THE DESIGN AND IMPLEMENTATION

OF A C COMPILER FOR SAFA

GAO YUGUANG

(B. Sci., Shanghai Jiao Tong University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. Especially, I am deeply indebted to my supervisor Professor Yuen Chung Kwong, whose help, stimulating suggestions, guidance, knowledge and encouragement helped me in all the time of research for and writing of the thesis.

I would like to thank Dr. Soo Yuen Jien, who gave me much knowledge and advice on SAFA environment and operation mechanism. I have furthermore to thank my friend, Mr. Ji Yong, who gave me much help on the compiler related knowledge. I want to show my thanks to Dr. Wang Haichen, Mr. Cheng Wenyuan, Ms. Wang Xiaoxue, Mr. Gao Yufeng and Mr. Sun Jialei for all their help, support and valuable hints and suggestions to my work.

My thanks also go to all the people who gave me help during the study. Grateful thanks should also give to School of Computing and National University of Singapore.

Finally, I am deeply grateful to my parents. They kept giving me their constant love, understanding, support and encouragement.

Table of Contents

Summary

SAFA (Stack And Frame Architecture) is designed aiming to overcome some of the disadvantages of a stack based architecture, e.g. array manipulation support. SAFA program is composed of stack and frame manipulation instructions. The thesis concerns the design and implementation of a C to SAFA compiler to meet the need to execute C on SAFA.

The design of C to SAFA compiler is the primary part of the thesis. We researched the most significant differences between C to SAFA compiler and common C compilers to develop a framework for compilation, and gave solutions to various specific issues.

In the thesis, implementation of the compiler is also described. Working most importantly for the compiler, where the greatest difference, compared to common C compilers, lies in the implementation of the intermediate code generation. The intermediate code generation and target code generation are implemented according to the definition of SAFA instruction set, which is composed of several self contained sections, and each section can be loaded to the memory to be executed. Code samples and performance data are also done in the thesis.

List of Figures

Chapter 1

Introduction

1.1 Stack Architecture and Stack and Frame Architecture

1.1.1 Stack Architecture

Hardware supported Last In First Out (LIFO) stacks have been used on computers since the late 1950's [1]. During the 1980's, stack architecture was one of the most popular alternative computer architectures to accumulator architecture and general purpose register architecture. The addressing and storing of the operands is the main differentiating feature for these architectures. For stack machine, operands are implicitly on top of the stack, in accumulator architecture one operand is implicitly the accumulator, i.e. stored in the accumulator, and general purpose register architectures have only explicit operands, either registers or memory locations, referred to register numbers or memory addresses respectively.

The main strength of the stack machine can be summarized as below:

- A basic and natural tool that is used for processing well-structured code
- Can execute applications requiring stacks (like expression evaluation, method/ function invocation, parameters passing in subroutine, etc.) much faster than other architectures
- Compiler written for these machines tends to be simpler and more efficient
- More compact binary code size

A number of well known stack machines were designed and received moderate market success, e.g. the Burroughs family, Eclipse, HP3000, ICL2900, CRISP, Dragon, etc [3].

However, the success of stack machine was quite short lived. The market was virtually flooded with variants of general purpose register architectures like Intel 80x86, SPARC, MIPS, PowerPC, Alpha, etc in the 1990's. Although there are some differences between these architectures, it is clear that the stack architecture is not employed within them. The main reasons for the downfall of the stack architecture lie in the inherent weakness of the design:

- Super-scalar execution techniques like pipelining, out-of-order execution can not be applied, because the lack of instructional level parallelism in a stack program. This severely limits the execution speed of a stack machine.
- Poor support for indexing memory access, e.g. element access in an array, and records in files. These operations are frequently used in most high-level programming languages. Cumbersome, inefficient supports for these operations seriously handicap the stack architecture.

1.1.2 Stack and Frame Architecture

Stack and Frame Architecture (SAFA), which is devised by Yuen [2], and simulated by Soo [3], as a stack machine architecture that can avoid some of the disadvantages, have three major features [3]:

- Hardware stack structure that uses reservation stations and reorder buffer to support instruction level parallelism
- Instructions as well as hardware support for stack and data frames, used for procedure and function, entrance and exit, variables scoping and accessing.
- Improved array support for high level programming

The following diagram (Figure 1-1) shows the hardware structure of a SAFA CPU. However, the hardware details are peripheral to the issues of compiling and will not be discussed in detail. SAFA at present exists only in emulated form.

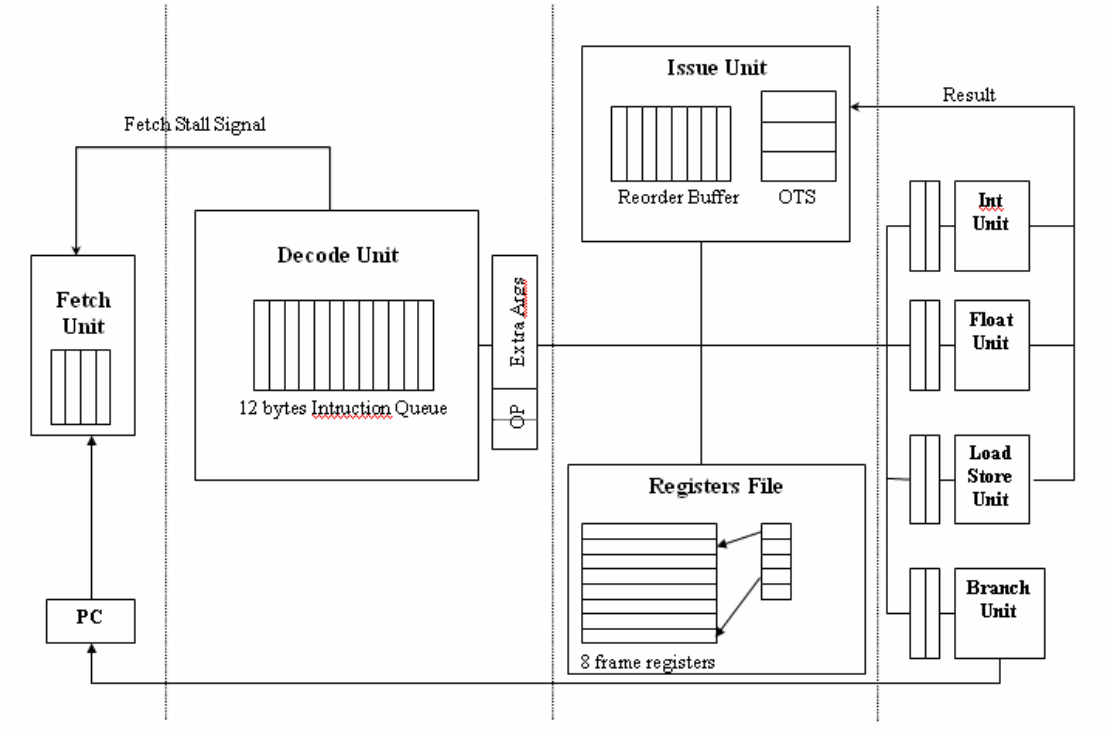


Figure 1-1 SAFA Architecture

1.2 Programming Language and Program

1.2.1 Programming Language

A Programming language is a formal notation for expressing algorithms. Machines are driven by programs expressed in machine code, where each instruction is just a bit string that is interpreted by machine to perform some defined operation. In the early days, programs were written directly by machine code.

Clearly, machine code programs are extremely difficult to write and modify, and almost impossible to understand. The symbolic language, which is much easier to understand and is prepared to run by manually translating each instruction into machine code, is defined. The symbolic notation is formalized and can be termed as an assembly language.

Today, the vast majority of programs are written in programming languages of high level languages, by contrast with machine languages and assembly languages which are low level languages. In this thesis, C Language is the high level language of interest, while SAFA instructions provide the low level language, but with intermediary abstract machine and SAFA assembler also playing a part.

1.2.2 Program

A program is a notation for specifying algorithms in a form acceptable to a computer. As such a program has two main purposes: formalization of the problem to be solved and abstraction from machine specific implementation

details. Programs can be classified according to the formal model of computation and they mimic most closely in two main groups: imperative and functional. The two models are equivalent in computing power. Functional programs have some structural connection to stack machines.

According to the level of abstraction, the implement programs can be characterized as low level program or high level program. The former reveals more of the machine's hardware structure, allowing more efficient code to be written, while the latter provides more hardware independent facilities.

1.3 Compiler

A translator which performs the translation of a high level language into an intermediate language or a machine language can be defined as a compiler. Figure 1-2 elaborates the relationships involved. The target program of a compiler generally needs further processing before it can be executed.

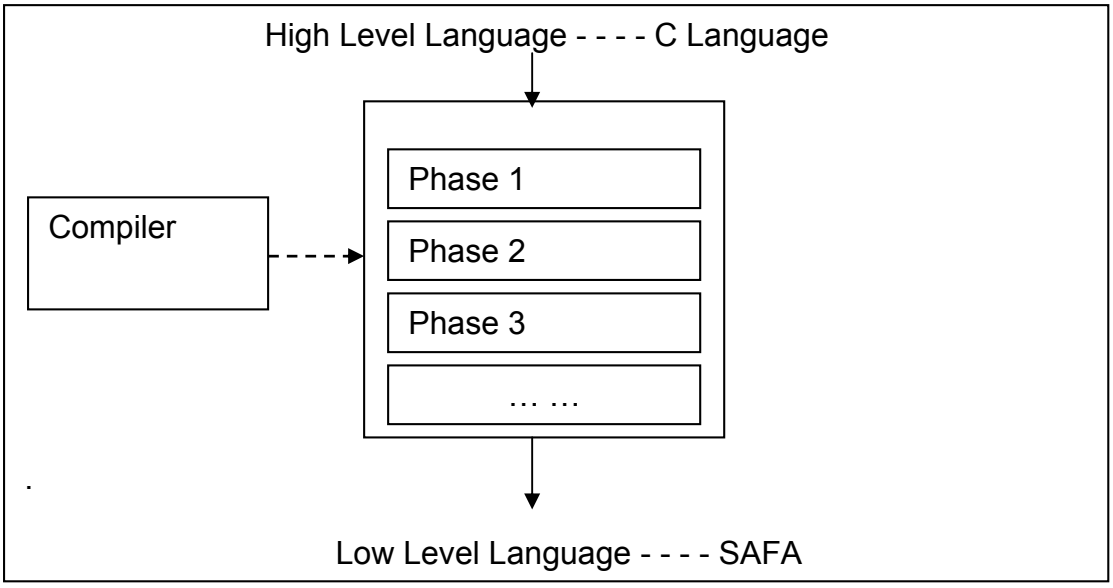


Figure 1-2 Compiler: High Level Language to Low Level Language

Figure 1-3 shows a typical language processing system. The compiler generates assembly code that is translated by an assembler into relocatable machine code. The linker links the machine code with files of relocatable code from libraries and adjusts addresses, so that the final code can actually run on the machine. [4] In this thesis, we merely focus on the compiling phases.

C to SAFA compiler is implemented in C environment. It has the potential of compiling itself so that SAFA can run the compiler for its own execution, but this has not been achieved at present. A separate machine is used to provide code for input to the SAFA emulator.

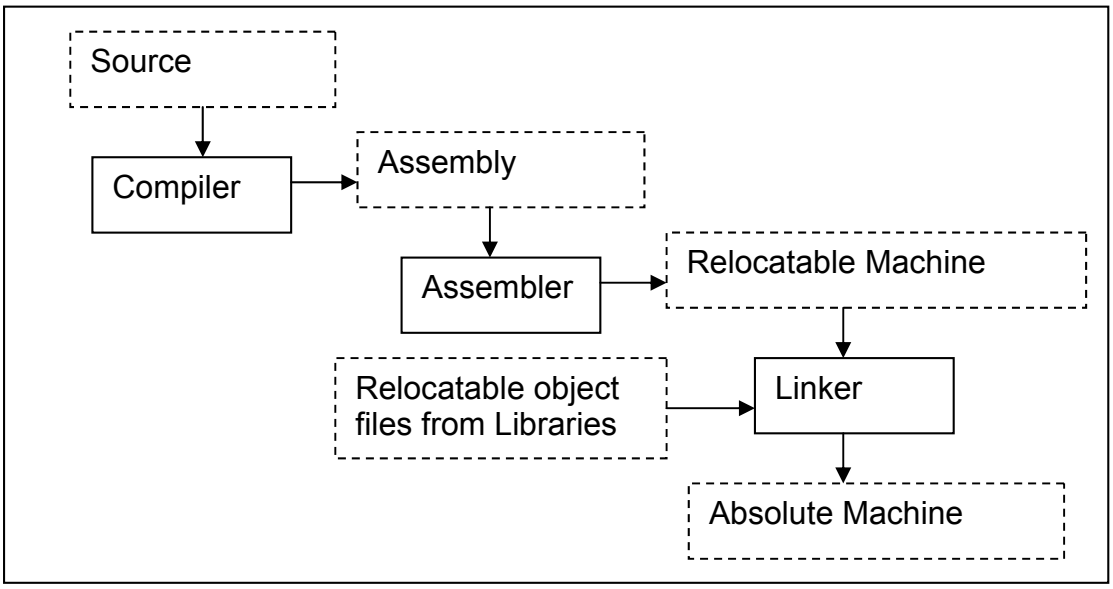


Figure 1-3 Structure of a Language Processing System

1.4 An Overview of Thesis

The thesis consists of seven chapters:

Chapter 1 gives an overview of some concepts in stack architecture, programming language, and compiler, as well as an overview of the thesis.

Chapter 2 is the starting point of the thesis work. The primary definition of C Language and SAFA instruction set with the general description of SAFA and its mechanism are proposed. Further more, we present some compiling techniques with brief survey regarding the needs of C to SAFA compiler. Based on the result of analysis, we design a compilation structure for C to SAFA compiler. In addition, the details of the compiler are elaborated.

Chapter 3 is focused on the design and implementation details of the compiler's front end as addressed in the previous chapter. The design and implementation method for the first phases of the compiler are presented.

Chapter 4 describes the back end of C to SAFA compiler. The details and the special design of the intermediate code generation phase in the compiler is given. Based on the stack scheduling method, the compiler optimizes the intermediate code. Particularly, the setting-up of frame and the allocation of frames, as well as the issues of dealing with frame registers and arrays are taken into details for discussion. The generation of assembly code is also referred.

Chapter 5 gives some samples, indications and results to show the features of SAFA design helps C in the machine code section. Furthermore, the

influence that the design of context-sensitive frame register brings in SAFA is also presented by the comparison to the ones without the mechanism of context-sensitive frame register.

Chapter 6 shows a practical sample and performance of C to SAFA compiler. We try to give a practical picture to show what the compiler does in each compilation phase. We conduct some applications and performance analysis of the compiler to show that we have attained the objective to compile the C program into SAFA program, and the efficiency of the compiler is acceptable.

Chapter 7 is the conclusion of the thesis. It summarizes the earlier parts of the thesis. Some possible improvements and future work are also proposed in the chapter.

Chapter 2

Concepts and C to SAFA Compiler Structure

In this chapter, we give a general description of SAFA, and the details of SAFA program. Additionally, we present some features of C Language and C program briefly. At last, some compiling techniques with brief survey regarding C to SAFA compiler are discussed.

2.1 Stack and Frame Architecture and SAFA Program

2.1.1 Stack and Frame Architecture

As referred in the previous chapter, Stack and Frame Architecture (SAFA) has three major features:

- Hardware stack structure that uses an recorder buffer to support instruction level parallelism
- Instructions as well as hardware support for high level programming program execution, especially procedure, method, function, entrance and exit, variables scoping and accessing.
- Improved array support for high level programming

To achieve the features above, some special mechanism is designed as shown below [2] [3]:

- High Level Programming Languages Support

One of the most frequently used operations in high level programming languages is to transfer the thread of control from one module to another,

e.g. function call, procedure transfer. Further more, the mechanism of transferring and bookkeeping of threads of control, accessing scoped variable is also considered important. In SAFA, the information needed for activation of a procedure is usually collected in a record called a stack frame. As a compromise between flexibility and hardware economic, SAFA is designed to have some frame registers with more information stored in each register as shown below:

Global Frame: describes global information

Caller Frame: describes the caller of the current procedure. Additionally, the destination of return when current procedure finishes.

Host Frame: describes the host (enclosing block) of current procedures, mostly used for accessing non local variables.

Current Stack Frame: describes the current running procedure.

Current Data Frame: describes the data frame where the data is stored currently.

Previous Data Frame: the frame that previously stored in Current Data Frame. It is automatically updated when the Current Data Frame is changed.

