Code Generation Method for Embedded Processors with Application Domain Specific Instruction Set

January 2008

Hiroaki TANAKA

Code Generation Method for Embedded Processors with Application Domain Specific Instruction Set

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2008

Hiroaki TANAKA

Publications

Journal Articles (Refereed)

[J1] Hiroaki Tanaka, Yoshinori Takeuchi, Keishi Sakanushi, Masaharu Imai, Hiroki Tagawa, Yutaka Ota and Nobu Matsumoto: "Generation of Pack Instruction Sequence for Media Processors Using Multi-Valued Decision Diagram," IEICE Trans. on Fundamentals of Electronics, vol. E90, no. 12, pp. 2800–2809, Dec, 2007.

[J2] Hiroaki Tanaka, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai: "A Code Optimization Technique for Processors with SIMD instructions Considering Permutation Instructions," IPSJ Journal (in Japanese, to appear).

International Conference Papers (Refereed)

[I1] Hiroaki Tanaka, Shinsuke Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai: "A Code Selection Method for SIMD Processors with PACK Instructions," in Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 66–80, Sep., 2003.

[I2] Hiroaki Tanaka, Yoshinori Takeuchi, Keishi Sakanushi, Masahau Imai, Yutaka Ota, Nobu Matsumoto and Masaki Nakagawa: "Pack Instruction Generation for Media Processors Using Multi-valued Decision Diagram," in Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 154–159, Oct., 2006.

[I3] Hiroaki Tanaka, Shiro Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai, "A Block-Floating-Point Procesoor for Rapid Application Development," in Proceedings of IEEE International Conference on Acoustics. Speech and Signal Processing 2007 (ICASSP2007), vol. II, pp65–68, 2007.

Domestic Conference Paper

[D1] Hiroaki Tanaka, Shinsuke Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai: "A Code Selection Method for SIMD Processors with PACK Instructions," Technical report of IEICE. VLD, vol. 103, no. 145, pp. 43–48, Jun., 2003 (in Japanese).

[D2] Hiroaki Tanaka, Hassan M. AbdElSalam, Shiro Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai, "Implementation fo 2D-FFT on a Block-Floating-Point DSP," IPSJ Symposium Series, vol. 2004, no. 8, pp. 91–96, Jul., 2004 (in Japanese)

[D3] Hiroaki Tanaka, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai, "A Code Generation Method for Processors with SIMD Instructions Considering Dependency and Distance between Operations," IPSJ Symposium Series, vol. 2005, no. 9, pp. 79–84, Aug., 2005 (in Japanese)

[D4] Hiroaki Tanaka, Shiro Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi and Masaharu Imai, "Implementation and Evaluation of an Instruction Set Processor with Herarchical Block-Floating-Point Arithmetic Capability," In the Proceedings of 21th Signal Processing Symposium, Nov., 2006 (in Japanese)

Summary

Application specific instruction set processors (ASIP) realize high cost-performance embedded systems. However, there is a difficult problem in software development for ASIPs. In typical software development for general purpose processors, programmers write programs in high-level programming language and then generate assembly code using compilers. On the other hand, in software development for ASIPs, programmers often have to write assembly code in some cases because compilers cannot efficiently generate application specific instructions in assembly code. Programming using assembly code requires much development effort and loses portability of programs. Therefore, there is a great need for a code generation technology to generate assembly code which brings out processors's performance.

This thesis discusses two types of code generation method for processors with application domain specific instruction set. First, this thesis proposes a code generation method with special functions, which are mapped into specific instructions, in high-level programming languages. To ease software development of the target application domain, programming scheme and code generation method for block-floating-point processors are proposed. Experimental results show the proposed code generation method successfully generates assembly code for block-floating-point processors and quality of generated code is good enough for practical applications.

The second code generation method is automatic generation of application specific instructions. To enable programmers to use application specific instructions without much programing effort, automatic utilization of application specific instruction is proposed. Media processors are selected as target processors in this thesis. Difficulties in compilation for media processors are extraction of operations to perform in parallel and to deal with positions of data in registers. Media processors have two types of instructions, SIMD instructions which operate on subword data in registers and permutation instructions which reorder or repack data in registers. Utiliza-

tion of both SIMD and permutation instruction is crucial to maximize the performance of media processors. In this study, code optimization problem for media processors is formulated into integer linear programming problem. Then, optimum and heuristic code generation methods are proposed and evaluated. Experimental results show the proposed methods generate assembly code with SIMD and permutation instructions. High performance improvement is shown by the proposed code generation methods.

Acknowledgements

I would like to thank my supervisor, Prof. Masaharu Imai, for his guidance throughout my undergraduate and graduate researches. I am especially grateful for his useful advice which develops my expertise in the research area and academic mentality.

I would like to thank Prof. Takao Onoye, Prof. Akihisa Yamada and Dr. Yoshinori Shigeta for a review of this thesis, useful discussion and comments on my study. Also, I would like to thank Prof. Onoye for treating me friendly throughout my student life. I enjoyed with his talks and thoughts very much.

I would like to thank Prof. Yoshinori Takeuchi. His comments greatly improved this thesis. I am also grateful for his guidance, support and encouragement through everything in my student life.

I would like to thank Prof. Keishi Sakanushi for his guidance, giving comments on my research activities.

I would like to thank Prof. Shinsuke Kobayashi and Mr. Kentaro Mita for introducing me to this research area and guiding my early study. They gave me instruction in research activity, presentation skills, programming skills, etc.

I would like to thank Dr. Yuki Kobayashi, Mr. Noboru Yoneoka and Mr. Tatsuhiro Yoshimura. They provided me good time through my days in Integrated System Design Laboratory in Osaka University. I always enjoyed all kinds of activities in the laboratory by them.

I thank Dr. Kyoko Ueda, Dr. Mohamed AbdElSalam Hassan, Miss. Yukako Nishikawa, Mrs. Motoko Higashide, Mr. Ittetsu Taniguchi, Mr. Takashi Hamabe, Mr. Hirofumi Iwato, Mr. Takuji Hieda, Mr. Takeshi Shiro, Mr. Takahiro Itoh, Mr. Akira Kobashi, Mr. Yu Okuno, Mr. Hitoshi Nakamura, Mr. Ayataka Kobayashi, Miss Aiko Watanabe, Mr. Kazuhiro Kobashi, Mr. HideYuki Okajima and other current and former members of Integrated System Design

Laboratory in Osaka University. They gave me a lot of comments on my study. They also made my life enjoyable with their talks and acts.

I would like to thank collaborators through my doctoral research. I would like to thank Dr. Shiro Kobayashi and Mr. Wang David from Asahi Kasei Corporation for giving discussion and comments on the study of the block-floating-point processor. They also helped me with implementation of the processor and programs for evaluation. I would like to thank Mr. Nobu Matsumoto, Mr. Yutaka Ota and Mr. Hiroki Tagawa from Toshiba Corporation. They provided me discussion and comments on the study of code generation for media processors. They also provided me tools for evaluation including compiler, simulator and so on.

Last, I would like to thank my father Yasuo, my mother Nobuko, and my brothers Katsuyuki and Masahiro. They gave me the encouragement and continuous support throughout my life in Osaka.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The progress of semiconductor manufacturing technology and design methodology of integrated circuits enable us to realize various electronic devices based on digital circuits. As integration scale of semiconductor grows, electronic devices can be manufactured in smaller size. The progress of design methodology enables to implement large scale electronic systems. Nowadays, electronic systems are applied to products in various fields; mobile platforms such as cell-phones and PDAs, media players such as MP3 players and DVD players, engine controller for vehicles, and all kinds of consumer electronics. These kinds of electronic systems have been referred to as *embedded systems*. Embedded systems have several features which general purpose systems like personal computers do not have.

- Embedded systems perform application specific tasks.
- Embedded systems have to meet tight design constraints which come from application specification.

Performance requirements for embedded systems are becoming high, and functional requirements are becoming various. These requirements result in complex embedded systems. Product vendors have to make a large effort to develop electronic products. Rising development cost is the most serious problem in the design of electronic systems. Embedded systems have been developed as Application Specific Integrated Circuit (ASIC), which is dedicated specific applications, in early era of electronic system development. However, designing ASICs for all

products is not realistic approach in recent embedded system design. Flexible hardware which cannot be used for only a specific application but also various applications is required.

