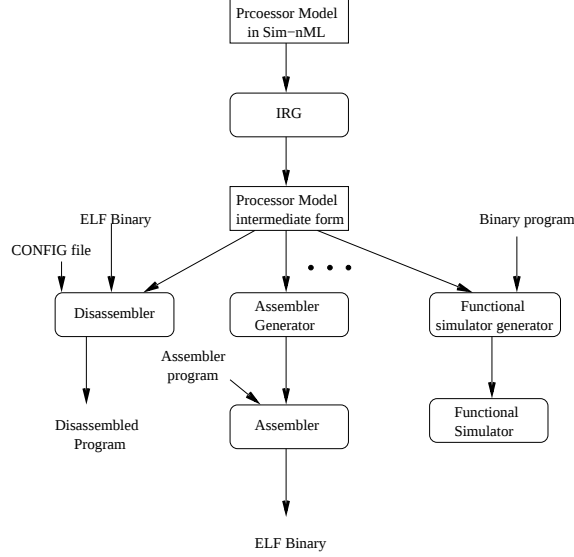


Figure 3.1: Approach for Tool Design with Sim-nML

In order to disassemble an instruction, the bit pattern from the object file is used to identify the assembly language instruction. Algorithm for identifying an instruction is based on a combination of depth first search and backtracking. Starting from the root node *instruction* in the Sim-nML specification tree, images of different nodes are matched with the given input bit string in depth first order. Since the instruction specifications may be distributed over the specification tree, nodes in a path from the root to the leaf contribute in defining an instruction. The image formed by collecting the images of all such nodes in a path should match in order to identify the instruction. Once such a path is identified, syntax attribute of all nodes in the path are used to get the syntax of the identified instruction. The match is carried out using DFS and when it fails, backtracking is used to search in another path.

The disassembler also analyzes the object file in various ways to make the disassembler output readable. In particular the disassembler uses the symbol information from the ELF file and also generates symbols whenever needed. These symbols are used in place of addresses in the instructions. The process of disassembly is divided

```

1 from the binary ELF, extract the addresses of various functions;
2 put these addresses in start_code_block list;
3 i = 0;
4 while start_code_block list is not empty {
    a. // take out the address from the list.
      address = listout(start_code_block);
    b. for j = 0 to i-1 { // if address is already traced, ignore it.
        if address  $\geq$  code_block_start[j] &&
        address  $\leq$  code_block_end[j] {
            goto step 4;
        }
    }
    c. code_block_start[i] = address;
    d. follow binary till an instruction that changes PC;
    e. if the instruction is unconditional branch {
        code_block_end[i] = address of the current instruction;
        append the target address in the start_code_block list;
        i = i + 1;
        goto step 4;
    }
    f. if the instruction is unconditional return {
        code_block_end[i] = address of the current instruction;
        i = i + 1;
        goto step 4;
    }
    g. if the instruction is some other PC changing instruction {
        append the target address in the start_code_block list;
        goto step 4.d;
    }
}
5 merge the adjacent code blocks;

```

Figure 3.2: Identifying Code Blocks

into following steps.

3.2.1 Identification of Code Blocks

The first kind of analysis on the object file is to find out the code and data area. The generic disassembler first finds out the code blocks (see the algorithm given in figure 3.2). The algorithm works as follows. From the ELF binary, addresses of various functions in the program are read, if available. This information is stored in the symbol table section of the binary object file. Assuming that each of these will be starting/entry point of a code block, the code is traced till a branch unconditional or return unconditional instruction is found. Branch unconditional or return unconditional instructions are used to denote the end of a code block. While tracing the code blocks, if a call or conditional branch instruction is encountered, the target address of the instruction defines another code block. If the entry point for the target address is already traced, tracing is continued with other untraced starting points. The tracing process is repeated as long as an untraced starting point is there. Call instructions may refer to the system library functions. Tracing is not done for entry points corresponding to such functions. With the help of symbol table of the ELF binary such starting points are identified and marked as untraceable. At the end of code block analysis, all the adjacent code blocks are merged into one.