- Array Indexing

The most commonly used data structure in high level programming language - array is specially emphasized in SAFA. The frame register is

used to cope with the array indexing problem. The content of a frame register comprises five fields:

- Base: Starting address of an array
- Interval: Number of elements skipped for each iteration
- Index: The position of the current element accessed
- Limit: Upper bound of array
- Size: Size of each element (bytes)

To collaborate with this structure, some operations are designed, e.g. load current array element to stack, store element at top of the stack to the current position, increase the index by on stride, decrease the index by one stride, compare index to limit and leave result on stack.

2.1.2 SAFA Program

SAFA Program is composed of a set of SAFA instructions. Some of the sample instructions are shown below as in Figure 2-1:

OpCode	Pop (operands)	Push (result)		Usage
<0x56>	1	1	0x56	Increment
<0x60>	2	1	0x60	Add Float Word
<0x28>	0	0	0x28	Set current to global

Figure 2-1 Sample Instructions of SAFA

A full list of SAFA instructions can be found in Appendix A. To help better understanding the compilation process of C to SAFA compiler, we describe some features of SAFA. The key elements and procedures in a SAFA program are [3]:

- Frame Information

The frame information of a SAFA program are composed by Base (4 bytes), Limit (2 bytes), Index (2 bytes), Size of Element (1 byte: currently valid size 1,2,4,8), Interval (1 byte), and the information is stored in the format shown in Figure 2-2. And all the instruction that interact with frame information adheres to the format above.

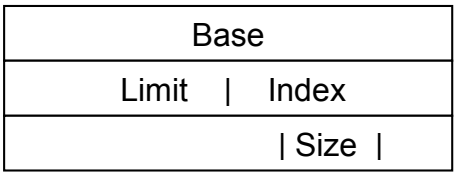


Figure 2-2 Format of Frame Information

- Boot Up

1. When emulator started, the following frame registers are set:

a. Global Frame Pointer, the frame contains the addresses of various procedure definitions in program.
b. Own Frame Pointer, a frame for main program is set up. 16 Memory words are allocated for this frame.
c. Current Frame Pointer, points to Own Frame Pointer.
d. Both Host Frame Pointer and Caller Frame Pointer are set to zero.

2. Code for procedures is stored in respective segment (separated by 0xffffffff). The Global Frame serves as a Segment Directory that stores starting address of all procedures.
3. Stack Segment (for running stack frames) are allocated.

- Procedure Entry

1. When the instruction 0xb0 (Enter) is reached, the following conditions must be satisfied for a legal procedure entry:

a. Current Frame Pointer points to the newly set up stack frame for Callee.
b. Previous Frame Pointer points to the Host stack frame for Callee.
c. The topmost word in stack is the address of callee.

2. One possible way of setting up a legal procedure entry is as follows:

a. Switch to Global Frame and load destination procedure address.

a. Search for Host Frame for callee and set current to it.
b. Set up a new frame, and let current points to it.
c. Store the following information:
d. Dynamic/Static Links for callee
e. Parameters
f. Enter procedure.

3. Immediately after the Enter Instruction, the following conditions hold:

a. Current Frame Pointer now points to the newly entered procedure's frame
b. Own Frame Pointer, Caller Frame Pointer and Host Frame Pointer is set up correctly.
c. PC of caller is saved in offset 0x20 in the caller's frame.

- Procedure Exit

1. When the instruction 0xb8 (Exit) is reached, the following conditions must be satisfied for a legal procedure exit:

a. Current Frame Pointer points to Caller's caller stack frame.
b. Previous Frame Pointer points to Caller's Host stack frame.
c. The topmost word in stack is the return address.

2. One possible way of setting up a legal procedure exit is as follows:

a. Store returns result in offset 0x00 and 0x04 if any.
b. Switch to Caller Frame and load return address.
c. Load Host frame info and store in current frame pointer.
d. Load Caller frame info and store in current frame pointer.
e. Exit procedure.

3. Immediately after the Exit Instruction, the following conditions hold:

a. Current Frame Pointer now points to the caller procedure's frame
b. Own Frame Pointer, Caller Frame Pointer and Host Frame Pointer is restored.

2.1.3 Structure of Global Frame

As in most of the platforms, a SAFA program is compiled into several self contained segments according to the program structure. Each of the resultant segments can be loaded into any portion of the memory without causing any problem. The entry points to the segments are recorded in an array (usually named segment directory in other platform). To enter a

particular procedure (segment), the corresponding index in the segment directory and the offset to the starting of the segment are needed.

Since the segment directory can be considered as part of a global environment of a program, it is included in the Global Frame on the platform. This global frame also contains other bits and pieces of contextual information like global data, command-prompt parameters etc. For ease of access, a global frame pointer (a frame register) will always points to the Global Frame during the execution of a SAFA program. Figure 2-3 below is included to illustrate the relation between segments, segment directory, global frame etc.

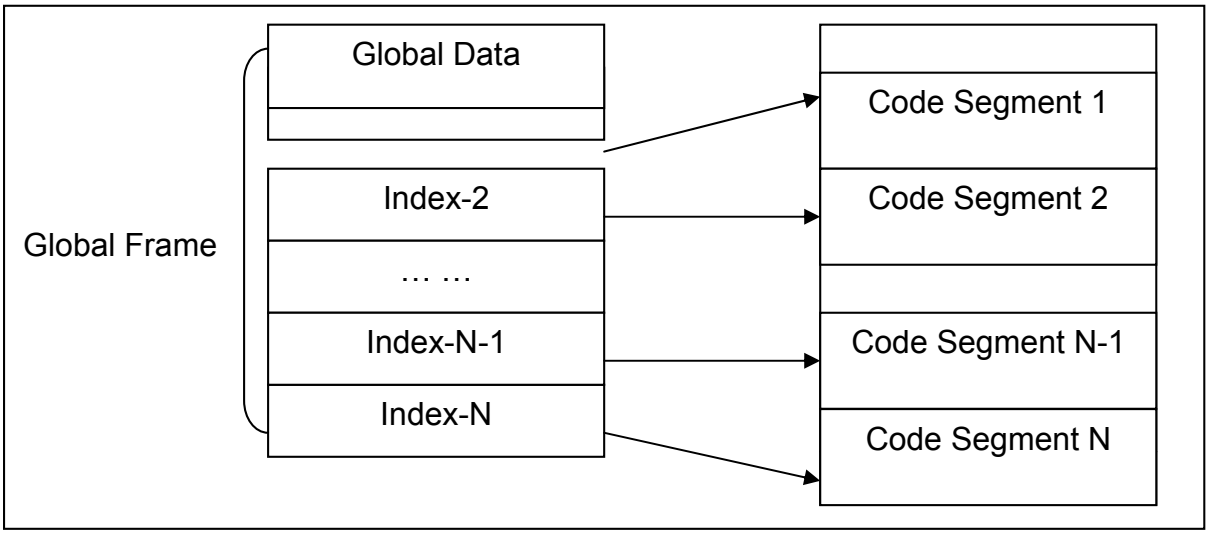


Figure 2-3 Structure of Global Frame

2.2 C Language

C Language is a high level programming language. It has proved to be a powerful and flexible language that can be used for a variety of applications. Although it is a high level language, C is much closer to

assembly language than most other high-level languages. It is close to the underlying machine language and has low level nature.

All C programs consist of at least one function, but it is normally that a C program comprises several functions. The only function that has to be present is the function called main. For most programs the main function act as a controlling function calling other functions. The main function is the first function that is called when the program executes. It may also contain global variables. Typically, the top of a program is a few boilerplate lines, followed by the definitions of the functions. Each function is further composed of declarations and statements. When a sequence of statements should act as one, they can be enclosed in braces. The simplest kind of statement is an expression statement, which is an expression followed by a semicolon. Expressions are further composed of operators, objects, and constants.

Regarding the lexical elements in a C program, some are words, which are either keywords or identifiers. Also, there are both constants and operators which introduce new values into the program and manipulate variables and values. Punctuation characters indicate how the other elements of the program are grouped. Furthermore, the preceding elements can be separated by spaces, tabs characters, and the returns between lines. C Language has several extensions, and each has some of its own particulars, although they are all based on the standard of C Language. In the thesis, we choose the ANSI C as the source program.

2.3 Structure of C to SAFA Compiler

2.3.1 Compiler Structure

Generally, a compiler consists several phases/ modules to fulfill the entire compiling process. Being two different languages, C program and SAFA program have relatively divergent differences, not only on the form of codes, but on the mechanism. Compared to some regular C compilers which generate machine code for the general purpose register architecture, such as Intel X86, SPARC, etc, generating machine code for stack architecture, such as Forth which is a stack architecture based language or SAFA in the thesis, will be of high difference.

The front end of C to SAFA compiler performs lexical, syntactic, and semantic analysis, and some simple optimizations. The back end of the compiler is the intermediate code generation, assembly code generation and target code generation. The premise of the classification is whether the phases involved are machine dependant or machine independent. Obviously, the phases that are involved in the front end are machine independent, while the phased in the back end are machine dependent. The design and implementation of the back end highly depends on the target machine – SAFA. C to SAFA compiler is designed as shown in Figure 2-4.

As shown in the structure of compiler (Figure 2-4), C to SAFA compiler consists of four major phases, which are lexical and syntax analysis, intermediate code generation, assembly code generation, and target code

generation. Within the four major phases, there are optimizations involved. The details will be addressed in the subsequent parts in this chapter. Between each phase of the compiler, the information that is transferred among each phases is presented as the gray color rectangles. E.g. when intermediate code generation phase finishes, the intermediate code will be transferred to the target code generation phase for subsequent processing.

Between the front end and the back end of C to SAFA compiler is the intermediate code that is generated as the output of the front end. The back end of the compiler is completely based on the mechanism of SAFA; therefore, we define the left phases in the compiler as the back end.

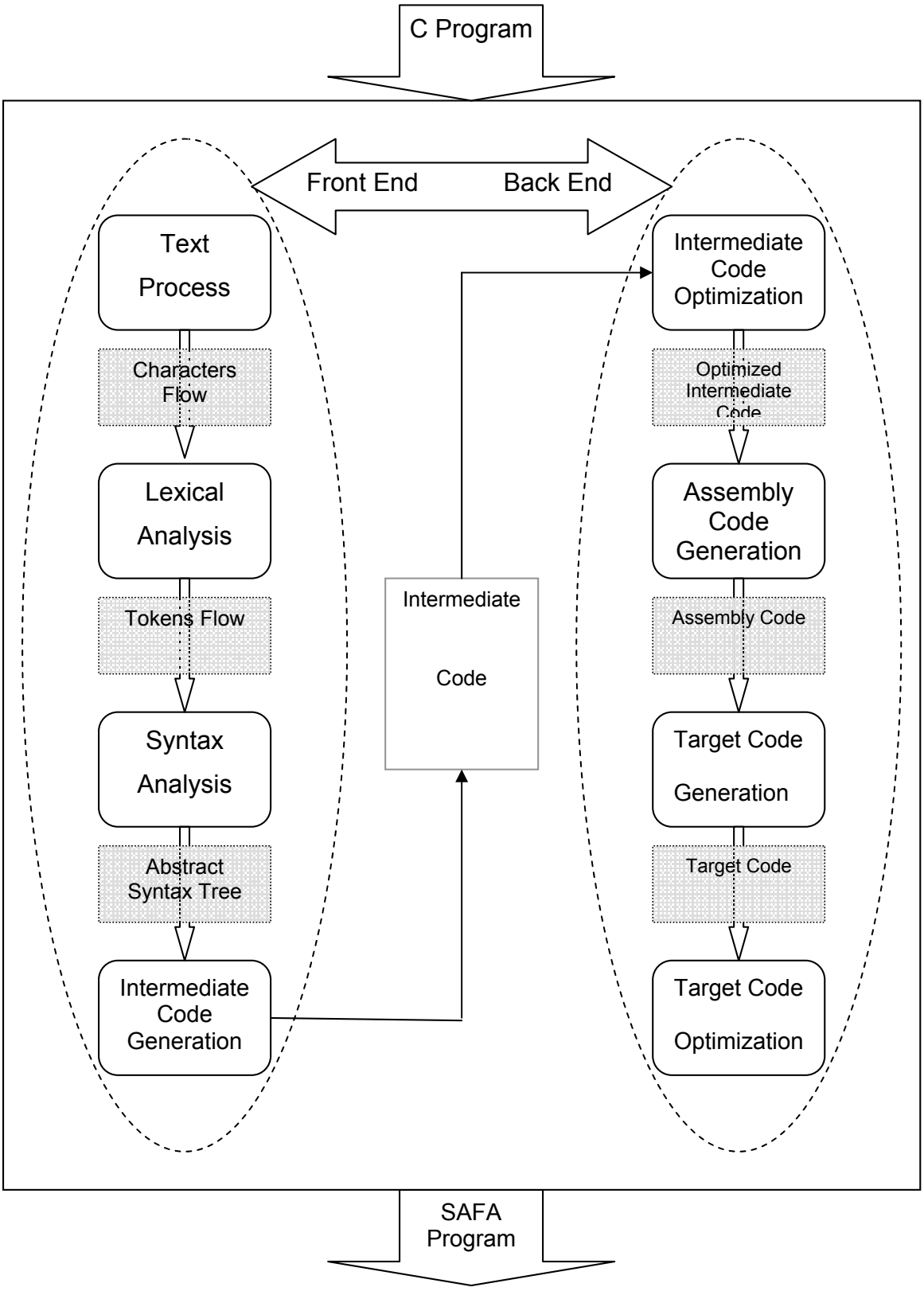


Figure 2-4 Structure of C to SAFA Compiler

2.3.2 Survey of Compilation Techniques

Considering the functions and relationships between each phase, some compiling techniques upon to each phase can be generally described:

- The text process phase receives the source program and processes it into a flow of characters. It may also switch to other files, as in most of high level language program may include some external files. This function may require cooperation with the operating system on the one hand and with the lexical analyzer on the other.
- The lexical analysis phase isolates tokens in the input stream and determines their class and representation. In this phase, it is possible to do some limited interpretation on some of the tokens.
- The syntax analysis phase converts the flow of tokens into an abstract syntax tree. Normally, syntax analysis consists of two sub phases. The first one reads the token flow and calls a function from the second sub phase for each syntax construct. It recognizes; the functions in the second module then construct the nodes of the abstract syntax tree and link them. This has the advantage that one can replace the abstract syntax tree generation module to obtain a different abstract syntax tree from the same syntax analyzer.
- The intermediate code generation phase translates language-specific constructs in the abstract syntax tree into more general constructs. The general constructs then constitute the intermediate code. Deciding what is a language-specific and what a more general construct is

reasonably straightforward up to the machine code depending on various machines.

- The intermediate code optimization phase performs preprocessing on the intermediate code, with the intention of improving the effectiveness of the target code generation phase.
- The assembly code generation rewrites the abstract syntax tree into a linear list of target machine instructions, in more or less symbolic form. To this end, it selects instructions for segment of abstract syntax tree, allocates registers to hold data and arranges the instructions in the proper order.
- The target code generation phase converts the symbolic machine instructions into the corresponding bit patterns. It determines machine address of program code and data and produces tables of constants and relocation tables.

Chapter 3

Design and Implementation of Compiler Front End

3.1 Lexical Analysis

Lexical analysis and syntax analysis are the first two compulsory and most common phases in a compiler. The lexical analysis and syntax analysis were firstly deigned to use a lexical and syntax analyzer generator. The advantage of this method is that it reduces the complexity of the lexical and syntax analyzer; however, it costs nearly half of the compilation time as we examined. Considering the situation, we design and implement the lexical and syntax analyzer for C to SAFA compiler.

3.1.1 Lexical Analysis

As defined, the lexical analysis is to read the C program and produce tokens. The lexical analyzer is the only part of the compiler that looks at each character of the source text. It is not unusual for lexical analysis to account for half of the execution time of a compiler.[7] This is the most significant drive for us not to use a third party lexical analyzer generator, such as Lex. The main activity of the lexical analysis is to move characters, so minimizing the amount of character movement helps increase speed.

In C to SAFA compiler, the lexical analysis comprises two modules: The input module and the recognition module.

The input module reads the source in blocks, usually more than one line in source code, and it helps arrange for complete tokens to be present in the

input buffer when they are being examined, except identifiers and string literals. To minimize the overhead of accessing the input, the input module exports points that permit direct access to the input buffer.

The recognition of tokens can be considered according to the number of classes of tokens in C, which is keyword, identifier, constant, string-literal, operator, and punctuator. Thus the recognition module can be divided into for major parts: recognition of keywords, identifiers, numbers, character constants and strings.

3.1.2 Implementation Concerns

We implement the lexical analysis and syntax analysis referring to some of the implementation strategy of lexical analysis and syntax analysis in LCC [7]. LCC's hand-written lexical analyzer and parser are of sufficiently effective based on ANSI C. To fulfill lexical analysis, several functions are created according to the design strategy in the previous section.