To address the problem of design productivity, instruction set processors are widely used in embedded system design. Instructions-set processors realize desired functionality with given application programs. There is no need to develop new hardware for new products if instruction set processors are employed. Instruction set processors greatly reduce development cost in embedded system design. Many applications are realized by instruction set processors in today's electronic products. Generally, instruction set processors designed for general purposes. However, general purpose processors are often inefficient from the point of view of cost-performance since they are not optimized for a specific application. Such inefficiency causes high manufacturing cost of products or low energy efficiency in application run-time. Recent electronic devices required high performance and high energy efficiency. General purpose processors usually do not meet requirements in recent electronic products.

To achieve required performance in short development period, Application Specific Instruction set Processors (ASIP) are developed and used. ASIPs are the instruction set processors which have custom hardware and instructions for dedicated applications. Such custom instructions can process applications effectively. Especially, high performance is required in digital signal processing field, and processors for digital signal processing applications are widely used in recent years. Though ASIPs realize high cost-performance embedded systems, there is a considerable problem from the point of view of software development of ASIPs. In a typical software development for general purpose processors, programmers write programs in high-level programming language, then, generate object code using compilers. On the other hand, in software development for ASIPs, programmers often have to write assembly code because compilers hardly generate application specific instructions in assembly code in some cases. Programming in assembly code requires much development effort and loses portability of programs. Therefore, there is great need for a technology to generate assembly code which brings out processors's best performance.

This chapter gives an overview of embedded processors, then, requirements for compilers for these processors are summarized. Then, the contribution of this thesis is described. Finally, the outline of this thesis is described.

1.1 Overview of Embedded Processors

This section gives brief survey of several types of embedded processors and their characteristics.

1.1.1 Digital Signal Processors

Digital Signal Processors (DSPs) are dedicated to real-time digital signal processing such as FIR filtering, IIR filtering, FFT and so on. To meet severe performance requirements due to real-time processing, many architectural enhancements have been introduced into digital signal processors.

There are some concepts in design of DSPs.

- Efficient memory access mechanism to process a large amount of data, such as dual memory access, modulo addressing and memory accesses with address modifications.
- Complex operations to process data efficiently such as multiply-accumulation, memory access with shift and parallel arithmetic operations.
- Zero overhead loop mechanism to reduce loop overhead.
- Instruction set which supports Fixed-Point Arithmetic.

The first successful DSP in industry was TMS320C1x series[2] provided by Texas Instruments. Because the TMS320C1x is early generation of DSPs, some characteristics enumerated above are not introduced. However, some DSP specific features such as memory access with address modification, complex operations and support of fixed-point arithmetic are found.

The next generation of TI's DSP, TMS320C2x series[3] has all features listed above. These features enable to fetch two operands from data memories and perform one multiply-accumulate operation within one cycle. This is one of the most remarkable feature of TMS320C2x.

1.1.2 Media Processors

Media processors target real-time media encoding and decoding. In recent years, the standards for audio CODEC, video CODEC, and picture compression have been proposed and used in

many kinds of electronic devices. Since encoding and decoding of multimedia digital data require high performance for electronic devices, existing general purpose microprocessors and DSPs did not have enough processing ability to perform the codec standards. The emerging technologies to meet higher performance require exploitation of parallelism in two different levels.

- SIMD instructions which operate on subword data in registers
- Multiple instruction issue mechanism as VLIW architecture

SIMD instructions exploit data level parallelism in applications. In media processing applications, usually there exist a lot of operations which can perform in parallel. By mapping such operations in one instruction, higher performance can be obtained than conventional instruction which perform one operation per one instruction. Multiple instruction issues mechanism which is referred to as VLIW (Very Long Instruction Word) exploits instruction level parallelism. In assembly code, instructions which have no dependence among them are found. By issuing such instructions simultaneously, higher performance can be obtained than single issue processors.

Media processors have been provided by several companies. Texas Instruments has provided TMS320C6x series [4]. TMS320C6x series is a advanced generation of TMS320C2x, and dedicated not only to digital signal processing applications but also to digital media applications. Processors in TMS320C6x series have all features of DSPs and both of SIMD instructions and VLIW architecture features. TMS320C6x series are VLIW processors and allowed to issue up to 8 instructions simultaneously. 2 parallel and 4 parallel SIMD instructions are also supported. PNX1300 media processors [5] which provided by NXP Semiconductors are also VLIW based media processors. PNX1300 media processors are able to issue up to 5 instructions and support SIMD instructions. Several DSP features are also supported by PNX1300 media processors.

1.1.3 Microcontrollers

Microcontrollers are used for controlling several peripheral devices. Microcontrollers receive signals from various peripheral devices such as sensor signals and interrupts, and then send signals to control other peripheral devices. The increasing usage of microcontrollers in recent

years is controllers for vehicles. Conventional mechanisms of vehicle control, mechanically controlled systems like oil pressure, are to be replaced with electronical control systems. Required performance to microcontrollers is not so high compared to media processors. However, microcontrollers should be dependable, and requirement of real-time reactivity must be met because of drivers's safety.

1.1.4 Application Specific Instruction Set Processors

All processors reviewed in previous sections are designed by processor vendors, and system developers just use them without any modification. Unlike those processors, configurable processors are redesigned or modified by product developers using processor design tools provided by tool vendors. Since processor designers can configure their processors, processors optimized to specific applications can be obtained. Software development tools such as compilers, assemblers and linkers are typically provided by tool vendors.

Many academic and commercial processor design tools are presented.

Chess/Checkers [6] [7] is an embedded processor design tool-suite provided by Target Technologies. In the Chess/Checkers processor development environment, processor designers design processors using nML processor description language. Chess takes target processor description written in nML language and application program written in C language, compiles the application program, and generates assembly code for the target processor. The processor performance can be estimated using a retargetable instruction set simulator, Checkers. The processor HDL is generated by HDL generator Go from the processor description written in nML. Processor designers can explore the processor design space by modifying the processor descriptions and using Chess/Checker tool-suite.

Processor Designer [8] [9] provided by Coware is also an embedded processor design tool. Processor Designer uses LISA processor description language like Chess/Checkers's nML language. Generation of HDL of processors and software development tools such as compiler, assembler and simulator are also supported by Processor Designer.

Tensilica's Xtensa [10] is an configurable processor and its architecture parameters and instruction set are customizable. Processor designers are able to add instructions using TIE lan-

guage which describes additional instructions [11]. Similar to Chess/Checkers and Processor Designer, HDL generation of processors and generation of software development tools are supported. The feature of Tensilica's processor design tool-suite support automation of processor customization. Xtensa Xplorer takes target application written in C programs, then, automatically customize the target processor to process efficiently the target programs.

1.2 Requirements for Compilers

The main role of compilers in embedded system design is to ease software development. Writing assembly code is a hard task because of the low-level of abstraction of programing model. Additionally, the requirements of a huge variety of functionalities of recent embedded systems make embedded software large and complex. Developing all embedded software by assembly language is not feasible in recent embedded system development. To use compilers in developing embedded software is not unusual while embedded software developers are used to program in assembly language in the past.

The most significant task of compilers is to translate programs written in high-level programming language into assembly code which is an instruction sequence of a target processor. Moreover, there are some additional requirements around compilers.

- Compilers should support a programming language which gives a suitable programming model to describe applications in the target application domain.
- Compilers should generate optimized assembly code which take advantage of the target processor.

The first requirement stems from the need of high software productivity. The abstraction of programming language should be not only high-level but also suitable for the target application domain. The translation of algorithm of applications into programs should be easy for programmers, if the programming model is suitable to the application domain. As a result, programmers can rapidly develop software.

The second requirement stems from the requirements for performance. Unless the application specific architecture features and instructions are utilized, high performance cannot be

obtained. Traditional compilation methodology does not take into account most application specific features. Generally, it is difficult for compilers to utilize architecture specific features. However, compilation method to utilize application specific features are essential for embedded system development.

1.3 Contributions

This thesis discusses code generation method for application domain specific instruction set processors. Challenges involved by the requirements mentioned in the previous section are tackled.

As a compilation method of application specific programming model, code generation method for block-floating-point processors are studied. Block-floating-point processors have instruction set to perform operations based on block-floating-point arithmetic. A challenge in compilation for block-floating-point processors is to bridge a gap between the programming model of block-floating-point arithmetic and usual programming model of a programming language. The method to describe the computation procedure of block-floating-point arithmetic in high-level programing language and to compile programs is the main topic of this study.