For this analysis, the disassembler needs to know the type of the instruction (branch vs. non-branch). The instruction type is determined with the help of a configuration file. Instructions can be one of the four types - branch, call, return and simple. The configuration file contains entries for the first three instruction types. Following line is taken from a sample configuration file. It specifies that the subtree rooted at node with label *branch_uncond* contains all the branch unconditional instructions.

```
% BRANCH_UNCOND branch_uncond
```

After the code block analysis, data analysis is done. Data sections consist of data objects defined in the program, strings etc. used in the program. Symbol table of the ELF binary contains information about the objects in data sections, if present. We need to identify the beginning of objects in data sections so that

```

data_section = contents of the data section from the ELF binary;
offset = 0;
while offset is not equal to the length of data section {
    if an object is found in the symbol table of the binary ELF for this offset {
        get the size/object_name/alignment information from the symbol table;
        offset = offset + size_of_object;
    }
    if an object is not found in the symbol table {
        if data_section[offset] is a non-printable character {
            use pseudo op .byte for this offset;
            offset = offset + 1;
        }
        if data_section[offset] is a printable character {
            find length of the string of printable characters starting at offset;
            if the identified string is NULL terminated
                use pseudo op .asciz for the identified string;
            if the identified string is not NULL terminated
                use pseudo op .ascii for the identified string;
            offset = offset + length_of_string ;
        }
    }
}

```

Figure 3.3: Algorithm for identifying object and patterns in data sections

appropriate information (size, scope, label etc.) can be written with data contents of the data objects. Algorithm used for identifying objects and patterns is given in figure 3.3. It works as follows. For every offset of the data section, a search is made in the symbol table, to check if it is the beginning of an object. If an object is found, symbol table is used to get the object scope (global/local), label and size information. In case no object is found, depending upon the access pattern of the data it is defined as follows. Array of bytes (*.byte* pseudo op), if data is a sequence of non-printable characters or a string of printable characters (*.ascii/.asciz* pseudo op). If the string of printable characters terminate with a NULL, *.asciz* pseudo op is used otherwise *.ascii* pseudo op is used. Offset is increased by the size of objects identified in symbol table or the patterns identified by the disassembler internally.

3.2.2 Generation of Symbolic Names

In the second part of object file analysis, an association is made between the unique symbolic names and the addresses of the instructions referred to by other instructions. The generic disassembler generates symbolic names for the starting addresses of new code blocks, if the corresponding symbolic name is not already available in the symbol table of the ELF binary. These generated symbolic names are entered in a symbol table maintained by the generic disassembler. The symbol table is initialized with the symbol table information read from the ELF binary.

The symbol table thus generated is used in the process of disassembly as described in the next section. In our approach, the unique symbols are generated as .L0, .L1, .L2 etc. (i.e. by prefixing a string ".L" to a counter value).

3.2.3 Disassembly

The final part of the object code analysis consists of generation of the assembly language program from the ELF binary. At this point symbolic names and code block information is available. This step starts with writing general information in the output such as the source file name from which object file has been generated (*.file* pseudo op).

```

for i = 0 to total_reloc_entries - 1 {
    if reloc_table[i].offset is equal to target_address {
        relocated_symbol = reloc_table[i].symbol ;
        relocation_type = reloc_table[i].type;
        if operator is determined for relocation_type
            write relocation information with operator and relocated_symbol;
        if operator is not determined for relocation_type
            write @Reloc(relocated_symbol,relocation_type);
    }
}

```

Figure 3.4: Algorithm for expressing relocation information

Labels are written for the first instruction of each code block, in the beginning of its disassembly. The symbolic label for the address is obtained from the symbol table maintained by the generic disassembler. If the instruction is the first instruction of a function then corresponding to its label, other pseudo instructions such as *.type*, *.align*, *.size*, *.globl* or *.local* are also written.