- Input:

To minimize the overhead of accessing the input, the input module exports pointer that permit direct access to the input buffer: extern unsigned char *cp; extern unsigned char *limit;. cp points to the current input character, limit points one character past the end of the characters in the input buffer, and *limit is always a new-line character and acts as a sentinel. The important sequence of this design is that most of the input characters are accessed by *cp, and many characters are never moved. Only identifiers

(excluding keywords) and string literals that appear in executable code are copied out of the buffer into permanent storage. Function calls are required only at line boundaries, which occur infrequently when compared to the number of characters in the input. Specially, the lexical analyzer can use *cp++ to read a character and increment cp. If *cp++ is a new line character, however it must call next line, which might reset cp and limit. After nextline, if cp is equal to limit, the end of the file has been reached.

- Recognition:

Tokens in C can be classified into six type excluding white spaces, tabs, newlines, and comments: key words, indentifier, constant, string-literal, operator, and punctuator. We referred the strategy in reference [7].

The lexical analysis exports two functions and four variables: Exported functions: extern int getchar ARGS((void)); extern int getok((void)). Exported data: extern int t; extern char *token; extern symbol tsym; extern Coordiante src; getok returns the net token, get char returns, but not consume, the next nonwhite-space character. The values returned by getok are the characters themselves, enumeration constants for the key words, etc.

To be summarized, the implementation strategy comprises: read the input in large chunk into a buffer and examine the characters to recognize tokens.

3.2 Symbol Table Maintenance

The symbol tables are the central repository for all information within the C to SAFA compiler. All parts of the compiler communicate via the symbol tables and access the data in them. Symbol tables map names into sets of symbols. The symbol table deals with the symbols themselves, as well as handles the scope or visibility rules in ANSI C, e.g. declarations of variables in a function make the identifiers visible until the end of the function, which means each statement or parameter have their own scope.

To represent symbols, the name and all of other attributes of a symbol is collected into a single symbol structure. Symbol tables are to implement the name spaces in ANSI C. extern table constants; extern table externals; extern table globals; table identifiers; extern table labels; extern table types. The symbol table is presented as a list of hash table, each of which represents one scope (function) in a C program.

A table value, e.g identifiers, points to a table structure that holds a hash table for the symbols that are in one scope. Entries in the hash table lists hold a symbol structure and a pointer to the next entry in the list.

The value of the global variable level and the corresponding tables represent a scope. To change a scope, level is increased when entering a new scope and decreased with removing the corresponding identifiers and types when scope exits. To put variables in a table, function install I(name, tpp, level, arena) fulfills the task, where tpp points to a table pointer. There are also functions for dealing with labels and constants.

This symbol table structure is also similarly employed in the maintenance of the program, procedure and block information in the intermediate code generation.

3.3 Parsing

The output of the lexical analysis is a sequence of tokens. The identifiers in the sequence need some identification and further processing for the benefit of macro processing and subsequent syntax analysis. Referred to the parser in LCC, we build the parser in three major components, which are parsing expressions, statements and declarations.

3.3.1 Expressions

According to the syntax for ANSI C, C has eleven types of expression nonterminals in the grammar. C to SAFA compiler parses the expressions in five major categories, which are assignment expressions, conditional expressions, binary expressions, unary and postfix expressions and primary expressions. Each is associated with a function, which builds a tree to represent the expression and do type checking for the tree.

Expressions have to be correct in both syntax and semantic. It is necessary to do the two basic analyses on semantic issues, which are implicit conversions and type-checking. Implicit conversions are conversions that do not appear in the source program and must be added by the compiler in order to adhere to the semantic rules of the standard. E.g. in a-b*c, if a, b, c are of the different types of variables, that a is float,

b and c are integers. A conversion is done to b and c to convert them into float. Type checking confirms that the types of an operator's operands are legal, determines the type of the result and computes the type-specific operator that is used.

What is to be emphasized here is the analysis of the function calls, which are easy to parse but difficult to analyze. [7] The semantic analysis for function calls handle with calls to both new style and old style functions in which the semantics imposed by the standard affect the conversions and argument checking, the order of evaluation of the arguments, passing and returning structures by values, and actual arguments that include other calls.

3.3.2 Declarations

Declarations in a C program are to specify the types of identifiers, define structure and union types, and give the code for functions [7]. The parsing of declarations is mainly divided into five segments, which are the translation units, declarations, declarators, functions declarations, and structure specifiers.

3.3.3 Statements

Statements in C can be summarized as conditional statements, labels and GoTos, loops, switch statements, return statements, and compound statements according to the syntax of C statement.

In C to SAFA compiler, an execution point will be assigned under the following conditions: a compound statement's entry and exit, and a function's entry and exit. The design here it to provide sufficient information for the intermediate code generation to load and store the stack frame or frame register information for each block of the program. The details will be discussed in the subsequent chapter.

3.3.4 Implementation Concerns

The implementation of the parser consists of three stages according to the design described above, which are parsing expressions, parsing statements and parsing declarations.

3.3.4.1 Parsing Expressions

Parsing an expression is based on a tree, which is defined as the algorithm shown in Figure 3-1.

```
Typedef struct tree *tree
Struct tree {
Int p;
Type type;
Tree children [2];
Union {
Symbol symbol;
} ;
};
```

Figure 3-1 Definition of Expression Tree

P holds a code for the operator, type points to a type for the type of the result computed by the node, children point to the operands. Identifiers are categorized by their scopes and lifetime and their types. The identifiers' scope and storage class to determine its addressing operator, and then

uses its type to determine the shape of the tree that accesses it, as well as stores a pointer to the symbol table entry in the symbol in the tree.

3.3.4.2 Parsing Statements

The parsing function of statements employs the current token to identify the kind of statement and switches to statement-specific code. Each type of statements will be transferred to the corresponding handling function for parsing.

```
  Void statement (int loop,  switch swp, int lev) {
  If (aflag>=2 && lev ==15) exit; //too many levels of statements
  Switch (t) {
  Case if: <if statement> break;
  Case do: <do statement> break;
  …
Default: <expression statement>;
}
```

Figure 3-2 Parsing Statement

Chapter 4

Design and Implementation of Compiler Back End

In this chapter, we describe the design and implementation of the most significant phases in C to SAFA compiler, the compiler's back end. SAFA has some distinguishing specifications, such as stack frame, frame register, support for high level programming and array indexing, which leads the compilation to be specific. To discuss more clearly, we give some of the details of SAFA's parameters, and mechanisms. Particularly, we emphasize on some special issues in these phases.

4.1 Intermediate Code Generation

An intermediate code in C to SAFA compiler is designed to represent a kind of abstract machine language that can express the target machine operation. The front end of the compiler as we discussed does lexical analysis, and parsing with semantic analysis, as well as intermediate code generation. Why we put the intermediate code generation in the back end of the compiler to discuss is that it has very tight relationship with the assembly code of SAFA in the compilation. Isolating it from the assembly code generation will be difficult to address the ideas adhered. In fact, the intermediate code and the assembly code of C to SAFA compiler have great similarities to make it possible to generate the intermediate code into assembly code directly. Meanwhile, before generating into assembly code, we also do some optimizations for the intermediate code. There are several representations of intermediate code, such as postfix

representation, three-address code representation, XML representation, etc. Because SAFA code is a stack machine based code, we select the postfix representation which can work with the stack architecture more easily.

4.1.1 Representation and Maintenance of Code

SAFA has an important feature that is the support for high level language. SAFA program consists of several procedures and the procedure is the basic components in it. The SAFA assembler is also designed to represent the SAFA code as several chunks correspondingly. To fit in with the assembler, we design a representation of the code to meet the requirement. Further more, the maintenance of the representation mechanism should also be taken into consideration.

We consider the statement is the smallest block in a program. The semantic of statements consists of evaluation of expressions, sometimes mixed with jump and label, which means transfer of control. Expressions are compiled into trees and then converted to postfix representation of intermediate code. For each function, these structures are strung together in a code list, which represents the code for the function. In the front end of the compiler, the code list is built, and in the back end of the compiler, the code list will be generated into the intermediate code representation.

To give a clearer picture of the code list structure, the abstract structure of a code list is shown in Figure 4-1.

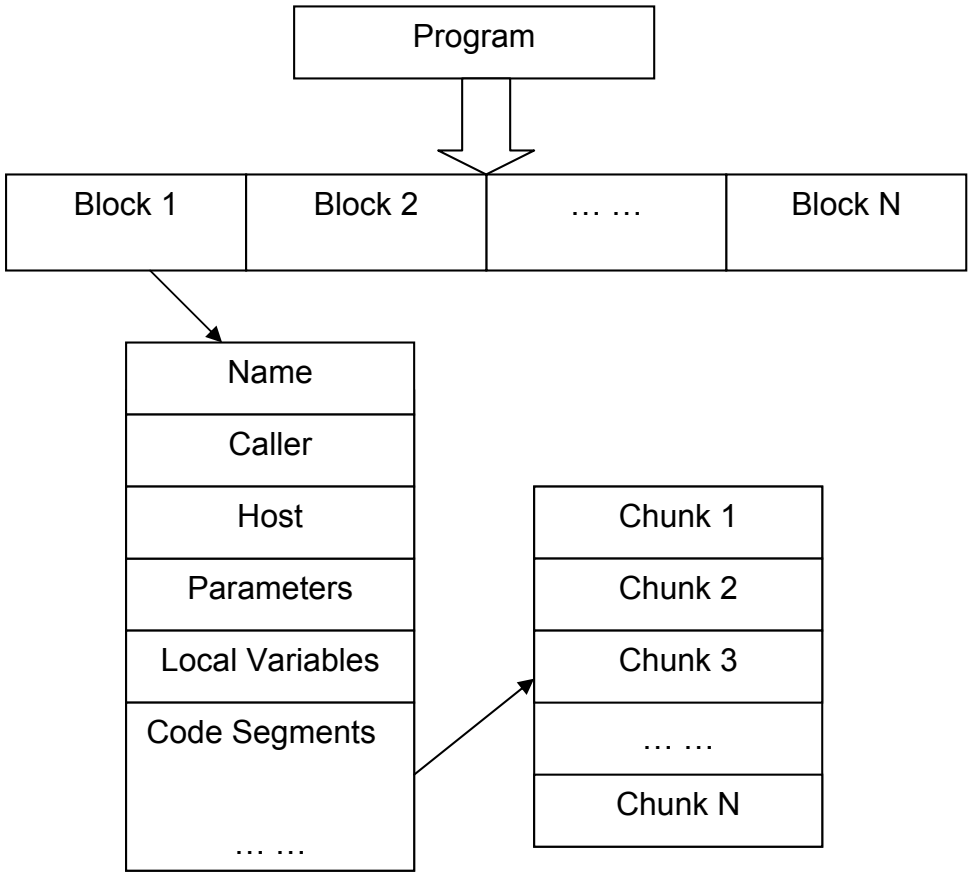


Figure 4-1 Code List

A program will be divided into several blocks, with each block representing a function. The blocks are interconnected by a list structure. The details of a function, such as the name, the caller, the host, the parameters and local variables, etc is stored. This design is quite coincided with the structure of the global frame in SAFA (The structure of a global frame can be seen in Figure 2-3).

The code segments in the block are also divided into several chunks and stored. An evaluation expression, e.g. a=b+c; can be a chunk. An if statement can also be a chunk, e.g. if…else…The aiming of this structure is to coincide with the design of SAFA assembler by storing sufficient and structured information to make the generation from intermediate code to

assembly code much easier. The implementation algorithm of the code list can be presented as in Figure 4-2.

```
Typedef struct codelist *codelist
Struct code{
Codlist name;
Codelist previous, next;
<idx> caller;
<idx> host;
<Parameter> parameters;
<Variable> localvariables;
Code *code;
<other information>
}

Typedef struct code *code;
Struct code {
Enum {blockbegin, blockend, local ,address, definitionpoint, label, start, gen,
jump, switch}; kind;
Code previous, next;
Union {
<blockbegin>
<blockend>
<local><address>
<definition point>
<label>
<start>
<gen>
<jump>
   <switch> } u;
};
```

Figure 4-2 Algorithm for Code List

There are some special cases that need to be specially considered, such as dealing with array, loop, etc in C. The details will be discussed in the subsequent sections in this chapter.

4.1.2 Generating Intermediate Code

Before entering into the generation of intermediate code, we have to recall and make the design of symbol table more specific for SAFA.

We see that in order to generate target code correctly, the compiler must keep track of all the identifiers introduced by the source code. For each identifier, we must record what the identifier stands for in the source language, and on which construct it is mapped in the target language. This information is usually recorded in a “housekeeping” data structure called symbol table. Whenever a new identifier is encountered in the source code for the first time, the compiler adds its description to the table. Whenever an identifier is encountered elsewhere in the program, the compiler consults the symbol table to get all the information needed for generating the equivalent code in the target language. Here is an example of the C program (Figure 4-3), the corresponding symbol table and the intermediate code for the main function.

```
typedef BankAccount () {
            int nAccounts;
            int bankCommission;
            int id;
           String owner;
           int balance;
}
int commission(int x) {
          if id<=1000 the acct owner is a bank employee so commission is 0
          if (id>1000) return (x*bankCommission)/100;
                    else return 0;}
void main (int sum, bankAccount from) {
int i,j,k;
let balance=(balance+sum)-commission(sum*5); }
```

Figure 4-3 C Program for Symbol Table

Class-scope symbol table			**Method-scope (**transfer**) symbol table**			
Name	**type**	**#**	**name**	**Type**	**kind**	**#**
naccounts	int	0	this	BankAccount	argument	0
bankCommission	int	1	sum	int	argument	1
id	int	0	from	BankAccount	argument	2
Owner	String	1	when	Date	argument	3
balance	int	2	i	int	var	0
j		int	var		1	
k		int	var		2	
d1		Date	var		3	

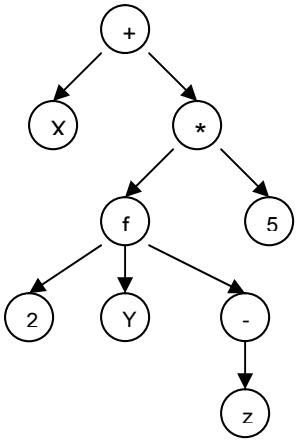
Figure 4-4 Symbol Table for C Program

```
push balance
push sum
add
push this
push sum
push 5
multiply
call
commission
sub
pop balance
```

Figure 4-5 Partial Intermediate Code for C Program

Expressions are the most important component in a C program. It can appear in any part of the program. It can independently exist as a statement, or be part of a condition statement, e.g if or switch, etc. To evaluate an expression, the mechanism is to depth-first traverse the syntax tree we generated from the syntax analysis and generate it into the postfix representation of the intermediate code. Here's an instance, an

expression in a C program w=x+f(2,y,-z)*5, then the syntax tree and abstract intermediate code will be (Figure 4-6):




```
push x
push 2
push y
push z
unary
call f
push 5
multi
add
store
```

Figure 4-6 Syntax Tree and Abstract Intermediate Code

An example for the strategy used in algorithm is like (Figure 4-7):

```
Code(exp):
if exp is a number n then output “push n”
if exp is a variable v then output “push v”
if exp = (exp1 op exp2) then Code(exp1); Code(exp2) ; output “op”
if exp = op(exp1) then Code(exp1) ; output “op”
if exp = f(exp1 … expN) then Code(exp1) … Code(expN); output “call f”
```

Figure 4-7 Example of Strategy Used in Generation

4.2 Setting up Frame for Procedures

In a SAFA program, the first procedure is always main procedure. If there’s only one procedure (main procedure) in the program, there will be no more operations for setting up frames. If there is more than one procedure including main procedure, the frame that is used for running the callee procedures have to be set up before entering the callee procedure in the main procedure (caller procedure). Here’s an example to show the process. Figure 4-8 shows a program for bubble sort, and we omit the

other functions (BubbleSort) in the program, other than LCG and main functions, to simplify the discussion, because setting up frame for most procedures are quite similar. Figure 4-9 shows the corresponding SAFA program.

```
void LCG(int ia[], int n, int a, int c, int m)
{
   int i,seed=1;

   for (i = 0; i < n; i++){
      seed = (a*seed + c) % m;
      ia[i] = seed;
   }
}
void main()
{
   int array[100];
   int i;
   LCG(array,100,1277,0,131012);
   BubbleSort(array,100);
}
```

Figure 4-8 C Program for Setting up Procedures

```
1 | 5A FA 03 0A 00 73 42 0A 2D 24 2C 24 42 04 B1 04
2 | 3C 3A 2B 2C 24 44 04 FD 42 00 46 00 01 FF C4 2B
3 | 3A 28 3A 46 00 15 00 08 03 44 04 00 34 2D 1C 2D
4 | 18 2D 14 2D 10 2D 0C 2D 08 2D 3C 2D 38 2D 34 2D
5 | 30 2D 2C 2D 28 2D 24 28 2C 04 03 B0 04 3A 2B 2C
6 | 24 2B 3A 28 3A 46 00 13 00 08 03 44 04 00 34 2D
7 | 1C 2D 18 2D 14 2D 10 2D 0C 2D 08 2D 30 2D 2C 2D
8 | 28 2D 24 28 2C 08 03 B0 E0
```

Figure 4-9 SAFA Program for Setting up Procedures

From the last instruction “2B” (set current frame pointer to own) in Line 2 to the instruction “B0” (enter procedure) in Line 5 are the instructions for setting up frame for procedure LCG.