Compiler optimization method for media processors is also studied in this thesis. Media processors play an important role in recent embedded system design because of the spread of digital media applications among many kinds of electronic devices. Compilation method to fully utilize media processors is crucial for development of embedded software. The assembly code generation method for a class of instruction called SIMD instructions is focused in this study. Difficulties in compilation for SIMD instructions are extraction of operations to perform in parallel and to deal with positions of data in registers. To minimize the number of arithmetic operation instructions, compiler should map the maximum possible operations to SIMD instructions. However, since each operation in SIMD instructions process data existing the same position among different registers, permutation instructions which reorder or repack data in registers may be required. Permutation instructions take additional execution cycles hence it is desirable to use as less permutation instructions as possible. By determining positions of data in registers appropriately, the number of permutation instructions will be small. However,

the positions of data in registers is determined inappropriately, overhead caused by permutation instructions becomes large. The code generation method to use SIMD instructions considering the permutation instructions is the main topic of the latter half of this thesis.

The main contributions of this thesis are as follows. The first contribution is on compilation method for block-floating-point processors.

- First, the code generation method for block-floating-point processors is proposed. With the consideration to describe the block-floating-point arithmetic in the high-level programming language, the processor architecture, programming scheme and compilation method are studied. A complete evaluation of processors and compilers have been performed.

The second contribution is on compilation method for media processors.

- Code generation for media processors with SIMD and permutation instructions is formulated into integer linear programming (ILP) problem.

- The code generation method using ILP solver is proposed. The effectiveness of the method is demonstrated by applying it to a set of programs from the domain of digital signal processing applications.

- A heuristic for the code generation with SIMD and permutation instructions based on data flow graph representation of programs is proposed. The output of this heuristic is not always optimal. However, this heuristic outputs solutions in very shorter time than ILP based code generation method.

- The method to generate permutation instruction sequence which is necessary to heuristic code generation method is presented. This method computes an instruction sequence to generate desired data permutation with given permutation instruction set.

- The heuristic code generation method is evaluated using a real commercial media processor and several digital signal processing applications.

1.4 Thesis Outline

The remainder of this thesis is as follows.

In chapter 2, related work of this thesis is summarized. Code generation methods for application domain specific instruction set processors are reviewed in this chapter. Ideally, assembly code of any processors should be generated from descriptions of processor independent high-level programming languages. However, such compilers are not always provided or desired performance cannot be always obtained. Several researches or practical approaches to deal with this problem are reviewed. Compilation techniques to generate high performance code from processor independent high-level programming languages are also reviewed.

In chapter 3, a compilation method for block-floating-point processors is described. An instruction set processor supporting H-BFP arithmetic and its application development method are proposed. The processor is designed based on the RISC architecture to enable compiler-based development. The programming scheme using intrinsic functions and compilation method for the block-floating-point processors are presented and evaluated.

In chapter 4, a code generation method for SIMD instructions considering permutation instructions is described. The code generation method is based on a code selection problem formulated into integer linear programming problem. Code generation by solving the formulated problem using ILP solver is evaluated.

In chapter 5, a heuristic code generation technique for SIMD instructions with permutation instructions is described. This method identifies SIMD instructions by finding and grouping the same operations in programs. After the SIMD instruction identification, permutation instructions are generated. In this permutation instruction generation, Multi-valued Decision Diagram (MDD) is introduced to represent and to manipulate sets of packed data. The code generation method integrated the heuristic to identify SIMD instructions and permutation instruction sequence is evaluated.

Conclusion and future work are described in chapter 6.

Chapter 2

Related Work

This chapter discusses code generation methods for embedded processors. Then, related work of the topics studied in this thesis is summarized.

2.1 Overview of Code Generation Method for Embedded Processors

There are some code generation methods for embedded processors. A difficulty in code generation for embedded processors is to utilize instructions dedicated to specific application domain. The most desirable way to use application specific instructions is automatic generation of assembly code by compilers. Code optimization techniques for application specific instructions or architectures have been studied for any kinds of instructions or architectures [12], [13],[14], [15].

Automatic generation of assembly code from processor independent programs in high-level languages is a challenging task because of the gap between model of programming language and the model of applications in the target domain. Some approaches have been investigated to manage this gap [16], [17], [18], [19]. [20], [21], [22]. After the review of code generation methods, existing code generation methods for two classes of embedded processors are discussed.

2.1.1 Assembly Language or Compiler Intrinsic Functions

When compilers are not available or do not generate application specific instructions, the simplest solution to request for using application specific instructions is to write software in assembly code. Assembly programming of performance critical processes is still a prevailing solution in current software development. Programming with compiler intrinsic functions which are functions to be mapped into specific instructions in high-level programming languages is also taken to use application specific instructions. There are some variations of this approach.

- Program entire software in assembly language.
- Program performance critical functions in assembly language, then integrate them with programs written in high-level programming languages at linking time.
- Use previously designed libraries.
- Embed assembly code in programs written in high-level programming languages.
- Use compiler intrinsic functions in programs written in high-level programming languages.

If compilers are not available or the target application is small, assembly programming of entire software is a possible way to use application specific instructions. Programming performance critical functions in assembly language is another possible approach to exploit processors's potential. Since small portion of software is frequently executed typically in run-time, programming in assembly only critical portion code is effective.

Some processor vendors provide pre-compiled functions as libraries, which are frequently used for applications in target application domains. Application programmers can use application specific instructions through functions collected in libraries [23], [24].

Using application specific instructions in high-level language is also a possible approach to exploit application specific instructions. Some compilers have mechanism called inline assembly. Inline assembly generate fragments of assembly code written in programs of high-level programming language as it is. Programmers can embed assembly code directly into target application programs. Compiler intrinsic functions can be used in some compilers. Compiler

intrinsic functions are the functions in high-level languages mapped into specific instructions by compilers. Programmers can use specific instructions by writing compiler intrinsic functions like usual functions.

2.1.2 Language Extension or Construction

To ease application development and tasks of compilers, special high-level languages have been proposed.

are extended or new languages for specific application domain are constructed. Compilation of general high-level languages for application specific instructions would be a hard task due to the gap between programming model of languages and instruction sets. Limited data type, arithmetic operations and program control flow model in high-level programming languages make production of efficient assembly code difficult. Sequential programming model is an obstacle to exploit instructions to perform parallel operations.

Language extension to break the limitation of existing high-level languages by introducing new data type, new operations, new control flow model and parallelism has been studied before [19]. [20], [21]. Construction of application domain specific languages has been also studied [22]. These approaches allow programmers to use application specific instructions easy.

2.1.3 Automatic Code Generation and Optimization

The most desirable way to use application specific instructions for programmers is automatic generation of assembly code making the best use of target processors. Many code generation and optimization methods for many kinds of processors have been studied [25],[26], [12], [15].

There are several approaches to generate and optimize assembly code or programs [26],[15]. One approach to optimize programs is transformation of intermediate representation of programs. Processor independent and dependent optimization algorithms at intermediate representation level have been studied a lot [26]. Another approach to optimize programs and assembly code is generation of high quality assembly code in assembly code generation phase [15]. In assembly code generation, compilers have three tasks, instruction selection, instruction scheduling and register allocation. Optimization algorithms in these tasks have been also

studied. Since this thesis focuses method to utilize application specific instructions, the code generation and optimization methods at intermediate representation and instruction selection are reviewed only in this section.

Early generation of digital signal processors have single-issue architecture, and have complex instructions such as memory access operations with complex addressing mode and multiply-accumulation operations to achieve efficient instruction encoding [3]. Such instruction set causes multiple translation into assembly code for a given program. A problem of translating programs into most efficient assembly code has been studied by many researchers [27], [28], [29], [30].

In the research on the code generation for digital signal processors, code generation methods to achieve high signal processing quality for fixed-point digital signal processors have been studied. In this topic, accuracy of outputs of signal processing is the matter to be considered. Program transformation from floating-point programs into fixed-point programs at high-level programming language have been studied [17], [18], [16].