After generation of the label, the instruction is disassembled as per the algorithm given earlier. The disassembled instruction may contain numeric addresses. In our approach, we convert these numeric addresses to the symbols. Symbolic names corresponding to numeric addresses are found in the symbol table maintained by the generic disassembler. The target address of an instruction might be relocatable. If the target address is found in the relocation table, algorithm given in figure 3.4 is used to express the relocation information. Relocation table entry for the target address determines the relocation type and the relocated symbol. Relocation type is specific to a processor. Target processors can be uniquely identified with the information present in the header structure of the ELF binary. The relocation information is expressed using an operator on the relocated symbol. For example, with relocation type *R_PPC_ADD16_HA* and relocated symbol *.L0* for PowerPC603, the information is written as *.L0@ha* where *@ha* is the operator on the symbol *.L0*. If the corresponding operator cannot be determined for

a relocation type, disassembler writes relocation information in the generic format *@Reloc(relocated_symbol, relocation type)*.

Finally, the contents of data sections are written to the output. In the earlier steps information like objects, patterns etc. is already collected. For each object found in the symbol table, information such as its section, alignment, size, scope, label etc. is written before the contents, using appropriate pseudo-ops. This information is taken from the symbol table of the binary ELF. For the patterns identified by the generic disassembler, pseudo ops such as *.byte/.asciz/.ascii* are written along with the contents.

Chapter 4

Conclusions

The generic disassembler is tested for PowerPC603, Motorola 68HC11 and MIPS R10000 processor models. Output produced by the generic disassembler is compatible to the GNU assemblers and semantically similar to the output produced by the GNU C compilers. The output differs from the gcc-compiled assembly program in the names of the labels. In addition, depending upon the specifications, alternate instructions are generated by the generic disassembler. For example, in place of the instruction *mflr 0* produced by the GNU C compiler for PowerPC603, the generic disassembler produces a semantically equivalent instruction *mfpr 0,256*. Similarly, instructions in the pairs such as *mfspir 0,256* and *mflr 0; ori 0,0,0* and *nop; addi 0,0,0* and *li 0,0; or 31,1,1* and *mr 31,1* etc. are semantically equivalent and have the same machine coding.

In the output produced by the native compiler, redundant labels are generated that are not referred to by any instruction. All such redundant labels are not present in the disassembly produced by the generic disassembler. Since the generic disassembler generates the labels by code analysis, these labels are different in the compiler generated assembly program. The labels are however positioned at the same location. Therefore, though the labels are different, effect is the same.

Bibliography

- [1] Executable and Linking Format (ELF). *Tools Interface Standard (TIS), Portable Formats Specifications*. Unix System Laboratories, Version1.1.
- [2] V. Rajesh and R. Moona. *Processor modeling for Software Hardware co-design*. In Proc. of Int. Conf. on VLSI Design, pp. 132-137, Jan. 1999, Goa, India.
- [3] M. Freerick. *The nML Machine Description Formalism*. Tech. Rep. 1991/15, TU Berlin, Fachbereich Informatik, 1991, Berlin.
- [4] Sarika Kumari. *Generation of Assemblers using High Level Processor Models*. MTech Thesis, Department of CSE, Indian Institute of Technology Kanpur, Feb. 2000, Kanpur
- [5] Subhash Chandra and Rajat Moona. *Retargetable Functional Simulator Using High Level Processor Models*. In Proc. of 13th Int. Conf. on VLSI Design, pp. 424-429, Jan. 3-7, 2000, Calcutta, India.
- [6] Rajiv A. R. and R. Moona. *Retargetable Cache Simulation Using High Level Processor Models*. In Proc. of 6th Australasian Computer Syst. Architecture Conf. 2001, pp. 114-121, Jan. 29-30, 2001, GoldCoast, Australia
- [7] Shishir Mondal. *Compiler Back End Generation from nML Machine Description*. MTech Thesis, Department of CSE, Indian Institute of Technology Kanpur, Jun. 1999, Kanpur.
- [8] A. Fauth, A. Knoll. *Automated Generation of DSP Program Development Tools Using a Machine Description Formalism*. In Proc. IEEE ICASSP-93, pp. 457-460, Apr., 1993, Minneapolis.