The instructions in Figure 4-9 fulfill the activities referred above. Here we address some primary instructions. “2B” in Line 3 is to set the current

frame to own frame and “3A” in Line 4 loads the current frame information to stack. “28 3A” in Line 3 acts to set current frame to global frame and load the current frame to stack. These four instructions load own frame information and global frame information to stack. The instructions from the first “2D” in Line 3 to the 7th instruction in Line 5 (“24”) are to store the parameters and the local variables in procedure LCG. “28” which is next to “24” in Line 5 is to set current frame to global frame. “B0” is to enter the procedure LCG. This is the most common and compulsory process to setting up frame for procedures.

The most common processes for setting up frame for procedures are:

- Set current frame to own frame
- Load current frame information to stack
- Set current frame to global frame
- Load current frame information to stack
- Set current frame to the frame used
- Stack a new frame
- Store parameters and local variables in the procedure
- Set current frame to global frame
- Set current frame to the frame used
- Enter procedure

4.3 Allocating Frames and Dealing with Frame Registers

SAFA has some specifications as a stack architecture. A very important issue in generating intermediate code is to establish and calculate the

essential information that is needed to transfer to the SAFA assembler to generate the operations on stack frame and frame register. Before presenting the strategies, we have to review some of specifications for further discussion.

- To call a procedure， what should be delivered to the assembler is the frame pointer that contains the callee's frame and the name of the procedure.
- To exit a procedure, the frame pointer that contains the host's frame and the frame pointer that contains the caller's frame should be delivered.
- To establish a procedure, the name of the procedure, the number of parameters, and the number of local variables are required.
- When a procedure, except the main procedure, is involved, the frame content that the procedure used must firstly be saved for backup. When the procedure exits, the information will be restored from the place it is saved.

To work with SAFA assembler better, we define three instructions in the intermediate code:

- ENTER fnum, sub_name

fnum: The frame register number that contain the callee's frame

sub_name: The name of the procedure

- EXIT hfnum, cfnum

hfnum: The frame register that contain the host's frame

cfnum: The frame register that contain the caller's frame

- PROCEDURE pname paranum localnum

pname: Name of the procedure

paranum: Number of paramters

localnum: Number of local variables

When the compiler wants to generate the actions above into intermediate code, the compiler will produce the information required to make it ready for assembly code generation.

- When the compiler is to establish a procedure, the intermediate code carrying with name of the procedure, number of parameters, and number of local variables will be prepared. All the information that is adhered to the procedure will be pushed into the stack and stored in stack frame. The intermediate code will help to fulfill the activities. According to the design of SAFA program, procedures can be placed in any order as long as callee should be placed before the caller. A main function must be defined and placed as the first procedure.
- When call procedure is occurred, the frame pointer that contains the callee’s frame and the name of the procedure will be produced and the corresponding intermediate code will settle down for the generation of assembly code.

- When a procedure exits, the instructions for exit in the intermediate code with the frame pointer that contains the host's frame and the frame pointer that contains the caller's frame are produced. Similarly, all addresses and information needed will be pushed into stack and loaded into the frame register for further processing.
- When a procedure, except the main procedure, is involved, the stack frame that the procedure uses should firstly be saved for backup. When the procedure exits, the information will be restored from the place it is saved. This produces the intermediate code "SAVEFRAME" and "RESTOREFRAME" with the addresses to implement.

The structure of the stack frame is shown in Figure 4-10:

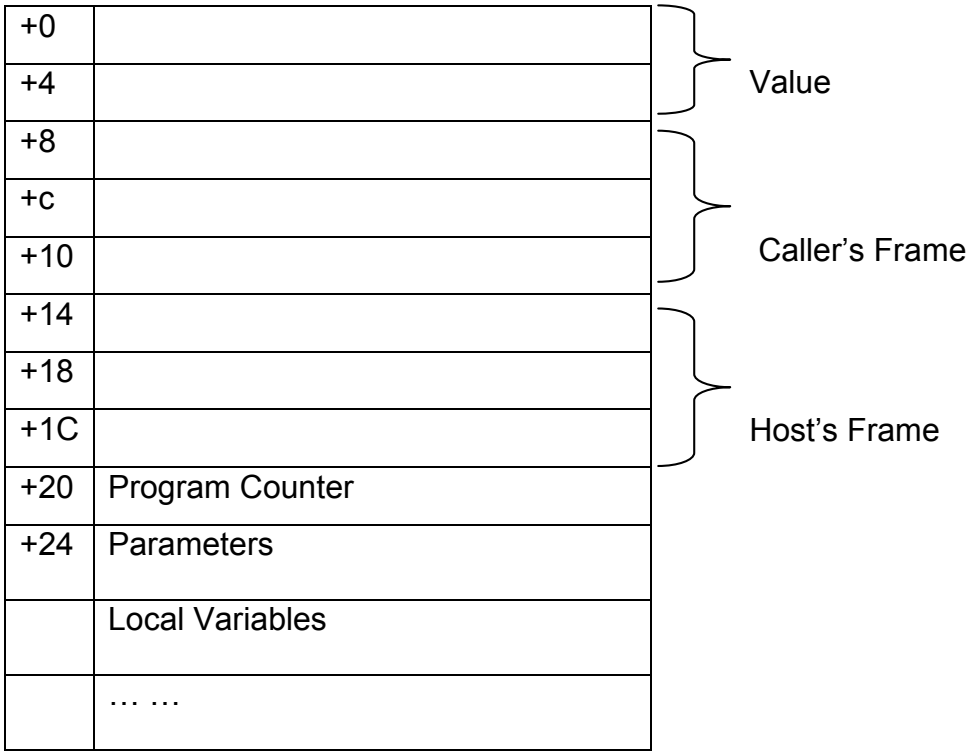


Figure 4-10 Structure of Stack Frame in SAFA

The most common processes for generating the intermediate code for frame registers that are adhered to the procedures (except main procedure) are:

- Load the information of the procedure from code list, including the name of the procedure, the number of parameters and the number of local variables in the procedure, the caller of the procedure and the host of the procedure.
- Load the procedure to the stack frame by loading the caller's frame register, host's frame register, parameters, local variables, body of the procedure, etc.
- Check availability of frame registers and set the current frame pointer for the procedure. Normally, we assign a No. 3 to No. 7 frame register to the procedure upon availability.
- Store the information on the stack to the current frame which is allocated to the procedure.
- Set the current frame register to own.
- During the setup of the stack frame information for the procedures, all the frame registers and address that are assigned will be stored in a frame and address record list.
- The abstract structure for the frame and address record list is shown in Figure 4-11. Before assigning frame and addresses, the compiler will look up the frame and address record list to check the availability. For stack frame, each frame is assigned a tag for availability. When the frame is available, the value of the tag is 0. When the frame is not

available, the value of the tag is the name of the procedure. After the procedure exits the value of the tag will be 0 and wait for next usage. For the addresses, the mechanism is similar to what it is for the frame registers.

According to the definition, SAFA has eight frame registers which can be assigned to the six different frame pointers. We also design the frame and address record list to coincide with it, because the each of the frame pointers may point to one stack frame.

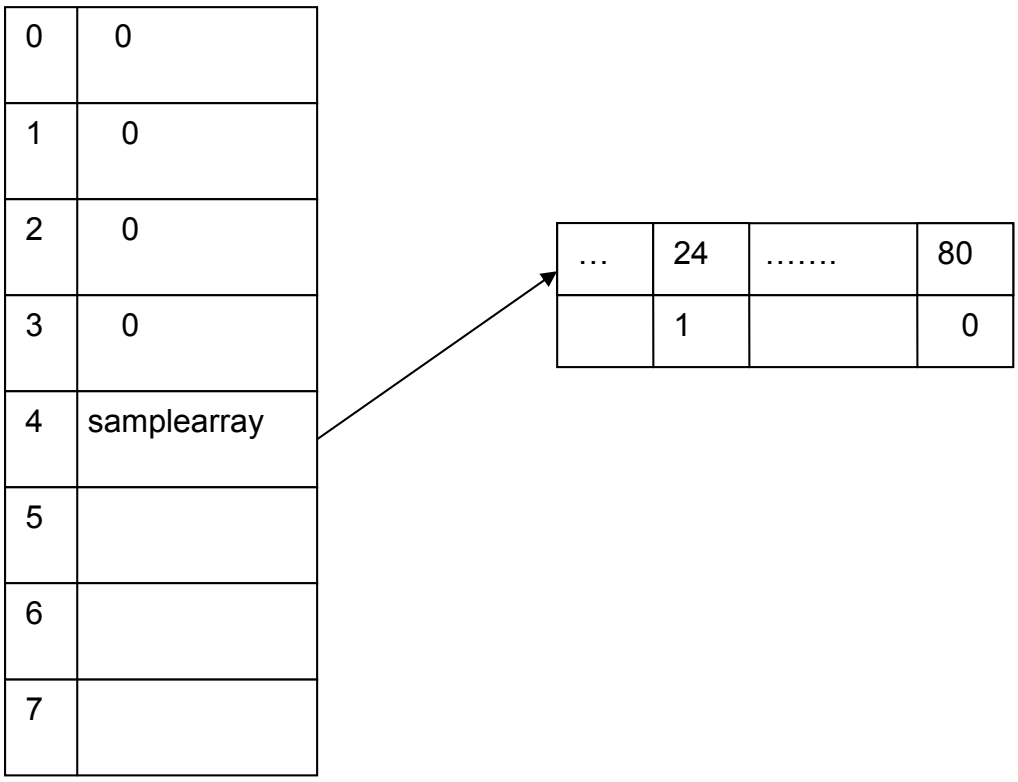
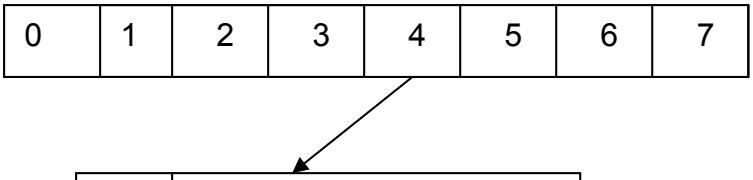


Figure 4-11 Frame and Address Record List

After the execution of the procedure, the frame information that is saved after the entering of the procedure is restored and the procedure will execute exit.

The storage of the stack frames and addresses which are assigned to the variables and the blocks in the procedures is another significant issue to be considered. After generating the syntax tree into intermediate code, many of the elements in the blocks (serials of statements) should be assigned with an address in the stack frame for execution purpose. We design an address assignment table (Figure 4-12) for blocks, statements or variables in procedure.



	…… Caller Host Previous ……
24	Parameters
28	
……	
3C	Local Variables
……	
……	Value of Variables
80	……

Figure 4-12 Address Assignment Table

There are several actions for this table:

- Look-up: To look up an item in table

- Load: Normally, load action acts with look-up action, the load action is to look up the table and load the item from the requested address.
- Store: Normally, store action acts with look up function. Store action will lead to looking up the requested address and store the content in the corresponding place. If the content area of the address is not empty (normally it should be empty if the address is assigned by the compiler, because the validation of the address is maintained by the frame and address record list, and when an address is assigned, the distribution of address will be avoided from distributing an address more than once), a warning will be return to the compiler and the compiler will assign a new address to it.
- Extension of Length of Item: Each of the content area in the table is word based, if the content that should be stored needs a larger size space than a word or an array requests space, the extension action will look up the table and look for a contiguous and sufficient space for the content, if the space is available the items will be stored and the address of the beginning of the content and the length of the content will be returned to the compiler, meanwhile the compiler will update the frame and address record list concurrently. If the space is not available, the situation will be more complicated. The extension action will send warning information to the compiler. When receiving the warning, the compiler will finish processing all the actions to the frame and address record list and address assignment table before this extension action, and send a request to the frame and address record list for new frame register. After the new register is return to the

compiler, the compiler will act the extension action once again. This process will continue to act until the available space is assign.

- Kill: Kill action is to clear all the contents in the table. A kill action often happens when a procedure exits.

Although the number of the stack frames in SAFA is not the same as the frame registers in SAFA, the reason why we design the two structures for storing information and assigning addresses can be summarized as below:

- Each frame register/ frame pointer may point to a stack frame
- The stack frames are assigned to an actual procedure in a SAFA program
- Storing the information for each procedure can provide good efficiency for the compiler to search, assign, load, modifying information.
- It provides a good channel for the compiler to allocate frame registers

To maintain the frame pointers in the program for the procedure, the global, current, previous, host, caller, and own frame registers of the procedure can be got by looking up the address assignment table, and frame and address record list. All the information that needs to maintain the frame pointers can be got and calculated. This is the significant reason why we design the number of elements in frame and address record list and address assignment table the same as the number of frame registers. The mechanism here makes the correspondence straightforward.

Since SAFA has a unique feature on array accessing, we will concentrate on generating intermediate code for array in the next section.

4.4 Array Generation

SAFA has a specification on support for array, which is the most commonly used data structure in high level programming language. Arrays are frequently used for mathematical operation, such as matrix manipulations, or as the building block for other data structures. Array indexing that is actually accessing the elements in an array, requires frequent operation of a particular value (the base of the array), which is not suitable for a stack [3]. In SAFA, the frame register is used to solve the problem. A frame register consists of five fields as presented in the early chapter. To represent an array by a frame register, the information of an array is:

- Base: Starting address of an array
- Interval: Number of elements skipped for each iteration
- Index: The position of the current element accessed
- Limit: Upper bound of the array
- Size: Size (in bytes) of each element

It is easy to see that the structure above keeps most of the important information of an array. The operations when doing a create-array instruction are:

- Load current array element into stack
- Store element at the top of the stack to current position
- Increase the index by one stride by using the formula: NewIndex = Index + Interval + 1

- Decrease the index by one stride
- Compare index to limit (upper bound) and leave result on stack

The support for array indexing is not available until the later phase of SAFA emulator. When an array is employed in the source program, after parsing, the name, the content of each element, and the size of the array will be passed to the intermediate code generation phase for processing. Without the support for array indexing, the strategy of implementing an array is to consider the array as a series of local variables. The compiler assigns a base which means the address of the first element in the array to the array and the base of the array will be pushed into the stack and loaded into the stack frame. Then the compiler pushes the elements in the array and load into stack frame one by one. All the elements are considered as local variables. This kind of mechanism will highly increase the size of the target code and lower the performance of the compiler.

After the simulation of the support for array indexing in SAFA, it is easier to handle an array. What the machine needs are the base, the size, and the limit to define an array. By pushing the size of the array into the stack and load into the stack frame, the instruction “newarray” will perform the related activities. A frame register will be assigned to carry out with the array. To modify or access the elements in array, the information stored in the frame register (base, interval, index, limit, and size) will be employed to fulfill the corresponding activity. E.g. if we want to access an element in the array, we can simply provide the base and the size of the array and the index of the element that is requested to access.

4.5 Sample of Intermediate Code

Here, we give an example of the C program with the corresponding intermediate code, as well as the Code List, the frame and address record list and the address assignment table for it.

The C program is shown in Figure 4-13. The program here is to procedure 100 numbers based on the seed.

```
void LCG(int ia[], int n, int a, int c, int m)
{
    int i,seed=1;
    for (i = 0; i < n; i++){
        seed = (a*seed + c) % m;
        ia[i] = seed;
    }
}
void main()
{
    int array[100];
    LCG(array,100,1277,0,131012);
}
}
```

Figure 4-13 C Program for Number Generation Program

The intermediate code that is generated is shown in Figure 4-14.

```
PROCEDURE LCG <7> <5>    //function LCG
    SAVEFRAME 4 x48  //save stack frame
    ibload x24
    loadnextframe
    currentframeset4   //set current frame No 4.
    Currentframeinfostore
    currentframesetown
    ibload 1
    cfb_wstore x44
    ibload 0
    cfb_wstore x40
loop:                              //loop
    cfb_wload x40
    cfb_wload x30
    ige
    iftrue end
    cfb_wload x34
```

```
    cfb_wload x44
    imul
    cfb_wload x38
    iadd
    cfb_wload x3c
    idiv
    dup
    cfb_wstore x44
    cfb_wload x40          //operations on arrary
    cfset4
    idxstore
    frstore4
    pop
    cfsetown
    cfb_wload x40
    inc
    cfb_wstore x40
    goto loop
end: RESTOREFRAME 4 x48   //restore stack frame
    EXIT 1,2  //exit the procedure

PROCEDURE main <0>  <1>   //function  main
    ibload 100
    cfb_wstore x24 //store 100 in the address x24
    cfb_wload x24
    ibload 4
    newarray  //create new arry
    cfset4   //set current frame No. 4
    currentframeinfostore  //store current frame information
    currentframeinfoload   //load current frame information
    currentframesetown    //set current frame to own frame
    cfb_wload x24  load array information to stack
    ihwload 1277  //load 1277 to stack (hword)
    ibload 0  //load 0 to stack
    iwload 131012  //load 131012 to stack
    ENTER <3>,LCG  //enter function LCG
    Halt  //terminate the Program
```

Figure 4-14 Intermediate Code for Number Generation Program

4.6 Intermediate Code Optimization

Optimizing compilers for register machines usually employs a technique called register allocation. Register allocation maps the variables used in a section of code to the machine's registers in order to reduce access times. A similar optimization technique can be used for stack based processors.