2.2 Code Generation for Block-Floating-Point Processors

2.2.1 Block-Floating-Point Processors

There are two major types of digital signal processors classified by the supported arithmetic. The first one is floating-point arithmetic and the other is fixed-point arithmetic. Each of these DSPs has different features. From the application software development point of view, software for floating-point DSPs can be efficiently developed, because high signal quality can be easily achieved due to the nature of floating-point arithmetic. However, floating point units are expensive and not suitable to realize cost-effective digital signal processing systems. On the other hand, fixed-point DSPs do not have expensive hardware. For the reason of low hardware cost, fixed-point DSPs are widely used in consumer electronics. However, the development of software for fixed-point DSPs is time consuming task because it is difficult to develop fixed-point implementation which achieves high signal processing quality.

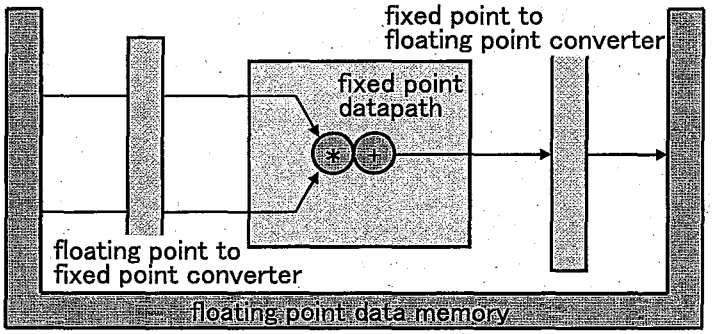


Figure 2.1: H-BFP Arithmetic

As a compromise between floating-point and fixed-point, block-floating-point (BFP) implementation is also used on fixed-point DSPs. Block-floating-point is based on the concept of floating-point; that is to say, the block-floating-point systems use floating-point format as number representation and performs arithmetic operations in floating-point manner. The difference between floating-point and block-floating-point is that several numbers share an exponent in block-floating-point systems while each number has its own exponent in floating-point systems. The advantages of block-floating-point arithmetic are lower performance requirement for processing devices than floating-point and suitable computation steps for fixed-point DSP. These advantages have motivated system developers to use block-floating-point arithmetic.

While a BFP implementation can realize high signal processing quality similar to floating-point on a low-cost fixed-point hardware, developing BFP implementation is still a hard task. BFP has been applied to some limited applications [31] [32], but efficient BFP implementations for other applications are not well known. Actually, there is a considerable trade-off between accuracy and hardware cost, in BFP arithmetic [33]. This trade-off makes development of BFP systems difficult. In order to solve the problem of lacking a good implementation approach, hierarchical block-floating-point (H-BFP) arithmetic [34] [33] and a processor with H-BFP instruction set have been proposed [35]. The basic concept of H-BFP is to keep data on the memory in floating-point format, while processing data in fixed-point format.

Fig. 2.1 illustrates the concept of H-BFP arithmetic. The data in memory are represented as floating-point numbers. The floating-point to fixed-point converter is used to convert the data. Data processing are performed on the fixed-point data path, and then the results are converted floating-point representation by the fixed-point to floating-point converter. The fixed-point to floating-point converter performs floating-point normalization. The results are stored to the data memory. With H-BFP arithmetic, desired signal processing quality and reasonable hardware implementation can be obtained simultaneously like usual BFP. Details of H-BFP is

2.2.2 Related Work on Code Generation Method for Block-Floating-Point Processors

There exists processors which uses block-floating-point arithmetic such as conventional fixed-point DSPs [4] and H-BFP DSPs [35]. However, compilation method for block-floating-point instruction set have not been studied in the past. The application development approach presented in [35] is writing assembly code from scratch referring to the floating-point implementation of the target application. Block-floating-point implementation of FFT on conventional fixed-point DSPs presented in [36] is also assembly programs. Manual translation of programs from high level language into assembly language is error prone and a time consuming task. A software development method which offers high productivity is required.

In this thesis, an instruction-set processor supporting H-BFP arithmetic and its application development method are proposed. Since this is the first research on the code generation by compilers for H-BFP instruction set, RISC based processor is originally proposed to separate compiler's task from difficulties which do not stem from H-BFP feature. In the proposed software development flow, H-BFP programs are implemented by modifying usual floating-point programs. The required modification is only to add special functions. The proposed code generation method converts the special functions into H-BFP instructions. Since the modification does not require complex program transformations by hand, H-BFP program can be easily developed.

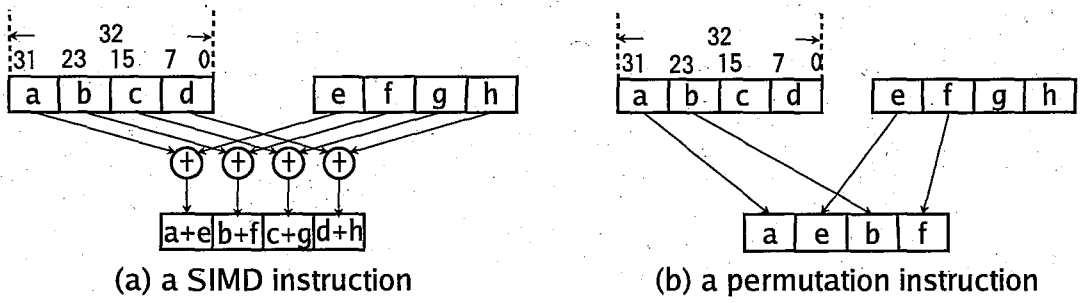


Figure 2.2: SIMD and permutation instructions

2.3 Code Generation for Media Processors

2.3.1 Media Instruction Set

Multimedia applications become popular applications which run on wide variety of platforms. Most multimedia applications demand high performance for electronic devices which execute those applications. Recent microprocessors are often customized to execute multimedia applications efficiently. The good nature in most multimedia applications is data parallelism in applications. Therefore, many processors adopt SIMD (Single Instruction Multiple Data) instructions to exploit data parallelism. SIMD instructions perform operations on multiple data packed in registers as shown in Fig.2.2(a). When a SIMD instruction is executed, the same operations are executed at the same time. Obviously, the processing efficiency of SIMD instructions is higher than that of conventional instructions which perform one operation at a time. Moreover, no special hardware is required to implement SIMD instructions.

SIMD instructions are useful, but most compilers have limited ability to use SIMD instructions. In view of this limitation, in order to exploit SIMD instructions, programmers need to use compiler intrinsics, special functions in high level programming languages, which are mapped to specific instructions, or to write programs in assembly languages. However, using compiler intrinsics or writing assembly programs are time consuming tasks, and decrease portability of programs. Therefore, a technique for automatically generating assembly programs including SIMD instructions is required.

The difficulty of code generation that exploits SIMD instructions stems from the data par-

allelism in registers. When using SIMD instructions, the positions of data in registers must be noted. When a SIMD instruction which operates a binary operator is executed, operands of each operation performed by a SIMD instruction must be at the same position in registers. If data and operations on the target application are well coordinated, SIMD instructions can be generated easily. If not, generation of additional *permutation* instructions which reorder or repack data in registers is needed. The permutation instructions take two source registers with packed data, and take some data elements from each register, and put these data elements into one target register. Fig.2.2(b) shows a typical permutation instruction which takes two elements from each source register and packs into them the target register. Although such data repacking instructions take run-time execution cycles, the total execution cycles can decrease by the effect of SIMD instructions. The combination of SIMD and data repacking instructions can achieve high performance improvements compared with the case that SIMD instructions are unused.

There are many problems around code generation with SIMD instructions. One of the most essential topic is finding permutation instruction sequence which generates required packed data from given packed data with given permutation instructions. Most processors do not have all possible permutation instructions, but have several permutation instructions. In addition, the number of all combination of data repacking is very large. Therefore, the permutation instruction sequence which generates required packed data is not always found easily because of limitation of available permutation instructions and large search space of data repacking. This is one of the most significant problem in the exploitation of SIMD instructions.

2.3.2 Related Work on Code Generation for Media Processors

Many publications have been released about automatic code generation of SIMD instructions[37, 38, 15, 39, 40, 41, 1, 42, 43].