- [9] F. Lohr, A. Fauth, M. Freericks *SIGH/SIM - An Environment for Retargetable Instruction Set Simulation*. Tech. Rep. 1993/43, TU Berlin, Fachbereich Informatik, Berlin, 1993.
- [10] G. Hadjiyiannis, S. Hanono, and S. Devadas. *ISDL: An Instruction Set Description Language for Retargetability*. in Proc. of the 34th Design Automation Conf., pp. 299-302, Jun. 9-13, 1997, Anaheim, California, USA.
- [11] Silvina Zimi Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD Thesis, Department of EECS, MIT, Jun. 1999, USA.
- [12] A. Halambi, P. Grun, et al. *EXPRESSION: A Language for architecture exploration through compiler/simulator retargetability*. In Proc. of the European Conf. on Design, Automation and Test (DATE), pp. 485-490, Mar. 9-12, 1999, Munich, Germany.
- [13] Asheesh Khare. *SIMPRESS: A Simulator Generation Environment for System-On-Chip Exploration*. MS Thesis, Department of Information and Computer Science, University of California Irvine, 1999, Irvine.
- [14] Norman Ramsey and Mary F. Fernandez. *Specifying representation of machine instructions*. ACM Trans. Program. Lang. Syst., volume 19, number 3, pp. 492-524, Jan. 1997.
- [15] Norman Ramsey and Mary F. Fernandez. *The New Jersey Machine-Code Toolkit*. In Proc. of the Usenix Technical Conf. 1995, 1995, pp. 289-301, Jan. 16-20, 1995, New Orleans, Louisiana.
- [16] J. Teich, R. Weper, D. Fischer, and S. Trinkert. *BUILDABONG: A Rapid Prototyping Environment for ASIPs*. In Proc. DSP-Deutschland, pp. 153-162, Oct. 2000, Munich, Germany.
- [17] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide* Specification and Validation Methods, Oxford University Press, pp. 9-36, 1995.

- [18] M. Anlauff. *XASM: An Extensible, component-based abstract state machine language*. In Proc. of Int. Workshop on Abstract State Machines ASM 2000, Lecture Notes on Computer Science, Volume 1912, pp. 69-90, Mar. 19-24, 2000, Springer.
- [19] M. Anlauff, P. Kutter, and A. Pierantonio. *Formal aspects of and development environments for Montages*. In 2nd Int. Workshop on the Theory and Practice of Algebraic Specifications, Workshops in Computing, Sep. 25-26, 1997, Springer-Verlag, Amsterdam, The Netherlands.
- [20] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Nenmann, and D. Vogenaner. *The MIMOLA Language - Version 4.1*. Tech. Rep., Lehrstuhl Informatik XII, University of Dortmund, Sep., 1994.
- [21] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Jun., 1997, First Edition, Kluwer Academic Publishers, Dordrecht, Netherlands
- [22] V. Kathail, M. Schlansker, and B. Rau. *HPL PlayDoh Architecture Specification: Version 1.0*. Tech. Rep. HPL-93-80, HP laboratories, Mar. 1994.
- [23] The MDES User Manual http://www.trimaran.org/docs/mdes_manual.pdf. Trimaran Release, 1998.
- [24] V. Zivojnovic, S. Pees, and H. Meyr. *LISA: machine description language and generic machine model for HW/SW co-design*. In Proc. of IEEE Workshop on VLSI Signal Processing, pp. 127-136, Oct., 1996, San Francisco, California.
- [25] C. Siska. *A Processor Description Language Supporting Retargetable Multi-Pipeline DSP program Development Tools*. In Proc. on 11th Int. Symposium on Syst. Synthesis, pp. 31-36, Dec. 2-4, 1998, Taiwan, China.
- [26] D. Lanneer, J. Van Praet, A. Kiffi, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. *Chess: Retargetable Code Generation for embedded DSP processors*. In P. Marwedel, G. Goossens *Code Generation for Embedded Processors*, pp. 85-102, Kluwer Academic Publishers, 1995.