Most current stack processors make use of a “stack buffer” to cache the topmost stack elements for improved performance [14]. In analogy to register machines, this makes access to stack elements faster than access to memory.

When compiling C to SAFA, local variables of a function are usually located in the corresponding stack frame in main memory. This creates an opportunity for optimization: instead of loading the contents of a variable from main memory onto the stack each time it is used, the compiler can keep a copy of the variable’s value on the stack and reuse this copy in the subsequent operations in the same scope. An important property of SAFA makes the optimization more difficult to implement than that in the register machines, because instructions in SAFA most often use the elements at the top of the stack, as most stack architectures do. If arguments to an instruction already resides in the stack, the stack must be manipulated so that they appear in the order the instruction expects them to be. Consequently, an optimization technique must deal with that limitations and nevertheless try to minimize the usage of stack manipulations.

Philip Koopman presented a technique for the optimization stack architecture based machine in reference [14], the “intra-block stack scheduling” methodology. Intra-block stack scheduling attempts to remove local variables fetches and stores by maintaining copies of variables go on the stack for each instruction [14]. The terminology “stack scheduling” is quite similar to “register scheduling”, in which variables are assigned to

registers in conventional compilers. Basically, stack scheduling substitutes loads of local variables by stack copying and manipulation instructions.

Here we address the algorithm applied on SAFA intermediate code optimization. Suppose we have a fragment of C program:

```
c=a+b;
a=b+d;
```

The intermediate code that is generated by C to SAFA compiler, with annotations that can be inferred easily from the stack code by symbolically executing it is shown in Figure 4-15.

No. 1	cfb_wload x38	(− −)	a
No. 2	cfb_wload x3C	(a − −)	b
No. 3	iadd	(a b − −)	<+>
No. 4	cfb_wstore x40	(c − −)	(c)
No. 5	cfb_wload x3c	(− −)	b
No. 6	cfb_wload 44	(b − −)	d
No. 7	iadd	(b d − −)	<+>
No. 8	cfb_wstore x38	(a − −)	(a)

Figure 4-15 Intermediate Code Optimization with Stack Scheduling

Stack scheduling starts by annotating each instruction of a basic block with information about the stack elements present at run time before executing the instruction (Stack Picture [15]). C to SAFA compiler determines whether a stack element contains the values of a variable or not. This can simply implement by looking up the Frame and Address Record list. Then, the algorithm tries to pair each instruction that loads the contents of a local

variable onto the stack with another instruction. Basically, the compiler walks through the intermediate code searching for load instructions. When such an instruction is found, the stack pictures of the preceding instructions are searched for an occurrence of the variable referenced by the load instruction. If such an instruction is found, it is inserted into a list of pairs of instructions together with the load instruction, and the algorithm continues to search for further pairs.

Considering what is shown in Figure 4-15, there are four load instructions. The ones which are in No. 1 and No. 2 load a and b for the first time respectively, and no pair instruction can be found to create a pair. The same holds true for the load instruction at No. 6, where d is loaded for the first and last time in the basic block. When b is reloaded at No.5, the search for a pair instruction is successful, because the stack picture of the iadd instruction at No. 3 includes b, and these two instructions can be paired and inserted into the list of pairs. The list of pairs is maintained by tables which are linked by a static list in C to SAFA compiler.

The following actions in the process of scheduling consist of sorting the pairs found in the previous step according to the distance between the two instructions. The idea behind sorting at all is that instructions closer to one another are more likely to be scheduled successfully.

The last step tries to schedule each pair of the list prepared in the previous steps, in which the pairs with a small distance will be processed first. A pair of instructions can be scheduled with the following premises' satisfaction:

- The variable can be copied to the bottom of stack by a stack manipulation in front of the first instruction (the one whose stack picture includes the variable of interest).
- The copy can be moved from the bottom of stack to the top of stack at the second instruction (the one loads the variable).

If a pair can be scheduled, the appropriate stack manipulation instruction is inserted in front of the first instruction of the pair. Then the load instruction can be replaced by another stack manipulation instruction to move the copied stack element to the top of stack. After scheduling a pair, the stack picture of all instructions lying in between the first and second instruction of the pair must be updated to include the newly created stack element.

As shown in Figure 4-16, the intermediate code is optimized by stack scheduling. Obviously, the instruction duplicate has been inserted in front of the first iadd instruction. The load instruction has been omitted, because the copy created by instruction duplicate resides at the top of stack at that point, there is no need for a further pairs to schedule. Until now, the intermediate code is optimized by stack scheduling.

No. 1	cfb_wload x38	(− −)	a
No. 2	cfb_wload x3C	(a − −)	b
No. 3	duplicate	(a b − −)	dup
No. 4	Iadd	(a b − −)	<+>
No. 5	cfb_wstore x40	(c − −)	(c)
No. 6	cfb_wload 44	(b − −)	d
No. 7	iadd	(b d − −)	<+>
No. 8	cfb_wstore x38	(a − −)	(a)

Figure 4-16 Intermediate Code Optimization with Stack Scheduling II

Now we can consider a more complicated example for further addressing. The C function in the following is the function for resolving Fibonacci problem (Figure 4-17).

```
   int fibonacci(int x)
{
      if (x == 0)
         return 0;

      if (x == 1)
         return 1;

      return fibonacci(x-1)+fibonacci(x-2);
}
```

Figure 4-17 Example for Stack Scheduling

As we can see, the variable x is used in the function for four times when the function operates once. We can use the stack scheduling to optimize the intermediate code. Figure 4-18 shows the intermediate code before using stack scheduling optimization on the right side and the intermediate code after using stack scheduling for optimization on the left side.

Obviously, the cfb_load (a type of load instruction) instructions is highly decreases, instead the duplication instruction keep the x in the topmost of the stack and x is reuse for times to achieve the stack scheduling.

PROCEDURE fib <1> <0> cfb_wload x24 ifeq return0 cfb_wload x24 dec ifeq return0 cfb_wload x24 dec PENTER (3),fib cfb_wload x24 ibload 2 isub penter 3,fib iadd return0: exit (1),(2)	PROCEDURE fib <1> <0> cfb_wload x24 duplicate ifeq return0 dup dec ifeq return0 dup dec PENTER (3),fib swap ibload 2 isub penter 3,fib iadd return0: exit (1),(2)

Figure 4-18 IC before and after Implementing Stack Scheduling

4.7 Assembly Code Generation and Target Code Generation

We design the structure of the C to SAFA compiler and generate the SAFA assembly code from intermediate code instead of generating SAFA target code. Generation of SAFA assembly code is straightforward. The intermediate code is quite similar to the SAFA assembly code not only on structure and contents, but also in mechanism. We generate the instructions in intermediate code to the corresponding SAFA assembly code and transfer the information which is compulsorily required by some of the instructions on the same format. The allocation and establishment of the stack frame information is fulfilled in the assembler.

SAFA assembler can generate SAFA program straightforward from SAFA assembly code.

Chapter 5

Results on SAFA Design

In this chapter, we give some samples, indications and results to show the features of SAFA design help C in the machine code section. Furthermore, the influence of the design of context-sensitive frame register in SAFA is also presented compared to the machines without the design of context-sensitive mechanism.

5.1 Frame Register

5.1.1 Setting up and Changing Frame Register

As any program of C/ SAFA must consist of one function/ procedure at least, there must be one frame register used in every program. The first few addresses in the frame register stores the basic information of procedures and the subsequent stores the parameters and local variables in the procedure.

When a “ENTER” in the intermediate code which means a procedure call occurs, the frame register for the callee has to be set up immediately. There are several steps to be fulfilled when setting up a frame register as discussed in the earlier chapters. Normally, to set up a frame register, the number instructions are consumed can be stated as: 32 + 2 * (number of parameters + number of local variables). It means that to set up a frame register for a procedure, at least 32 instructions (the procedure consists of no parameters and local variables) are used in a SAFA program.

To set up frame registers for procedures, a “penter” instruction is delivered to the assembler, and the assembler generates the SAFA instructions that are required. According to the definition of SAFA, for all procedures, except main procedure, before entering to run the procedures, the information of the frame registers that are used to store the procedures must be created proactively in the main procedure.

The overheads for setting up frame registers are most related to parameters and local variables that the procedures have. For each parameter or local variable, two more instructions will be consumed in the SAFA program.

Changing of frame register occurs when the program needs to switch from one frame register to another, e.g. a function call occurs in the main function in the C program, e.g. LCG(array,n,1277,0,131012);. It will lead to a serial of activities and the current frame register will be modified to LCG’s to continuing running. Typically, to modify frame register, several instructions will be involved:

- Store the information on the stack to current frame
- Set current frame register to own frame register

These two activities are fulfilled by two instructions in SAFA program. The overhead for changing frame register is two instructions.

5.1.2 Modifying Frame Register

The most representative operation of modifying frame register is the activities that happen on elements in array, e.g. insert element, modify element, etc. As discussed in the previous chapter, the array in SAFA is implemented by frame register. The frame register that is assigned to store array information is set up in the procedure when it is defined. To insert new elements into an array, the following activities will be involved:

- Set current frame number
- Store frame stack to current index
- Store element to memory
- Set current frame to own

Four SAFA instructions are consumed to achieve the activities above. These instructions insert the element into the array, which will lead to modify the frame register.

For comparison, Java's stack is used to store parameters and results of bytecode instructions, to transfer parameters to and return values from methods, and to keep the state of each method invocation. The state of a method invocation is called its stack frame. The vars, frame, and optop registers point to different parts of the current stack frame.

There are three sections in a Java stack frame: the local variables, the execution environment, and the operand stack. The local variables section contains all the local variables being used by the current method

invocation. It is pointed to by the vars register. The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the frame register. The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the optop register [24].

Although there are some differences between Java bytecode and SAFA instructions, some of the mechanism is still the same. Compared to those in Java, setting-up, changing and modifying frame registers in SAFA nearly need the same overheads.

5.1.3 Array

SAFA has a special mechanism to deal with array as described in the previous chapter. The information that needs to store in the frame register consists of index, interval, base, limit and size. When creating an array, the information that should provide to the frame register is the base, size and limit of the array which cost three words. Normally, array is stored as just a pointer in the programming language in C, which consumes just 1 word.

Although the implementation of array in SAFA needs more hardware space, it brings much benefit:

The hardware knows the upper bound of array. The limit sector in defining an array in SAFA can tell how big the array can be. This is of a significant

differentiation from C. In SAFA, the upper bound of an array is stored in hardware level. When the upper bound of the array reaches, an overflow warning feedback will be returned. Additionally, the enquiry of the size of the array is also straightforward.

The size of the elements is stored in hardware level in SAFA machine. By the mechanism, the frame register for storing the array information can proactively assign enough space for the array. In C, the array is represented as pointer and the space for storing array is created at runtime, which means, unless an operation on the array comes, the array will always be presented as a pointer, no matter the space for the array is ready or not. If the memory of the machine is not big enough, the problem of overflow will occurs if there is no enough space for storing the array. In SAFA the problem is much easier to handle, because the space of the array is set up when it is created.

5.2 Context-Sensitive Frame Register

Considering SAFA design, the objective of frame registers is to use frame registers frequently with relatively wasting little effort in managing them, which can also be stated as a context-sensitive frame register design. Each current context is used for extended periods and resetting the context occurs naturally [2]. A more practical statement of the idea is when the operation which leads to the change of frame register for processing, the change of frame register occurs, and when the operation

is fulfilled the current frame register will automatically point back to the original frame register.

The significant benefit of the context-sensitive design is that the cost of switching from one frame register to another is highly decreased. An obvious example is to multiply two arrays which are of the same size and store the result in the first array. Suppose we have two arrays each of which has 100 elements. The C program to solve the problem is shown below (Figure 5-1).

```
void LCG(int ia[], int n, int a, int c, int m)
{
    int i,seed=1;

    for (i = 0; i < n; i++){
        seed = (a*seed + c) % m;
        ia[i] = seed;
    }}
void main()
{
    int array1[100], array2 [100], i;
    LCG(array1,100,16807,0,214748);
    LCG(array2,100,1277,0,131012);

    for (i = 0; i < 100; i++) array1[i] =array[1]*array[2];

}
```

Figure 5-1 Sample C Program for Context-Sensitive Frame Register

To discuss the mechanism clearly, we use the SAFA assembly code (Figure 5-2) for further discussion.

```
PROC LCG 5 2  //Procedure LCG
SAVEFRM 5 x58   //Save frame at address x58
ibload x24
loadnextfrm
cfset4
cfinfostore
```

```
cfsetown
ibload 1
cfb_wstore x34
ibload 0
cfb_wstore x38
loop: cfb_wload x38
cfb_wload x30
ige
iftrue end
cfb_wload x34
cfb_wload x44
imul
cfb_wload x38
iadd
cfb_wload x3c
idiv
dup
cfb_wstore x34
cfb_wload x38
cfset5
idxstore
frstore5
pop
cfsetown
cfb_wload x38
inc
cfb_wstore x38
goto loop
end: RESTOREFRM 5 x58
exit 1,2

PROC main 0 3  //procedure main
ibload 100
cfb_wstore x24
cfb_wload x24
ibload 4
newarray    //create array1
cfset4    //use frame register 4
cfinfostore
cfsetown
cfb_wload x24
ibload 4
newarray   //create array2
cfset5  // use frame register 5
cfinfostore
cfsetown
iblod 0
cfb_wstore x50
ibload 1
cfb_wstore x54
cfset4   //array1
cfinfoload
cfbload x24
iwload 16807
```

```
ibload 0
iwload 214748
penter 3,LCG  //enter procedure LCG
cfset5  //array2
cfinfoload
cfb_wload x24
ihwload 1277
ibload 0
iwload 131012
penter 3,LCG  //enter procedure LCG
loop: cfb_wload x40 //loop for multiplying array1 and array2
cfb_wload 24
dec
ige
iftrue end
cfbload x50
cfset5
idxstore
frload5
cfsetown
cfb_load x50
cfset4
idxstore
frload4
cfsetown
imul
idxstore
frstore4
pop
cfsetown
cfb_wload x50
inc
cfb_wstore x50
goto loop
end: halt
```

Figure 5-2 SAFA Assembly Code for Context-Sensitive Frame Register

As we can see from the example, in each iteration the elements in the two arrays are multiplied and the result is stored in array1. There's no need to set the frame register back to array1 for preceding the actions. This mechanism saves nearly half of the running cost of the program. To make comparison to the program that with the context-sensitive frame register mechanism, we put an instruction particularly to set the frame register back. Further more, we compare the time with the same program that is running in GCC and same function program in a static JVM (with javac

execution). We have the result as shown in Figure 5-3. The result is based on average.

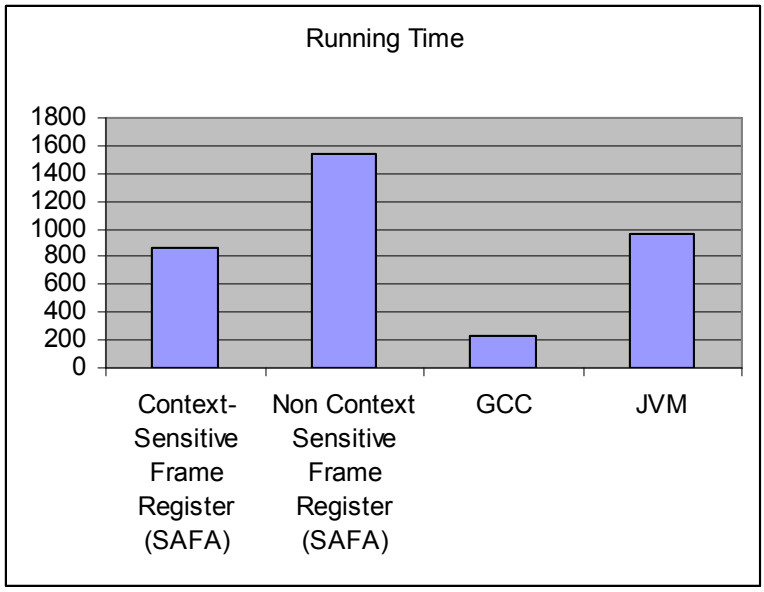


Figure 5-3 Cost Comparison (milliseconds)

Clearly, without the context-sensitive frame register mechanism, the operations among frame registers will cost much time on setting up, loading, saving frame register information and switching among frame registers. Comparing the two context we set for the comparison within SAFA, the cost saving is distinguishably obvious.