In [37], using several analysis and loop transformations, loops are vectorized to generate SIMD instructions. Though [37] aims at exploitation of SIMD instructions, [37] does not take account of data reordering . [39],[40] and [41] present a vectorization technique in the presence of misaligned memory access and data type conversions in loop bodies. Using the concept of *virtual vectors*, statements in loop bodies are vectorized. Misaligned memory access

and unaligned vector operations are aligned by pack and unpack operations. For the statements with mixed data length, data conversion instructions are also generated. [42] proposes a vectorization technique of loop bodies with interleaved data access of arrays. Utilizing two classes of data packing instructions, *extract* and *interleave* instructions, data repacking instruction sequence which arrange interleaved data in a non-interleaved form is generated. SIMD instructions are generated for vector operations on the rearranged data. [43] proposes a SIMD instruction utilization technique combined software pipelining. Functional units used by SIMD instructions and used by conventional scalar operation instructions are individually equipped in most processors which have SIMD instructions. Therefore, good utilization of both SIMD functional units and scalar functional units lead higher performance than conventional vectorization techniques. In [43], operations in loops are mapped to scalar operation instructions or SIMD instructions so that the number of instructions to be performed in a loop iteration is minimized.

All the above approaches target loops. On the other hand, there are some approaches targeting basic blocks or unrolled loop bodies. The advantage of the basic block level approaches is that it is applicable to wide range of programs. The loop level approaches such as [37, 39, 40, 41] and [43] are targeting well-formed loops generally. However, well-formed loops are not frequently appeared in practical programs. Additional tasks such as manually analyzing and rewriting programs may be necessary before applying loop level SIMD optimization so that the optimization works on programs. On the other hand, basic block level approaches are easily applicable to practical programs, because they are not sensitive to control structures in programs. In addition, parallelism within a basic block can be also utilized in basic block approaches, which is not considered in loop level approaches. In [15], pattern matching and covering problem with SIMD instructions are formulated to Integer Linear Programming (ILP). Solving the covering problem using ILP solver, highly optimized assembly codes with SIMD instructions are obtained. However, this approach takes too much time to solve ILP problems. For the latest SIMD instructions which handle 4 or 8 packed data, the time required to solve ILP problems may not be acceptable. Moreover, reorder or repacking data in registers is not handled in this method. In [38], SIMD instructions are generated by grouping statements in a basic block. Using data dependency and alignments information, statements which are executable in parallel

are grouped into *Pack Set* to minimize data packing cost. Performance improvements are larger than traditional vectorization. However, the way to generate permutation instructions and the related problem of packing are not mentioned. In [1], generation of SIMD and *permutation* instructions is presented. SIMD instructions are generated by grouping operations in a basic block represented by Data Flow Graph(DFG). After the grouping, permutation instructions are inserted between SIMD instructions. In [1], in order to generate *permutations* which mean packed data ordering in registers, *permutation decomposition backward tree* and *forward tree* are used. The backward tree represents the decomposition of the target permutation by permutation instructions. On the other hand, the forward tree represents generation of permutations by permutation instructions from input permutations. By matching the leaves of backward and forward trees and searching paths from the target permutation to input permutations, permutation instruction sequence to generate target permutation is obtained. The advantage of [1] is that more efficient code can be generated compared to [15], because of utilization of permutation instructions. Also, a method to generate instruction sequences to generate permutations is presented, which is not mentioned in [38]. However, the size of the backward tree used in [1] exponentially grows according to the depth of the backward tree. The larger the number of data in registers or the number of permutation instructions becomes, the longer the compilation time becomes.

In this thesis, code generation methods for media processors are proposed. At first, the code generation problem with SIMD and permutation instructions are formulated into integer linear programming (ILP) problem extending [15]. Then, the code generation method using ILP solver is proposed. In this method, data move operations are introduced in directed acyclic graphs representing programs. Moreover, permutation instructions are introduced in ILP formulation. The problem can be solved by using ILP solver. Consequently, compilers can generate assembly code including SIMD and permutation instructions. The advantage of the proposed method is that the SIMD instruction utilization is higher than without permutation instructions. As a result, performance and code size are improved at the same time. Secondly, a fast code generation method for media processors is proposed. This method generates SIMD instructions by finding operations which are executable in parallel and mapping them to SIMD instructions. In this method, *permutation* instruction generation technique based on

MDD is presented. Using MDD, the proposed technique exhaustively generates permutations and finds feasible instruction sequences which reorder or repack data elements in registers. Since MDD can represent and manipulate sets of permutations efficiently, permutation instruction sequences can be generated in a short time.

Chapter 3

Code Generation for Block-Floating-Point Instructions

This chapter is organized as follows. The H-BFP arithmetic is summarized in section 3.1, and an instruction-set processor with H-BFP arithmetic and its compiler are in section 3.2. Experimental results are described in section 3.3. Finally, this chapter is concluded in section 3.4.

3.1 Hierarchical Block-Floating-Point Arithmetic

In this section, conventional block-floating-point arithmetic and hierarchical block-floating-point arithmetic[34] [33] are briefly summarized.

3.1.1 Conventional Block-Floating-Point Arithmetic

Block-floating-point arithmetic (BFP) is based on the concept of floating-point. Each number is represented as a pair of the scale-factor (exponent) and mantissa, and arithmetic operations are performed by the similar way as that of usual floating-point (FP) such that computing and normalizing mantissa then computing scale-factor. The difference between usual FP and BFP is that the scale-factor is shared by some numbers in BFP while each number has its own scale-factor in usual FP. In BFP, a set of numbers sharing a scale-factor is referred to as a

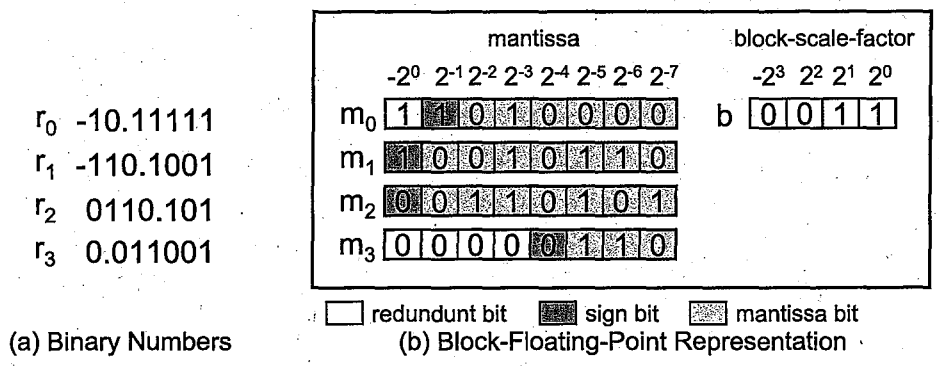


Figure 3.1: Block-Floating-Point Data Format

data block, and the shared scale-factor is referred to as a *block-scale-factor*. All numbers in a data block is normalized based on a block-scale-factor. Fig.3.1 shows a block-floating-point representation of 4 binary numbers. Fig.3.1(a) shows 4 binary numbers, and Fig.3.1(b) shows a block-floating-point representation of Fig.3.1(a). The block-scale-factor of the data block is determined by finding the largest scale-factor

Arithmetic operations are done in block basis. Scaling and normalizing mantissa are to be performed as same as usual FP. However, computational cost is less than that of usual FP because the number of scale-factor manipulations are small by sharing scale-factor. Also, main computation of BFP consists of integer operations such as shift, addition and multiplication. This means that arithmetic operations of BFP are suitable to perform on fixed-point DSPs. These are the motivation to introduce BFP.

3.1.2 Introduction of Hieracrchical block-floating-point Arithmetic

Hierarchical block-floating-point arithmetic, an improved version of BFP has been proposed in [33]. In H-BFP, each data element in a data block is represented as a floating-point number when it is on memories, and it is represented as a mantissa and a common block-scale-factor when it is computational data path. Note that the block-scale-factor is still available if all data elements in a data block are on memories, though the representation of a data element does not depend on the value of block-scale-factor. During data processing, data elements are loaded from memories to the registers, then, they are converted from floating-point format into fixed-point format in the block-floating-point manner. Arithmetic operations such as addition, multi-

plication and accumulation are performed on the data represented in fixed-point format. After the computations, results are converted into floating-point representation again, and stored into memories. Fig. 3.2 illustrates H-BFP arithmetic. As mentioned above, a set of data is located in data memory in floating-point representation. The floating-point to fixed-point converter is used to convert the data. Data processing are performed on the fixed-point data path, and then the results are converted floating-point representation by the fixed-point to floating-point converter. The fixed-point to floating-point converter performs floating-point normalization. The results are stored to the data memory.