- [27] P. Paulin, C. Liem, T. May, and S. Sutarwala. *FlexWare: A Flexible firmware development environment for embedded systems*. In P. Marwedel, G. Goossens, *Code Generation for Embedded Processors*, pp. 67-84, Kluwer Academic Publishers, 1995.
- [28] M. Theissinger, P. Stravers, and H. Veit. *CASTLE: An Interactive Environment for HW-SW Co-Design*. In Proc. of the 3rd Int. Workshop on Hardware-Software Co-design, pp. 203-209, Sep. 5-9, 1994, Grenoble, France.
- [29] Trung A. D. and John Paul S. *VMW: A Visualization based Micro-architecture Workbench*. IEEE Computer, Volume 28, Number 12, pp. 57-64, Dec. 1995.
- [30] Rajat Moona. *Processor Models for Retargetable Tools*. In Proc. of 11th IEEE Int. Workshop on Rapid Systems Prototyping (RSP 2000), pp. 34-39, Jun. 21-23, 2000, Paris, France.
- [31] The BUILDABONG Project.
<http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html>.
- [32] Lisa Processor Design Platform.
<http://klaus.ert.rwth-aachen.de/lisa/lpdp.html>.

Appendix A

User's Manual

Sim-nML specification of the target processor is first changed to Intermediate Representation with the help of IR-Generator tool.

A.1 IR Generation

IR-Generator tool is available with the source code of disassembler tool. Usage of IR-Generator are as follows

Use : `irg [-d level] [-h] [-w] -o ir_file Sim-nML_file`

-d : To get debug information in debug.tmp at different levels (1..4) of detail

Value 1 means minimum detailed information and 4 means maximum detailed information

-h : to get this message

-o : Intermediate code will be in file ir_file otherwise default file name is IR

-w : to get warning messages. Default is no warning.

Sim-nML_file : input file having Sim-nML specification of target processor

A.2 Usage of Generic Disassembler

Disassembler tool is compiled with *make* in the source directory. Usage of the disassembler tool are as follows

Use : `Disassembler_exe [-h] [-o output] -i ir_file -c config objfile`

-h : to get this help message

-o : Disassembler writes the disassembly to the file *output*.

If the output file name is not specified disassembler appends `_symdis.s` to object file name to produce the output file name. e.g. if object file name is `8q.o` and output file has not been specified, output is written to file `8q_symdis.s`.

-i : *ir_file* is generated with the Sim-nML specification of the target processor.

-c : *config* is the configuration file for processor specification in *ir_file*. This file contains information specific to Sim-nML processor specification. It helps in determining the instruction type and identifying the immediate modes.

obj_file : It is the ELF object file to be disassembled.

Disassembler_exe is the executable file for the disassembler tool. With the help of makefile generator script *genmake*, available with disassembler tool, tool name can be configured according to the target processor e.g. for PowerPC603, executable disassembler tool file is named *ppcdisa*.

For *Elf file*, native compiler for the target processor is required. Object file is produced with the help of compiler, using appropriate compilation flags.

A.3 Configuration file

Sim-nML specification is again used for producing the configuration file. Sample *Configuration file* for PowerPC603 Sim-nML description is given below.

%IMM_MODE IMM16, IMM24, SIMM, UIMM16

%BRANCH_UNCOND branch_uncond

%BRANCH_COND branch_cond

```
%CALL_UNCOND call_uncond
%CALL_COND call_cond
%RETURN_UNCOND bran_cond_lr
%RETURN_COND ret_cond
```

Here *branch_uncond* is the label of the subtree which contains all the branch unconditional instructions. All the labels should be written, comma separated, in case instructions for one category are in different subtrees. For writing a new configuration file, hierarchy of the Sim-nML processor specification should be analyzed. Labels of all the subtrees, which contain instructions for a particular type should be written, comma separated, in front of that instruction type. One can also write the opcodes of instructions in place of subtree labels.