The objective of SAFA that each current context is used for extended periods and resetting the context occurs naturally is proved to be achieved. Further discussion will be covered in the subsequent chapters.

The results we get here compared to JVM and GCC will be further discussed in the next chapter.

Chapter 6

Performance Evaluation of C to SAFA Compiler

In this chapter, we firstly give a sample of C to SAFA compiler by giving a problem which is solved by a C Language program, and giving the assembly code, as well as the SAFA program generated.

Regarding the performance evaluation, we give some programs that are compiled by C to SAFA compiler, based on which the performance of the compiler will be demonstrated.

6.1 A Practical Sample of C to SAFA Compiler

6.1.1 Source Program – C Language Program

The C program for solving the Sieve Algorithm is shown in Figure 6-1.

```
void Sieve(int ia[], int n)
{
    int i,curPrime,mul;
    for (i = 0; i < n; i++)
        ia[i] = 1;
     curPrime = 2;
    while (curPrime < n)
    {
        for (mul = curPrime*2; mul < n; mul+=curPrime)
            ia[mul] = 0;
        curPrime++;
        while (curPrime < n){
            if (ia[curPrime] != 0)
                break;
            curPrime++;}
    }
}
void main()
{
    int n,i;
    n=10
    int ia [10];
    Sieve(ia,n);
    for (i = 2; i < n; i++)
```

```
    {
        if (ia[i] != 0)
            printf (" %d",i," ");
    }
}
```

Figure 6-1 C Program for Sieve

The C program has two functions, which are main and sieve. The main function is the compulsory function of a C program. The function sieve is the implementation of the algorithm. Additionally, the sieve function is called in the main function.

6.1.2 Assembly Code

Before generating the SAFA program, we get the assembly code as shown in Figure 6-2. Clearly, we can see the two procedures in the assembly code, which are corresponding to the two functions in the C program.

```
 PROC Sieve 2 3  //Procedure Sieve with two parameters and three local
                    variables
    SAVEFRM 4 x40  //save stack frame (address x40)
    RESTOREFRM 4 x24 restore frame stack (address x24)
    ibload 0  //load 0
    cfb_wstore x34  //store word (address x34)
    ibload 2  //load 2
    cfb_wstore x38 //store word (address x38)
forLoop:   //for Loop
    cfb_wload x34  //load word (address x34, actually this is load the
                        value of n in C)
    cfb_wload x30
    ige   //greater or equal
    iftrue whileLoop //if true go to whileLoop
    cfb_wload x34
    cfset4  //set current frame No. 4
    idxstore  store frame stack to current frame index
    ibload 1  //load 1
    frstore4  //store element to memory, this is to put the value in the
            corresponding element in the array
    cfsetown  //set current frame to own
    cfb_wload x34
    inc
```

```
    cfb_wstore x34
    goto forLoop  //go back to forLoop
whileLoop: cfb_wload x38  //while loop
    cfb_wload x34  //load (address x24, value is n)
    ige
    iftrue end
    cfb_wload x38
    ibload 2
    imul  //multiply
    cfb_wstore x3c
innerFor: //for loop in the while loop
cfb_wload x3c
    cfb_wload x34
    ige
    iftrue innerForEnd
    cfb_wload x3c
    cfset4
    idxstore
    ibload 0
    frstore4
    cfsetown
    cfb_wload x3c
    cfb_wload x38
    iadd
    cfb_wstore x3c
    goto innerFor
innerForEnd: cfb_wload x38
    inc
    cfb_wstore x38
innerWhile: //while loop in the while loop
cfb_wload x38
    cfb_wload x24
    ige
    iftrue whileLoop
    cfb_wload x38
    cfset4
    idxstore
    frload4
    cfsetown
    ifne whileLoop
    cfb_wload x38
    inc
    cfb_wstore x38
    goto innerWhile
end: RESTOREFRM 4 x40  //restore the stack frame (address x40)
    exit 1,2

PROC main 0 3  //main procedure
    ibload 10  //load 10
    dup //duplicate the element on the top of the stack
    cfb_wstore x24  //store word (address x24)
    ibload 1
    newarray //create new array
    cfset4 //set current frame No 4. (this is for storing the information of array)
```

```
cfinfostore // Store information on stack into current frame
cfinfoload //load information from frame addressed from stack
cfsetown //set current frame to own
cfb_wload x24 //load word (address x24)
penter 3,Sieve //enter procedure Sieve (the parameters for Sieve are array
                    and n which are load in the last three instructions)
halt  //Terminate program
```

Figure 6-2 SAFA Assembly Program for Sieve

6.1.3 Target Program – SAFA Program

Processed by SAFA assembler, the assembly code is generated into SAFA program (Figure 6-3).

The SAFA program consists of three procedures. “5A FA” represents it is a SAFA program. There are “02” procedures in the program. The size of the three procedures is “00 38”, and “00 89” respectively. The first procedure is corresponding to the main function in the C program (the main procedure is always put in the first place in a SAFA program), while the second is corresponding to the sieve function in the C program.

1	5A	FA	02	0A	00	38	42	0A	D0	2D	24	42	01	B1	04	3C
2	3A	2B	2C	24	46	00	13	00	08	03	44	04	00	34	2D	30
3	2D	2C	2D	28	2D	24	2B	3A	27	2D	10	2D	0C	2D	08	28
4	3A	27	2D	1C	2D	18	2D	14	28	2C	04	03	B0	E0	00	89
5	04	3A	2B	2D	48	2D	44	2D	40	2B	42	24	38	04	3C	2B
6	42	00	2D	34	42	02	2D	38	2C	34	2C	30	5C	81	11	2C
7	34	04	23	42	01	1C	2B	2C	34	56	2D	34	9E	EC	2C	38
8	2C	34	5C	81	3C	2C	38	42	02	54	2D	3C	2C	3C	2C	34
9	5C	81	13	2C	3C	04	23	42	00	1C	2B	2C	3C	2C	38	50
10	2D	3C	9E	EA	2C	38	56	2D	38	2C	38	2C	24	5C	81	D0
11	2C	38	04	23	14	2B	8E	C8	2C	38	56	2D	38	9E	EC	2B
12	42	40	38	04	3C	2B	29	2C	20	3A	C3	D0	42	14	50	02
13	3E	3C	42	08	50	01	3E	3C	B8							

Figure 6-3 SAFA Program for Sieve C Program

In the first procedure, the value of n (=10) is loaded by “42 0A”. “D0 2D 24” is to duplicate the value of n and store it at address x24. “B1” is for

creating a new array. Then the current frame is set to No. 4, the information on the stack is stored to the current frame (No. 4) (“3C”. After loading information from frame addressed from stack (“3E”), the current frame is set to own frame (“2B”). The value of n is loaded by “2C 24” from the address that is just stored. The two parameters that the procedure Sieve need are ready now, which are the array and the value of n. Then the stack frame for the procedure Sieve is established by the left instructions before the entering of the procedure (“B0” in the 4th row”). “E0” represents halting the program. From “00 89” in the 4th row are the instructions for procedure Sieve in the C program. “B0” in Line 13 means exit the procedure.

6.2 Applications

To measure floating-point and arithmetic performance, we select the C version of the Linpack benchmark [29]. Linpack is a collection of subroutines that analyze and solve linear equations and linear least-squares problems. We employ a simplified Linpack to perform the test for C to SAFA compiler. The source code for the applications is in Appendix B.

Referring the six tests which is widely used and is contained in the CaffeineMark for Java, we do the same test for C to SAFA compiler. However, the test is for Java, and we select four of them and develop the C program for each test. The four tests include:

Sieve: The canonical sieve algorithm, which is to solve problem on prime numbers.

Loops: Uses sorting and sequence generation to measure compiler optimization of loops. We select quick sort and bubble sort as the testing program.

Method: Execute recursive function calls to evaluate method invocation efficiency. We select Fibonacci and Hanoi problems as the test involved.

String: Performs basic string manipulations. We select the classical Knuth-Morris-Pratt string matching algorithm to develop a C program to fulfill this test.

6.3 Evaluation Methodology

The test is performed on a 1.8GHz Pentium 4 running Cygwin on Windows XP with 512MB RAM. This platform allows us to run the gcc, lcc, java runtime and C to SAFA compiler.

We obtain the test results by running the same C program in gcc and C to SAFA compiler and the corresponding program in JVM. Although the source program for running in JVM is Java program, we try to write the programs in static form which are quite similar to the C program. JVM is a stack architecture machine and Java byte code is relatively quite similar to SAFA code, which is the reason why we select JVM as a candidate to make comparison with C to SAFA compiler.

6.4 Evaluation of Target Code Size

We run the seven test programs on the environment presented in the previous section to gather the information for the target code size.

	C to SAFA Compiler	GCC	JVM
Linpack	2065	13200	3800
Sieve	201	3670	1009
Bubble Sort	321	3510	949
Quick Sort	486	3660	1240
Fibonacci	151	2650	351
Hanoi	162	3580	987
KMP	277	4520	1578

Figure 6-4 Code Size Comparison among Compilers (bytes)

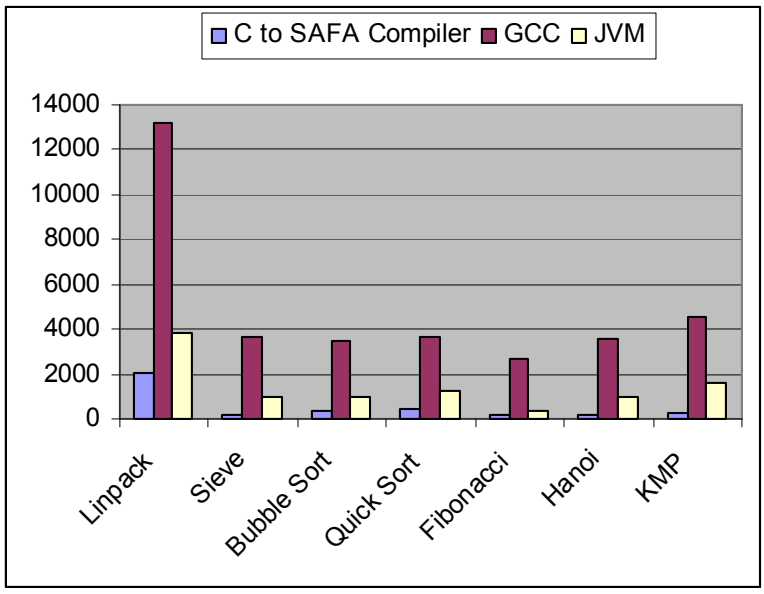


Figure 6-5 Comparison of Code Size among Compilers (bytes)

From a generally view of the comparison figure (Figure 6-5) target code size (in bytes) of gcc (without optimization), which can be a representative of general purpose register machines, is of the worst code size. Among the target code size of each machine, the target code size of gcc is of much more code size than that in C to SAFA compiler and in JVM. The result reflects the specialty of concise target code size of stack architectures as referred in the early chapter.

However, compared to JVM, which generates Java bytecode as the target code, C to SAFA compiler has a better performance which is generally, one third of the target code size compared to the target code generated by JVM. We have a promising result on the target code size, because the comparison between two stack architecture based machine are more reasonable. Compared to JVM, SAFA has some special features that help it on its better performance on code size, such as the support for high level programming, the design of frame register and context-sensitive. Anther issues can be considered is that Java is relatively more complicated a language compared to SAFA and the compilation produces some additional codes for some of its special features, such as its methods, objects, etc. Although the factors of differentiation on source program and the including of libraries for Java programs have influences on the result, the distinguished disparity is still presented.

6.5 Evaluation of Compilation Performance

The time referred in Figure 6-6 represents the comparison of performance of among C to SAFA compiler, GCC and JVM. The time which is tested on JVM is based on javac command to compile the source code. The time (in milliseconds) which is tested for C to SAFA compiler and gcc are all based on a full compilation process. The time consists of system time and user time. Since GCC, C to SAFA compiler and JVM generate target code for different types of machines, we do comparison in two dimensions, which are the comparison between general purpose register machines and stack architecture based machines, and the comparison between JVM and C to SAFA compiler, which are both stack based architecture based machines.

	C to SAFA Compiler	GCC	JVM
Linpack	6200	3200	5800
Sieve	2500	540	3900
Bubble Sort	1000	600	1200
Quick Sort	1300	780	1600
Fibonacci	2800	370	2500
Hanoi	2900	680	3200
KMP	3800	900	3900

Figure 6-6 Compilation Performance Comparison among Compilers (milliseconds)

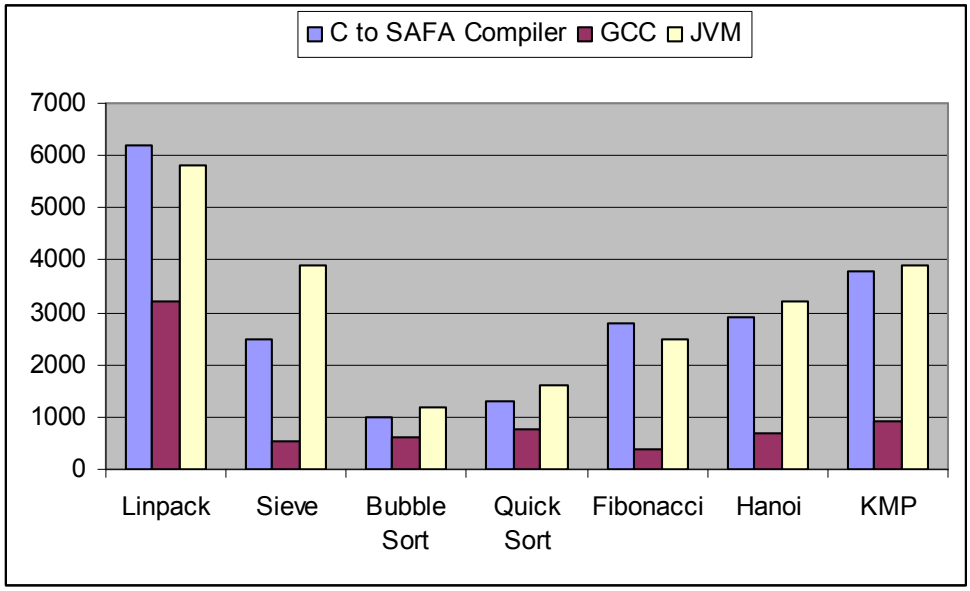


Figure 6-7 Compilation Performance Comparison among Compilers (milliseconds)

Comparing the performance of GCC with C to SAFA compiler and java on JVM, we have some indications on differentiation of the performance between register machine and stack architecture based compilers:

1. Increased Memory Use

Stack Architecture based programs use much more memory of comparable C++ programs to store the data. The transformation of information between memory and stack is very frequent due to the mechanism of the stack architecture. A larger memory footprint increases the probability that parts of the program will be swapped out to the disk. And swap file usage kills the speed like nothing else.

One of the significant benefits of GCC is memory locality. Newly allocated memory is adjacent to the memory recently used, and it is more likely to already be in the cache. One rather dated (1993) example shows that the

cache missing can be big cost: changing an array size in small C program from 1023 to 1024 results in 17 times slower. Although the effect is not that bad normally, with processor speeds increasing faster than memory, missing cache is probably an even bigger cost.

SAFA's context-sensitive frame registers try to achieve better performance by using frame registers frequently but waste little effort in managing them, in the hope that each current context is used for extended periods and re-setting the context occurs naturally.

2. Time Consumption in Compiling Processes

In the compilation process, the allocation of stack frames, the maintenance of frame registers, as well as the cooperation with SAFA assembler cost much time in C to SAFA complier. And regarding JVM, Java program's startup is rather slow. As a java program starts, it unzips the java libraries and compiles parts of itself, so an interactive program can be sluggish for the first couple seconds of use.

3. Lack of Optimization

Optimization of stack machine code received quite a little attention among the techniques of compiler. Little work has been done upon the intra block optimization. Global optimization is quite hard for a stack machine code because modern stack architecture machines are designed to operate the machine code based on procedures/ blocks, which makes it difficult to be optimized globally.

The comparison between C to SAFA compiler and JVM is more reasonable on the compilation performance. Because GCC and the two are of different architecture that the compilation processes are quite different, the comparability is not that high. C to SAFA compiler and JVM are both stack architecture based and the target codes of the both two are bytecode format.