The goal of H-BFP is to obtain reasonable implementation of digital signal processing systems with less development effort. H-BFP has the advantage in over conventional BFP arithmetic. In the development of conventional BFP based systems, design quality, such as signal quality, hardware cost and processing time, heavily depend on the implementation options. Application developers have to make effort to find a reasonable implementation. On the other hand, though H-BFP requires additional overhead in execution time due to data format conversions, high precision signal processing and low hardware cost are ensured. Fig.3.3 shows a comparison of the behavior of BFP to H-BFP. The upper part shows the case of BFP, and the lower part shows the case of H-BFP. Fig.3.3 depicts how the precision of numbers in a data block varies during a sequence of operations. In Fig.3.3, a data block is loaded from a data memory, used for certain operations as operands, stored to a data memory, then loaded from a data memory for next computation again. Comparing H-BFP with BFP, H-BFP require additional memory usage to keep element-scale-factor, additional computational cost for floating-point normalization. However, there is a remarkable point in H-BFP. The numbers in the data block loaded from memory for the next computation, which is the most left data block in Fig.3.3, some lower significant bit keep mantissa bit, while those of the data block of BFP lose some mantissa bit. Hence, H-BFP achieves higher accuracy than conventional BFP.

3.1.3 Principle of H-BFP Arithmetic

In this section, H-BFP arithmetic operations are described in detail.

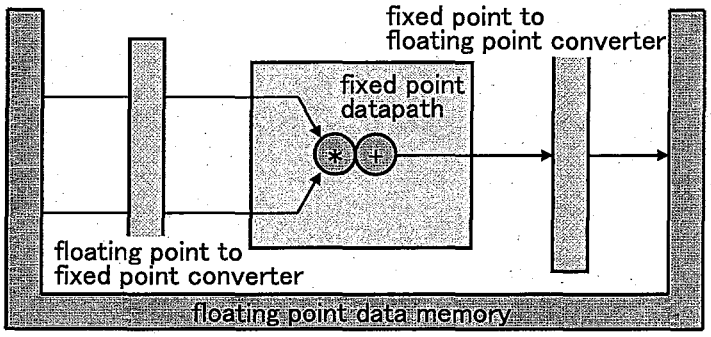


Figure 3.2: H-BFP Arithmetic

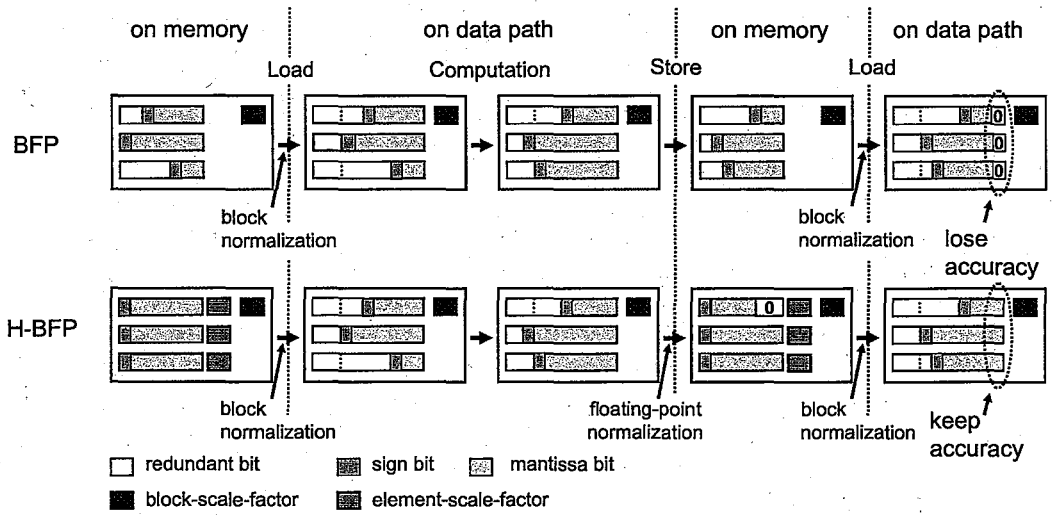


Figure 3.3: Comparison of conventional BFP and H-BFP

As mentioned in section 3.1.2, floating-point and block-floating-point representations are used in H-BFP. The floating-point representation of a data element consists of an element-scale-factor and a mantissa. Let the element-scale-factor and mantissa be represented as two's complement of binary numbers in this paper. The block-floating-point representation of a data element consists of a block-scale-factor and a mantissa. The block-scale-factor and mantissa in a block-floating-point representation are also two's complement of binary numbers. In the block-floating-point representation, guard bits are reserved at most significant bits of a mantissa. Guard bits prevents overflow in data processing and allows to omit input data scaling-

down[33].This leads higher accuracy than without guard bits. The bit width of the mantissa on memories and the one on data path do not have to be the same. Generally, the bit width of the mantissa on data path is longer than the one on memories to save memory capacity and memory access cost.

The procedural steps of H-BFP arithmetic operation are listed as follows. The input data blocks, data elements represented as floating-point numbers and block-scale-factors, are given on memories. The block-scale-factor of a data block is a maximum value of element-scale-factor in the data block. The computation results are obtained as output data blocks, data elements and block-scale-factors.

1. Compute *input block-scale-factors* which are block-scale-factors used for binary alignment of mantissa during computation. Initialize *output block-scale-factors* which are block-scale-factors of computation results.

2. Repeat below steps for every arithmetic operations

   (a) Loading data elements from memories, and extract and convert mantissas into fixed-point numbers referring to the input block-scale-factors.

   (b) Process data by fixed-point operations.

   (c) Convert fixed-point numbers into floating-point numbers, then store them to memories. Update output block-scale-factors simultaneously.

3. Save output block-scale-factors.

Looking the procedural steps, some H-BFP specific primitive operations can be found. Handling block-scale-factors, converting floating-point number to fixed-point mantissa and converting fixed-point mantissa are required to realize H-BFP. Realization of those hardware functions is the main topic of this paper.

3.2 H-BFP Processor and its Compiler

3.2.1 Design Concept

The goal of this study is to realize an instruction-set processor which efficiently executes H-BFP operations and provides a compiler for the processor. To achieve this goal, H-BFP processor is designed following principles:

- Based on a typical 32-bit RISC processor with integer instruction-set.
- Build a data path for H-BFP operations into a processor pipeline.
- Implement additional three types of instructions for H-BFP.
  - Integer arithmetic operations on scale-factors
  - Load floating-point data from memories, then convert them into fixed-point data
  - Convert fixed-point data into floating-point data, then store them to memories.

The first principle comes from a requirement for compilers. The compilation technology for 32-bit RISC is available today, such technology can be used for the proposed processor as it is. The second and third principles are lead by a requirement for realization of H-BFP operations. The fixed-point data path is originally supported by an integer processor. Handling scale-factors and supporting data format conversions are necessary to execute H-BFP operations. Hence, by implementing such operations on integer processors, H-BFP can be executed on the processors.

3.2.2 Hardware Implementation

A data path structure to execute H-BFP specific operations is shown in Fig.3.4. There are three parts enclosed with dotted line, float to fixed conversion, scale-factor manipulation and fixed to float conversion. The scale-factor manipulation is divided into two parts further, align-factor computation and scale-factor computation. scale-factor register file is shown in the center of Fig.3.4, fixed-point data path is shown at the bottom. The processing data is incoming from the left top, and is outgoing to the right top through two conversions and fixed-point data path. Implementations of individual components are as follows;