As we can get from Figure 6-6, the performance of C to SAFA compiler is relatively higher than JVM in most cases. C to SAFA compiler has less optimization processes than JVM, this may be one of the most important reason why the performance is presented better than JVM, because more processes of optimization lead to more analysis and checks of the codes during the compilation, where SAFA does less than JVM. Anther issue that can be considered is the source code. The JAVA source code is relatively more complicated than C source code. Although we try to develop all the functions in the programs for testing in Java static, the including of libraries, etc and the initialization of the compilation cost more time.

6.6 Evaluation of Target Code Running Time

To evaluate the execution time of the target code size, we run the same test for both SAFA code and Java byte code. Regarding the results we get as shown in Figure 6-8 and Figure 6-9, it suggests that the execution time of target code on emulated SAFA is slightly better that that of JVM. The result shows the promising result of SAFA and SAFA code.

	C to SAFA Compiler	JVM
Linpack	2100	2250
Sieve	850	1050
Bubble Sort	200	300
Quick Sort	200	350
Fibonacci	450	600
Hanoi	550	550
KMP	750	800

Figure 6-8 Running Time Comparison (milliseconds)

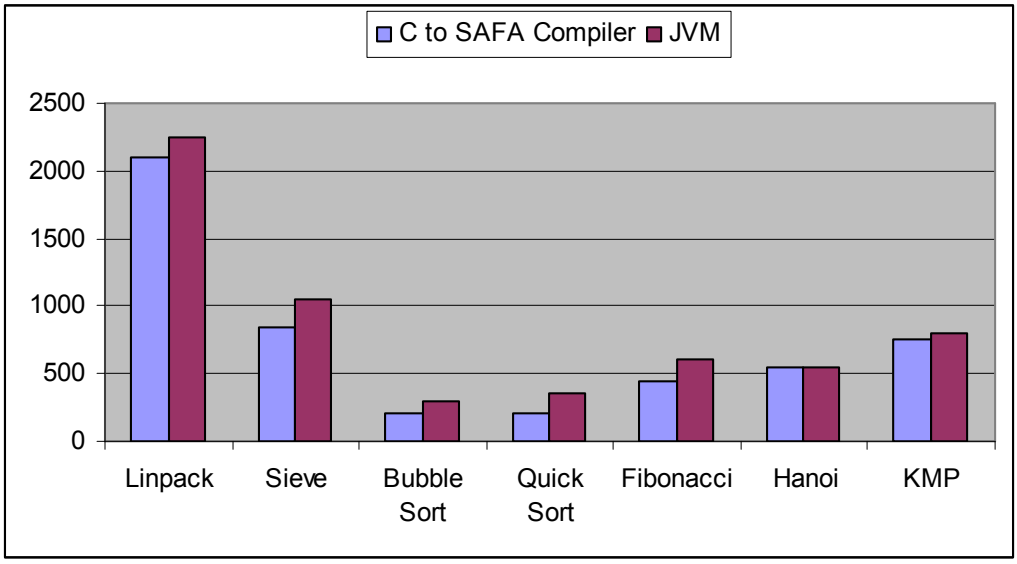


Figure 6-9 Running time Comparison (milliseconds)

Consequently, the compilation performance of C to SAFA compiler is promising. There are three issues that add value of the good performance:

- Hand-written lexical analyzer and syntax analyzer: The hand written lexical analyzer and syntax analyzer which are based on LCC save

much time for the compilation. A non-satisfactory lexical analyzer will cost half of the compilation time of the compiler [7].

- The good intermediate code which is relatively the similar to the assembly code: The intermediate code that is generated by C to SAFA compiler is quite similar to the SAFA assembly code not only on mechanism but also on format. This makes the generation from intermediate code to assembly code straightforward. The cooperation of the two phases has quite positive influence.
- The usage of special structures and mechanism within intermediate code generation: The several special structures that are to recode the necessary information for not only intermediate code generation but also the transformation of necessary information to the assembler makes the compiler more effective.

Chapter 7

Conclusion

7.1 Conclusion of C to SAFA Compiler

With concentrating on various aspects of compilation related topics, our work has ranged far and focused on design and implementation of C to SAFA Compiler which is to compile C program to corresponding SAFA program.

The thesis works is initiated with splitting the compiler into front end and back end after considering the common compilation techniques and the situation of C to SAFA compiler. However, we decided to put the intermediate code in the back end of the compiler because of its high dependency on the mechanism of target machine, SAFA. The front end of C to SAFA compiler is designed and implemented based on LCC which is presented in reference [7] after considering the performance of the compiler. The back end of the compiler is designed and implemented completely based on the requirement of SAFA instructions and program operation mechanism. We demonstrate some structures that are specially designed for C to SAFA compiler to serve for the compiler to fulfill the intermediate code generation. Further more, the stack scheduling is employed as the intermediate code optimization methodology and is proved to improve the code quality. The correspondence of the intermediate code and SAFA assembly code is very tight and drives us a very good opportunity to generate the intermediate code into the assembly code very sufficiently and conveniently. The SAFA program is

consequently generated after processed by the SAFA assembler. The SAFA program we generate is proved to be correct. The performance of the compiler is presented in the last chapter of the thesis. Compared to a Java compiler and a C compiler, the C to SAFA compiler shows a promising performance.

7.2 Future Work

We have given a sufficient implementation of C to SAFA compiler. Although we have acquired some achievements, we still need to consummate it. In the future the following efforts may be further taken into thoughts:

1. SAFA has a special design on dealing with record array. The idea is to implementing the record array in the frame register and dealing with special instructions. E.g., to visit a record in the array, it is can achieve by visiting the base and interval of the index; to visit an element in a special record, it is can achieve by loading the base, interval and offset without changing any other information. This mechanism makes the implementation of record array very convenient. However, it is hard to use compiler to compile effective SAFA instructions for this mechanism.
2. The optimization methodology is based on the idea of stack scheduling. An improved stack machine code optimization method, which is based on the Optimal DAG scheduling [21], may be considered to employ. However, there are some drawbacks in this method that is difficult to be implemented on C to SAFA compiler.

3. Another idea that can be considered is to compiler the Java code into SAFA code. There are two alternatives:

Based on the structure in this paper to compile the Java code into SAFA assembly code and let the SAFA assembler compile the target code. However, there are some difficulties because Java is an object-oriented language. If the Java program is static based, the compilation is very similar to C to SAFA compiler. But, if the Java program has objects, further consideration has to be taken.

Another idea is to write a cross-assembler to compile the Java assembly code into SAFA assembly code. This can be seen as a better alternative, because with Java compiler, it is easy to compiler the Java code into Java assembly code, and with SAFA assembler, it is easy to compiler the SAFA assembly code into SAFA code. To connect the serious of actions together, it may be a better solution to compile the Java program into SAFA program.

Bibliography

[1] Philip J. Koopman, Jr., Stack Computers: The New Wave, pages 15-48, P. Koopman/ Ellis Horwood Limited, 1989

[2] Yuen Chung Kwong, The Virtual Register Mapping Problem: Can We Combine Superscalar And EPIC Ideas?, National University of Singapore, 2000, http://www.comp.nus.edu.sg/~yuenck/stack

[3] Soo Yuen Jien, PhD Dissertation, National University of Singapore, 2004

[4] Henk Alblas and Albert Nymeyer, Practice and Principles of Compiler Building with C, pages 4-5, Prentice Hall, 1996

[5] Daniel L. Miller, Stack Machines and Compiler Design, Byte Magazine, pages 177-185, April, 1987

[6] John L. Hennessy, David A. Patterson, David Goldberg, Computer Architecture: A Quantitative Approach, 3rd Edition, pages 23-92, San Francisco, Morgan Kaufmann Publishers, 2002

[7] Christopher Fraser, David Hanson, A Retargetable C Compiler for ANSI C, ACM SIGPLAN Notices, Volume 26, No. 10, 1991

[8] Dick Grune, Henri E. Bal, Ceriel J.H., Jacobs & Koen G. Langendoen, Modern Compiler Design, pages 79-98, John Wiley &Sons, 2000

[9] Christopher Fraser, David Hanson, A Retargetable C Compiler: Design and Implementation, The Benjamin/Cummings Publishing Company, Inc., 1995

[10] W. M. Waite, The Cost of Lexical Analysis, Software: Practice & Experience, 16 (5): pages 473-488, 1986

[11] C. W. Fraser, D.R. Hanson, A Code Generation Interface for ANSI C, Software: Practice & Experience, 1991

[12] Martine Maierhofer, M. Anton Ertl, Optimizing Stack Code, Forth-Tagung 1997, Ludwigshafen, 1997

[13] Kyle Hayes, Parrot Virtual Machine and Register vs. Stack Machines, http://lists.tunes/org, 2002

[14] Philip Koopman, Jr., A Preliminary Exploration of Optimized Stack Code Generation, Journal of Forth Applications and Research, 6(3) pages 241-251, 1994

[15] J. L. Bruno, T. Lassagne, The Generation of Optimal Code for Stack Machines, Journal of the Association for Computing Machinery, Vol. 22, No. 3, pages 382-396, July 1975

[16] John Aycock, Converting Python Virtual Machine Code to C, University of Victoria, 2000, http://www.foretec.com/python/workshops/1998-11/proceedings/papers/aycock-211/aycock211.html

[17] Nisan, Schocken, The Elements of Computing Systems, pages 128-134, MIT Press, 2003

[18] M. Anton Ertl, Christian Pirker, The Structure of a Forth Native Code Compiler, EuroForth'97 Conference Proceedings, pages 107-116, 1997

[19] M. Anton Ertl, Martin Maierhofer, Translating Forth to Efficient C, EuroForth'95 Conference Proceedings, 1995

[20] Philip Koopman, Usenet Nuggets, Why Stack Machines?, Computer Architecture News, Vol. 21, No.1, March 1993

[21] Mark Smotherman, Sanjay Krishnamurthy, P.S. Aravind,, David Hunnicutt, Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling, Proceedings of the 24th Annual International Symposium on Microarchitecture, ACM Press, pages 93-102, Albuquerque, New Mexico, Puerto Rico, 1991

[22] Hayes J., An Interpreter and Object Code Optimizer for a 32 Bit Forth Chip, 1986 FORMAL Conference Proceedings, pages 211-221, 1986

[23] Sun Micosystem Computer Company, The Java Virtual Machine Specification, 1995

[24] Bill Venners, The Lean, Mean Virtual Machine: The Basic Structure and Functionality of the Java Virtual Machine, Java World Magazine, May, 1996

[25] Andreas Krall, Efficient JavaVM Just-in-Time Compilation, International Conference on Parallel Architectures and Compilation Techniques, 1998

[26] Bruno J., Lassage T., The Generation of Optimal Code for Stack Machines, JACM, July 1975, pages 382-396, 1975

[27] Couch J., Hamm T., Semantic Structures for Efficient Code Generation on a Stack Machine, Computer, May 1977, 10(5), pages 42-48, 1977

[28] Performance of Various Computers Using Standard Linear Equations Software", Jack Dongarra, University of Tennessee, Knoxville TN, 37996, Computer Science Technical Report Number CS - 89 – 85, 2005

Appendix A: SAFA Instruction Set

OpCode	Pop	Push	Usage
********** Frame Registers Instruction **********			
<0x00> + 0	0	0	0x00 - 0x07 Set Current Frm No.
....			
<0x07> + 0	0	0	
<0x08> + 0	0	0	0x08 - 0x0f Compare current idx with idx
....			
<0x0f> + 0	0	0	
<0x10> + 0	0	1	0x10 - 0x17 Load Element to Stack (base+idx*size)
....			
<0x17> + 0	0	1	
<0x18> + 0	1	0	0x18 - 0x1f Store Element to Memory (frm xxx)
....			
<0x1f> + 0	1	0	
<0x20> + 0	0	0	0x20 Add curent frma inc to idx
<0x21> + 0	0	1	0x21 cmp idx to limit (idx -limit)
<0x22> + 0	0	0	0x22 subtract cur frame inc frm idx
<0x23> + 0	1	0	0x23 store frm stack to current idx
<0x24> + 0	0	1	0x24 load current frm idx to stack
<0x25> + 0	0	1	0x25 load current frm no to stack
<0x26> + 0	0	1	0x26 load previos frm no.
<0x27> + 0	0	0	0x27 switch current and prev frm no.
<0x28> + 0	0	0	0x28 set cur to global
<0x29> + 0	0	0	0x29 set cur to caller
<0x2a> + 0	0	0	0x2a set cur to host
<0x2b> + 0	0	0	0x2b
<0x2c> + 1	0	1	0x2c load word to stack, cur base+byte
<0x2d> + 1	1	0	0x2d store word, cur base+ byte
<0x2e> + 2	0	1	0x2e load word to stack, cur base+hword
<0x2f> + 2	1	0	0x2f store word, cur base+hword
<0x30> + 4	0	0	0x30 change to new base, clear idx
<0x31> + 4	0	0	0x31 change to new base, idx = limit
<0x32> + 2	0	0	0x32 add halfword to cur base
<0x33> + 1	0	0	0x33 add byte to cur base
<0x34> + 0	2	0	0x34 stack a new frame
<0x35> + 0	0	0	0x35 pop frm
<0x36> + 0	0	0	0x36 chain frm (get new base frm base+B)
<0x37> + 0	0	0	0x37 chain frm (get new base frm base+H)
<0x38> + 0	0	0	0x38 load next frm base to stack
<0x39> + 0	0	0	0x39 load new frm base (current frm to stack)
<0x3a> + 0	0	3	0x3a Load cur frm info to stack
<0x3b> + 0	0	0	0x3b "", set idx = 0
<0x3c> + 0	3	0	0x3c Store info on stack into cur frm
<0x3d> + 0	2	0	0x3d ", idx = limit
<0x3e> + 0	0	0	0x3e load info from frm addressed from stack
<0x3f> + 0	0	0	0x3f make frm using frm info on stack
********** Load/Store to Stack Instruction **********			
<0x40> + 0	0	1	0x40 LDI False
<0x41> + 0	0	1	0x41 LDI True
<0x42> + 1	0	1	0x42 LDI B, leading 0

```
<0x43> + 1    1       1           0x43 LDI B, no change
<0x44> + 2    0       1           0x44 LDI H, leading 0
<0x45> + 2    1       1           0x45 LDI H, no change
<0x46> + 4    0       1           0x46 LDI W
<0x47> + 8    0       2           0x47 LDI DW
************************* Load/Store to Memory Instruction ***************************
<0x48> + 4    0       1           0x48 - 0x4b Load from full address
<0x49> + 4    0       1                Size : B,H,W,DW
<0x4a> + 4    0       1
<0x4b> + 4    0       1
<0x4c> + 4    1       0           0x4c - 0x4f Store to full address
<0x4d> + 4    1       0                Size : B,H,W,DW
<0x4e> + 4    1       0
<0x4f> + 4    1       0
************************* Integer Arithmetic Instruction *****************************
<0x50> + 0    2       1           0x50  Add Integer Word
<0x51> + 0    4       2           0x51  Add Integer Double Word
<0x52> + 0    2       1           0x52  Sub Integer Word
<0x53> + 0    4       2           0x53  Sub Ingeger Double Word
<0x54> + 0    2       2           0x54  Mul Word => Double Word
<0x55> + 0    2       2           0x55  Div Word => quotient,remainder
<0x56> + 0    1       1           0x56  Increment
<0x57> + 0    1       1           0x57  Decrement
************************* Integer Comparison Instruction *****************************
<0x58> + 0    2       1           0x58  EQUAL
<0x59> + 0    2       1           0x59  NT EQUAL
<0x5a> + 0    2       1           0x5a  GREATER
<0x5b> + 0    2       1           0x5b  LESSER
<0x5c> + 0    2       1           0x5c  GREATER R EQUAL
<0x5d> + 0    2       1           0x5d  LESSER R EQUAL
<0x5e> + 0    2       1           0x5e  SAME SIGN
<0x5f> + 0    2       1           0x5f  DIFF SIGN
************************* Float Arithmetic Instruction *******************************
<0x60> + 0    2       1           0x60       Add Float Word
<0x61> + 0    4       2           0x61       Add Float DWord
<0x62> + 0    2       1           0x62       Sub Float Word
<0x63> + 0    4       2           0x63       Sub Float Dword
<0x64> + 0    2       1           0x64       Mul Float Word
<0x65> + 0    4       2           0x65       Mul Float DWord
<0x66> + 0    2       2           0x66       Div Float Word
<0x67> + 0    4       2           0x67       Div Float DWord
************************* Float Comparison Instruction *******************************
<0x68> + 0    2       1           0x68  EQUAL
<0x69> + 0    2       1           0x69  NT EQUAL
<0x6a> + 0    2       1           0x6a  GREATER
<0x6b> + 0    2       1           0x6b  LESSER
<0x6c> + 0    2       1           0x6c  GREATER R EQUAL
<0x6d> + 0    2       1           0x6d  LESSER R EQUAL
<0x6e> + 0    2       1           0x6e  SAME SIGN
<0x6f> + 0    2       1           0x6f  DIFF SIGN
<0x70> + 0    4       1           0x70  DW Equal
<0x71> + 0    4       1           0x71  DW Not Equal
<0x72> + 0    4       1           0x72  DW Greater
<0x73> + 0    4       1           0x73  DW Lesser
```

```
<0x74> + 0    4       1           0x74  DW Greater or Equal
<0x75> + 0    4       1           0x75  DW Lesser or Equal
<0x76> + 0    4       1           0x76  DW Same Sign
<0x77> + 0    4       1           0x77  DW Diff Sign
<0x78> + 0    4       2           0x78  DW integer divide
************************** Boolean peration Instruction *******************************
<0x79> + 0    1       1           0x79      NEG INT (2s Complement)
<0x7a> + 0    2       1           0x7a      AND
<0x7b> + 0    2       1           0x7b      R
<0x7c> + 0    2       1           0x7c      R
<0x7d> + 0    2       1           0x7d      EQ
<0x7e> + 0    1       1           0x7e      INVERT
<0x7f> + 0    2       1           0x7f      MASK
************************** Branch/Flow Control Instruction ****************************
<0x80> + 1    1       0           0x80 BR F + byte offset
<0x81> + 1    1       0           0x81 BR T + byte offset
<0x82> + 2    1       0           0x82 BR F + halfword offset
<0x83> + 2    1       0           0x83 BR F + halfword offset
<0x84> + 1    1       0           0x84 BGT int
<0x85> + 1    1       0           0x85 BGT double int
<0x86> + 1    1       0           0x86 BLT int
<0x87> + 1    1       0           0x87 BLT double int
<0x88> + 1    1       0           0x88 BGE int
<0x89> + 1    1       0           0x89 BGE double int
<0x8a> + 1    1       0           0x8a BLE int
<0x8b> + 1    1       0           0x8b BLE double int
<0x8c> + 1    1       0           0x8c BEQ int
<0x8d> + 1    1       0           0x8d BEQ double int
<0x8e> + 1    1       0           0x8e BNE int
<0x8f> + 1    1       0           0x8f BNE double int
<0x90> + 0    0       0           0x90
<0x91> + 0    0       0           0x91
<0x92> + 0    0       0           0x92
<0x93> + 0    0       0           0x93
<0x94> + 0    0       0           0x94
<0x95> + 0    0       0           0x95
<0x96> + 0    0       0           0x96
<0x97> + 0    0       0           0x97
<0x98> + 0    0       0           0x98
<0x99> + 0    0       0           0x99
<0x9a> + 0    0       0           0x9a
<0x9b> + 0    0       0           0x9b
<0x9c> + 0    0       0           0x9c
<0x9d> + 0    0       0           0x9d
<0x9e> + 1    0       0           0x9e BR byte offset
<0x9f> + 4    0       0           0x9f BR word offset
************************** Datatype Conversion Instruction ****************************
<0xa0> + 0    0       0           0xa0
<0xa1> + 0    0       0           0xa1
<0xa2> + 0    0       0           0xa2
<0xa3> + 0    0       0           0xa3
<0xa4> + 0    0       0           0xa4
<0xa5> + 0    0       0           0xa5
<0xa6> + 0    0       0           0xa6
```

```
<0xa7> + 0     0       0             0xa7
<0xa8> + 0     0       0             0xa8
<0xa9> + 0     0       0             0xa9
<0xaa> + 0     0       0             0xaa
<0xab> + 0     0       0             0xab
<0xac> + 0     0       0             0xac
<0xad> + 0     0       0             0xad
<0xae> + 0     0       0             0xae
<0xaf> + 0     0       0             0xaf
************************** Subroutine Instruction ***************************************
<0xb0> + 0     1       0             0xb0 Enter (take addr from stack)
<0xb1> + 0     0       0             0xb1
<0xb2> + 0     0       0             0xb2
<0xb3> + 0     0       0             0xb3
<0xb4> + 0     0       0             0xb4
<0xb5> + 0     0       0             0xb5
<0xb6> + 0     0       0             0xb6
<0xb7> + 0     0       0             0xb7
<0xb8> + 0     0       0             0xb8
<0xb9> + 0     0       0             0xb9
<0xba> + 0     0       0             0xba
<0xbb> + 0     0       0             0xbb
<0xbc> + 0     0       0             0xbc
<0xbd> + 0     0       0             0xbd
<0xbe> + 0     0       0             0xbe
<0xbf> + 0     0       0             0xbf
************************** Special Stack Instruction ************************************
<0xc0> + B + 0         1       1                0xc0 shift/arithmetic shift
<0xc1> + B + 0         1       1                0xc1 rotate/rotate with carry
<0xc2> + 0     0       0             0xc2 - 0xc7 Erase Word
....
<0xc7> + 0     0       0             0xc7
<0xc8> + 0     0       0             0xc8 Reverse Top 2 words
<0xc9> + 0     0       0             0xc9 Reverse Top 2 dwords
<0xca> + 0     0       0             0xca Cycle Top 3 dwords B to T
<0xcb> + 0     0       0             0xcb "" Top to Bottom
<0xcc> + 0     0       0             0xcc Cycle Top 3 words B to T
<0xcd> + 0     0       0             0xcd "" Top to Bottom
<0xce> + 0     0       0             0xce Cycle Top 4 words B to T
<0xcf> + 0     0       0             0xcf "" Top to Bottom
<0xd0> + 0     0       0             0xd0 - 0xd3 Replicate Top Word xx+1
....
<0xd3> + 0     0       0             0xd3
<0xd4> + 0     0       0             0xd4 - 0xd7 Replicate Top DWord xx+1
....
<0xd7> + 0     0       0             0xd7
<0xd8> + 0     1       1             0xd8 duplicate hw within 1 word
<0xd9> + 0     1       1             0xd9 quadruplicate byte within 1 word
<0xda> + 0     1       4             0xda split w into 4 bytes
<0xdb> + 0     1       2             0xdb split w into 2 halfwords
<0xdc> + 0     1       1             0xdc count 1 bits in byte
<0xdd> + 0     1       1             0xdd " in hw
<0xde> + 0     1       1             0xde " in w
<0xdf> + 0     1       1       0      0xdf " in dw
```

Appendix B: Applications

1. Simplified Linpack

```
double abs (double d) {
  return (d >= 0) ? d : -d;
}

 double matgen (double a[10][11], int n, double b[10])
{
  double norma;
  int init, i, j;

  init = 1325;
  norma = 0.0;

  for (i = 0; i < n; i++) {
     for (j = 0; j < n; j++) {

         init = 3125*init % 65536;
         a[j][i] = (init - 32768.0)/16384.0;
         norma = (a[j][i] > norma) ? a[j][i] : norma;
   }
  }

  for (i = 0; i < n; i++) {
    b[i] = 0.0;
  }
  for (j = 0; j < n; j++) {
    for (i = 0; i < n; i++) {
      b[i] += a[j][i];
    }
  }
  return norma;
}

 void daxpy( int n, double da, double dx[], int dx_off, int incx,
       double dy[], int dy_off, int incy)
{
  int i,ix,iy;

  if ((n > 0) && (da != 0)) {
    if (incx != 1 || incy != 1) {

  // code for unequal increments or equal increments not equal to 1

      ix = 0;
      iy = 0;

      if (incx < 0)
```

```
          ix = (-n+1)*incx;
      if (incy < 0)
          iy = (-n+1)*incy;
      for (i = 0;i < n; i++) {
        dy[iy +dy_off] += da*dx[ix +dx_off];
        ix += incx;
        iy += incy;
      }
      return;

    } else {

  // code for both increments equal to 1

      for (i=0; i < n; i++)
        dy[i +dy_off] += da*dx[i +dx_off];
    }
  }
}

 double ddot( int n, double dx[], int dx_off, int incx, double dy[],
        int dy_off, int incy)
{
  double dtemp;
  int i,ix,iy;

  dtemp = 0;

  if (n > 0) {

    if (incx != 1 || incy != 1) {

  // code for unequal increments or equal increments not equal to 1

      ix = 0;
      iy = 0;
      if (incx < 0) ix = (-n+1)*incx;
      if (incy < 0) iy = (-n+1)*incy;
      for (i = 0;i < n; i++) {
        dtemp += dx[ix +dx_off]*dy[iy +dy_off];
        ix += incx;
        iy += incy;
      }
    } else {

  // code for both increments equal to 1

      for (i=0;i < n; i++)
        dtemp += dx[i +dx_off]*dy[i +dy_off];
    }
  }
  return(dtemp);
}
```

```
/*
  scales a vector by a constant.
  jack dongarra, linpack, 3/11/78.
*/
 void dscal( int n, double da, double dx[], int dx_off, int incx)
{
  int i,nincx;

  if (n > 0) {
    if (incx != 1) {

  // code for increment not equal to 1

      nincx = n*incx;
      for (i = 0; i < nincx; i += incx)
        dx[i +dx_off] *= da;
    } else {

  // code for increment equal to 1

      for (i = 0; i < n; i++)
        dx[i +dx_off] *= da;
    }
  }
}

/*
  finds the index of element having max. absolute value.
  jack dongarra, linpack, 3/11/78.
*/
 int idamax(int n, double dx[], int dx_off, int incx)
{
  double dmax, dtemp;
  int i, ix, itemp=0;

  if (n < 1) {
    itemp = -1;
  } else if (n ==1) {
    itemp = 0;
  } else if (incx != 1) {

    // code for increment not equal to 1

    dmax = abs(dx[0 +dx_off]);
    ix = 1 + incx;
    for (i = 1; i < n; i++) {
      dtemp = abs(dx[ix + dx_off]);
      if (dtemp > dmax)  {
        itemp = i;
        dmax = dtemp;
      }
      ix += incx;
    }
  } else {
```

```
    // code for increment equal to 1

    itemp = 0;
    dmax = abs(dx[0 +dx_off]);
    for (i = 1; i < n; i++) {
      dtemp = abs(dx[i + dx_off]);
      if (dtemp > dmax) {
        itemp = i;
        dmax = dtemp;
      }
    }
  }
  return (itemp);
}

 double epslon (double x)
{
  double a,b,c,eps;
  a = 4.0e0/3.0e0;
  eps = 0;
  while (eps == 0) {
    b = a - 1.0;
    c = b + b + b;
    eps = abs(c-1.0);
  }
  return(eps*abs(x));
}

void dmxpy ( int n1, double y[10], int n2, double x [10], double m [10][11])
{
  int j,i;

  // cleanup odd vector
  for (j = 0; j < n2; j++) {
    for (i = 0; i < n1; i++) {
      y[i] += x[j]*m[j][i];
    }
  }
}

 void dgesl( double a [10][11], int n, int ipvt [10], double b [10])
{
  double t;
  int k,kb,l,nm1,kp1;
  nm1 = n - 1;
    if (nm1 >= 1) {
      for (k = 0; k < nm1; k++) {
        l = ipvt[k];
        t = b[l];
        if (l != k){
          b[l] = b[k];
          b[k] = t;
        }
        kp1 = k + 1;
```

```
        daxpy(n-(kp1),t,a[k],kp1,1,b,kp1,1);
      }
    }

    // now solve  u*x = y

    for (kb = 0; kb < n; kb++) {
      k = n - (kb + 1);
      b[k] /= a[k][k];
      t = -b[k];
      daxpy(k,t,a[k],0,1,b,0,1);
    }
 }

int dgefa(double a [10][11], int n, int ipvt [10])
 {
  double col_k[10], col_j[10];
  double t;
  int j,k,kp1,l,nm1;
  int info;

  // gaussian elimination with partial pivoting

  info = 0;
  nm1 = n - 1;
  if (nm1 >=  0) {
    for (k = 0; k < nm1; k++) {
      col_k[k] = a[k][k];
      kp1 = k + 1;

      // find l = pivot index

      l =  idamax(n-k,col_k,k,1) + k;
      ipvt[k] = l;

  // zero pivot implies this column already triangularized

      if (col_k[l] != 0) {

    // interchange if necessary

        if (l != k) {
          t = col_k[l];
          col_k[l] = col_k[k];
          col_k[k] = t;
        }

    // compute multipliers

        t = -1.0/col_k[k];
        dscal(n-(kp1),t,col_k,kp1,1);

    // row elimination with column indexing
```

```
        for (j = kp1; j < n; j++) {
          col_j[j] = a[j][j];
          t = col_j[l];
          if (l != k) {
            col_j[l] = col_j[k];
            col_j[k] = t;
          }
          daxpy(n-(kp1),t,col_k,kp1,1,
            col_j,kp1,1);
        }
      }
      else {
        info = k;
      }
    }
  }
  ipvt[n-1] = n-1;
  if (a[(n-1)][(n-1)] == 0)
     info = n-1;

  return info;
 }
 void run_benchmark()
 {
  double residn_result = 0.0;
  double eps_result = 0.0;
  double a[10][11];
  double b[10];
  double x[10];
  double norma,normx;
  double resid,time;
  int n,i,info;
  int ipvt[10];
  n = 10;
  norma = matgen(a,n,b);
  info = dgefa(a,n,ipvt);
  dgesl(a,n,ipvt,b);
  for (i = 0; i < n; i++) {
    x[i] = b[i];
  }
  norma = matgen(a,n,b);
  for (i = 0; i < n; i++) {
    b[i] = -b[i];
  }
  dmxpy(n,b,n,x,a);
}

 void main()
 { run_benchmark();  }
```

2. Sieve

```
void Sieve(int ia[], int n)
```

```
{
   int i,curPrime,mul;
   for (i = 0; i < n; i++)
      ia[i] = 1;
    curPrime = 2;
   while (curPrime < n){
      for (mul = curPrime*2; mul < n; mul+=curPrime)
         ia[mul] = 0;
      curPrime++;
      while (curPrime < n){
         if (ia[curPrime] != 0)
            break;
         curPrime++;}
   }
   }
void main()
{
   int n,i;
   int ia [22500];
   scanf ("%d", &n);
   printf("Sieving Array of Size ", n);
   printf;
   Sieve(ia,n);
   for (i = 2; i < n; i++){
      if (ia[i] != 0)
         printf (" %d",i," ");
   }}
```

3. Bubble Sort

```
void LCG(int ia[], int n, int a, int c, int m)
{
   int i,seed=1;

   for (i = 0; i < n; i++){
      seed = (a*seed + c) % m;
      ia[i] = seed;
   }
}
void BubbleSort(int ia[], int n)
{
   int change,i,tmp;
   do{
      change = 0;
      for (i = 0; i < n-1; i++){
         if (ia[i] > ia [i+1]){
            tmp = ia[i];
            ia[i] = ia[i+1];
            ia[i+1] = tmp;
            change = 1;
         }
      }
   } while (change != 0);
```

```
}
void main()
{
   int array[100];
   int i;
   LCG(array,100,1277,0,131012);
   BubbleSort(array,100);
}
```

4. Hanoi

```
move(char a,char c ,int *count)
{
       (*count)++;
       printf("%6d:%c-->%c\n",*count,a,c);
}

hanoi(int n, int a, int b, int c, int *count)
{
      if(n==1) move(a,c,count);
      else
        {
          hanoi(n-1,a,c,b,count);
          move(a,c,count);
          hanoi(n-1,b,a,c,count);
}
}
main()
{
      int n,count=0;
      n=10;
      hanoi(n,1,2,3 ,&count);
}
```

5. Quick Sort

```
void LCG(int ia[], int n, int a, int c, int m)
{
   int i,seed=1;
   for (i = 0; i < n; i++){
       seed = (a*seed + c) % m;
       ia[i] = seed;
   }
}
void QuickSort(int ia[], int start, int end)
{
   int i,j,tmp,midValue;
   i = start;
   j = end;
   midValue = ia[(start+end)/2];
```

```
    while (i <= j){
        while (ia[i] < midValue)
            i++;
        while (ia[j] > midValue)
            j--;
        if (i <= j){
            tmp = ia[i];
            ia[i] = ia[j];
            ia[j] = tmp;
            i++;
            j--;
        }
    }
    if (start < j)
        QuickSort(ia,start,j);

    if (i < end)
        QuickSort(ia,i,end);
}
void main()
{
    int array[100];
    int i,n;
    LCG(array,100,1277,0,131012);
    QuickSort(array,0,99);
    }
```

6. Fibonacci

```
int fibonacci(int x)
    {
        if (x == 0)
            return 0;
        if (x == 1)
            return 1;
        return fibonacci(x-1)+fibonacci(x-2);
    }

void main()
{
        int result, f;
        f = 10;
        result = fibonacci(f);
    }
```