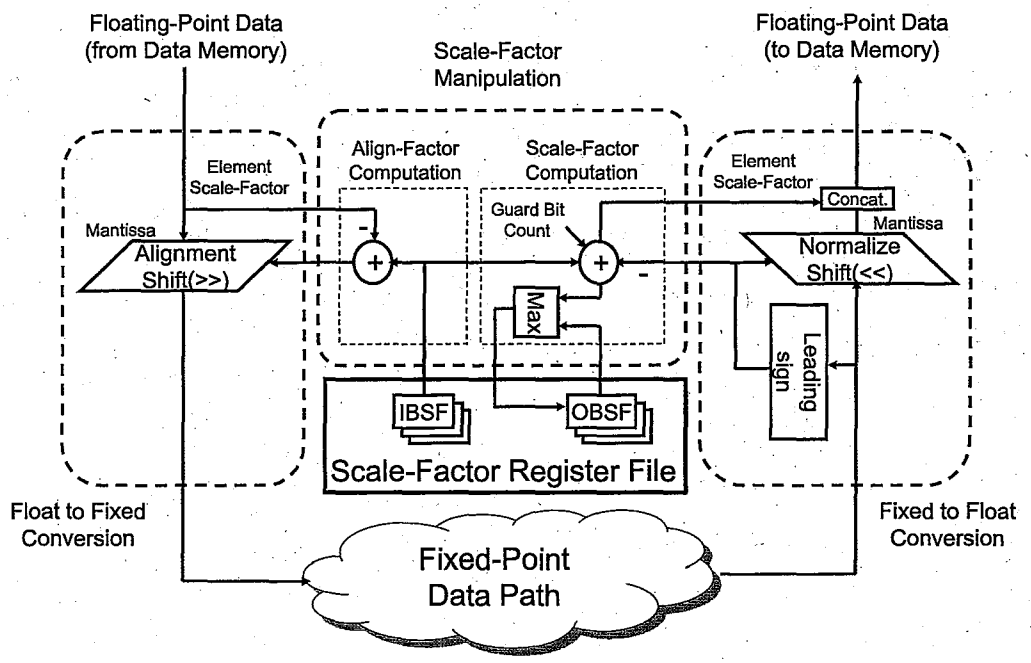


Figure 3.4: H-BFP Data Path

3.2.2.1 Scale-Factor Register File

The scale-factor register file is a register file to keep block-scale-factors. Input and output block-scale-factors (IBSF, OBSF) and temporary block-scale-factors (not appeared in Fig.3.4) are kept during data processing. The bit width of a register in this register file should be the same as the bit width of block-scale-factors of a H-BFP system.

3.2.2.2 Float to Fixed Conversion

In float to fixed conversion, floating-point numbers are converted into fixed-point numbers. A barrel shifter (Alignment Shift) is used in this conversion. The inputs of alignment shift are mantissa and a scale-factor. The mantissa is shifted by the value of the scale-factor generated by scale-factor manipulation, then, extended by the guard bit width.

3.2.2.3 Align-Factor Computation

In align-factor computation, the shift amount of mantissa for floating-point to fixed-point conversion is computed. The difference between input block-scale-factor and element scale-factor

is used typically.

3.2.2.4 Fixed to Float Conversion

In fixed to float conversion, fixed-point numbers are converted into floating-point numbers. A barrel shifter (Normalize Shift) is used in this conversion. The inputs of normalize shift are mantissa as a fixed-point number and a scale-factor to normalize the mantissa. The shift amount in this normalization is computed by a leading sign counter (Leading sign). The leading sign counter is implemented using a priority encoder. The element scale-factor is generated by scale-factor manipulation. The normalized mantissa and the element scale-factor are concatenated, then it is outputted.

3.2.2.5 Scale-Factor Computation

In scale-factor computation, the element scale-factors of floating-point number are computed. The element scale-factor is computed as (input block-scale-factor - leading sign count + guard bit count). The output block-scale-factors are also computed in this data path. The output block-scale-factor of a data block is the maximum element-scale-factor in the data block. Therefore, the output block-scale-factors can be computed by taking maximum number between temporary output block-scale-factor and generated element scale-factor for every time the floating-point number is generated.

3.2.3 Processor Description

Fig.3.5 shows the proposed H-BFP processor architecture. The H-BFP processor is based on a standard 5-stage pipelined RISC processor. The H-BFP data path is embedded in the pipeline. The target architecture is five stage pipelined; i.e., instruction fetch, instruction decode, execution, memory access, and write back stages. The H-BFP data path shown in Fig.3.4 is decomposed, and equipped over 4 stages. The scale-factor register file is placed in the second stage to allow scale-factors to be available in later stages. Float to fixed conversion and align-factor computation data path are placed in the five stage. Floating-point data loaded from data memories can be converted into fixed-point data before storing them to general purpose register

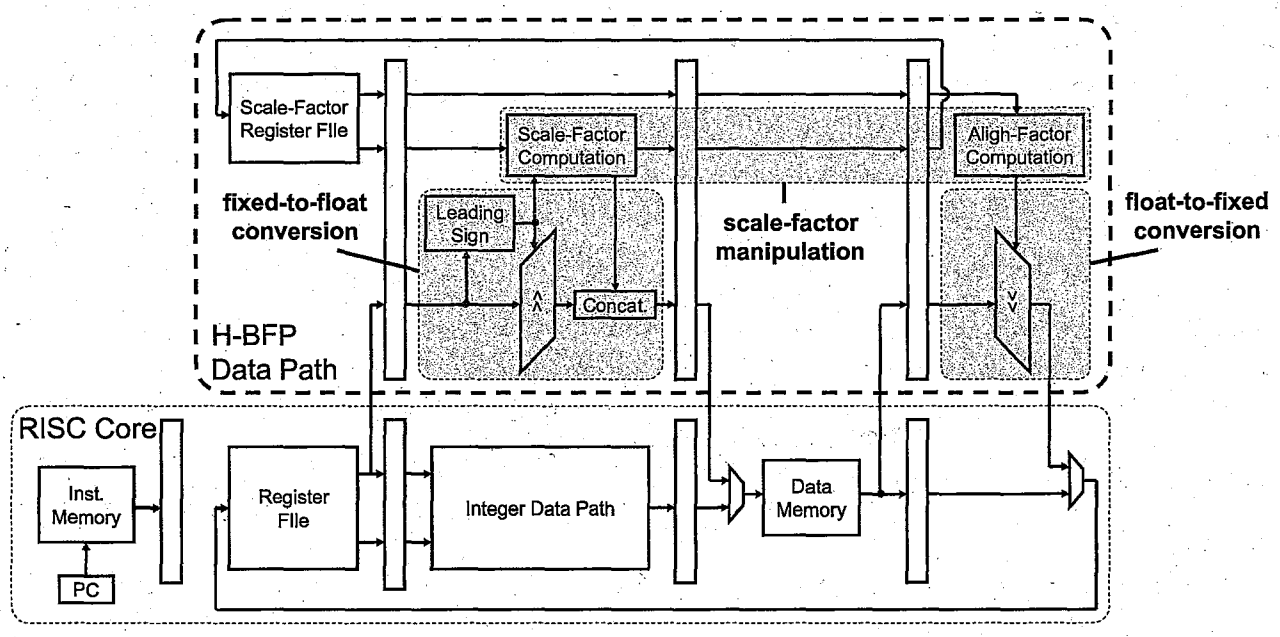


Figure 3.5: H-BFP Processor Architecture

file. Floating to fixed conversion and scale-factor computation data path are placed in the third stage. Floating-point data converted into from Fixed-point data can be stored to data memories right after the conversion.

For H-BFP based processing, instructions which perform primitive operations of H-BFP are implemented. As mentioned in section 3.2.1, there are three types of H-BFP specific instructions. As scale-factor manipulations, 13 instructions including to move block-scale-factors between scale-factor register file and data memory, several operations such as addition, subtraction and select maximum value are implemented. These instructions provides various block-based processing. Load instructions from data memory with floating-point to fixed-point conversion and store instruction with floating-point to fixed-point conversion are also implemented. There are 4 instructions for each load and store instructions. A variety of conversions can be performed by those instructions.

3.2.4 Compiler Support

In this section, software development approach for H-BFP processor introduced.

Software for H-BFP processor is developed as follows. A program based on floating-point arithmetic of the target application is written in a high level programming language first of all. Then, the program is rewritten into the program based on H-BFP arithmetic. The differences of the computation model between the floating-point and H-BFP are that data are non-blocked or blocked, and data conversions after/before operations are not needed or needed. Hence, the software for H-BFP processor can be developed by adding H-BFP specific operations to the floating-point arithmetic based program. To enable the programmers to add such operations, a compiler technique called compiler intrinsic is used. Compiler intrinsic functions are the functions in the high level programming language which are mapped to the specific instructions of the target processor. Using compiler intrinsic, H-BFP specific operations in H-BFP programs can be directly mapped to instructions of the H-BFP processor.

Fig.3.6 shows the development flow of conventional approa-ches and the proposed approach. In conventional fixed-point or BFP based software development, and H-BFP based software development presented in [35], floating-point arithmetic based program is developed first. Then, the target application is developed referring to the floating-point based program as a reference model. In the conventional fixed-point or BFP based software development flow, several implementations must be considered, and analysis of trade-off between signal processing quality and costs has to be performed. In the flow of [35], while the feature of H-BFP eases development, assembly programming is still needed. On the other hand, in the proposed approach, the target application program can be obtained easily because all have to do is program refinement by insertion of compiler intrinsic functions.

Fig.3.7 shows a program refinement example of a floating-point program. There are three program fragments in Fig.3.7. The left program is the floating-point implementation for processors supporting floating-point arithmetic. The upper and lower programs at the right in Fig.3.7 are the H-BFP implementation and floating-point implementation for the H-BFP processor, respectively. The upper right program can be obtained by inserting scale-factor manipulations and data conversions in block-floating-point manner. On the other hand, the lower right program

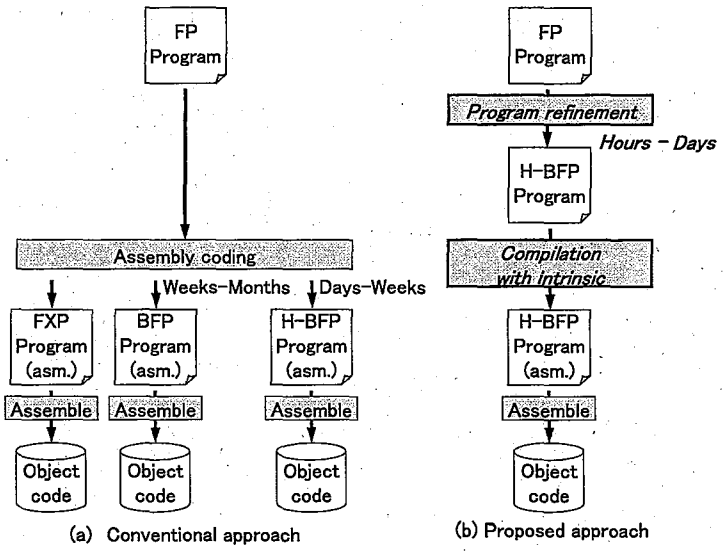


Figure 3.6: Comparison of Application Development Flow

can be obtained by inserting H-BFP specific operations in floating-point manner. Compiler intrinsic functions, `sfcselect`, `tofix`, `toflt`, are appeared into their corresponding instructions for the H-BFP processor.

All these example programs compute the addition of two vectors. In the upper right program in Fig.3.7, the addition of two vectors is interpreted as addition of two vectors which belong to different data blocks. The variables `asfb` and `bsfb` in Fig.3.7 hold the block-scale-factors for data blocks `a` and `b` respectively. The `sfcselect` function computes the block-scale-factor which determines the fixed-point data format on the addition. In the loop body, `tofix` function performs floating-point to fixed-point conversion, `toflt` function performs fixed-point to floating-point conversion. By inserting scale-factor manipulation such that the H-BFP processor performs the manipulation before every addition as shown the lower right program, floating-point implementation on H-BFP processor can be obtained. The floating-point implementation takes more execution cycles than H-BFP implementation in the runtime. However, floating-point implementation achieves higher precision of arithmetic operations than H-BFP implementation.

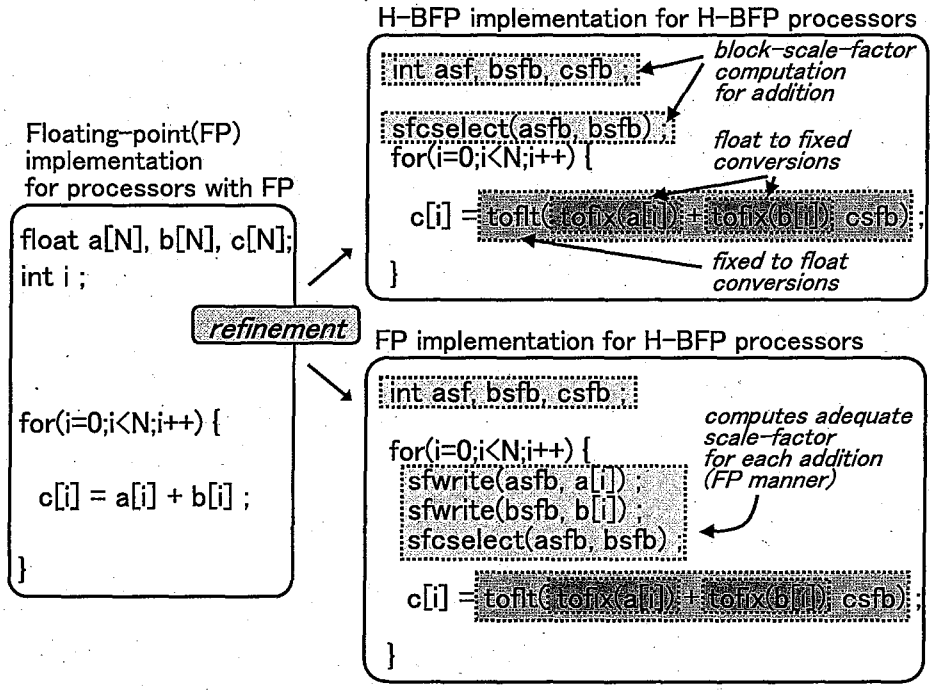


Figure 3.7: Program Refinement for H-BFP processor

3.3 Experimental Results

In this section, the experimental results are presented.

3.3.1 Experimental Setup

An HDL model of the H-BFP processor has been designed using an ASIP development tool, ASIP Meister[44]. and the compiler with compiler intrinsics has been developed. The H-BFP processor is based on the DLX[45] without floating-point operation instructions, **DLX-Int**. Adding the H-BFP data path and H-BFP instructions to DLX-Int, a H-BFP processor, **DLX-HBFP** is developed. Note that the original DLX[45] has no integer multiplication instruction, and it is supposed that floating-point multiplication should be used if the original DLX computes integer multiplication. Therefore, 16 bit multiplier and multiplication instruction are added to DLX-HBFP. The 16 bit multiplication is usually implemented in commercial digital signal processors. The bit width of integer and fixed-point data path of DLX-HBFP is 32 bits.

The floating-point format on memory is composed of 8bits exponent and 16 bits mantissa.

To compare the H-BFP processor with usual floating-point processor, DLX-Int with floating-point unit, **DLX-FP**, has been also developed. DLX-FP has a floating-point unit providing single floating-point computation with the IEEE standard[46]. The floating-point unit operates arithmetic operations such as addition, multiplication, and floating-point to integer conversion, integer to floating-point conversion, etc. DLX-FP take 10 cycles per a single floating-point addition/subtraction, and 13 cycles per a floating-point multiplication. If DLX-FP issues a floating-point operation, the internal pipeline of DLX-FP stalls until DLX-FP finishes floating-point operation.

A compiler to generate assembly code from H-BFP program, **H-BFP compiler**, is also developed for this experiment. H-BFP compiler generates H-BFP instructions from intrinsic functions in high-level program.

In this experiments, the design quality of hardware and application processing performance of DLX-HBFP are studied. DLX-HBFP, DLX-Int and DLX-FP are synthesized and estimated delay, frequency and gate count to evaluate DLX-HBFP in terms of the design quality. The performance of DLX-HBFP is evaluated using DSPstone benchmark suite[47]. DSPstone is a set of programs taken from digital signal applications. Compiling and executing programs of DSPstone, accuracy of output is evaluated. Static instruction count in generated assembly code is compared between H-BFP processor and floating-point processor. Moreover, execution time of programs run on DLX-HBFP and DLX-FP is compared to evaluate performance of the H-BFP processor. Finally, the outputs of DLX-HBFP and DLX-FP are compared with the output of double floating-point computation.

3.3.2 Results

In this section, experimental results are shown.

Table 3.1: Results of logic synthesis

	Delay [ns]	Frequency [MHz]	Area [gate]
DLX-Int	4.82	207.4	54161
DLX-HBFP	5.36	186.6	63107
DLX-FP	6.08	164.5	79087

Table 3.2: Breakdown of gate count of DLX-HBFP

Base(DLX-Int)		
		54161
Incremental gate count		
H-BFP components	1809	
Pipeline register	2712	
Multiplexer	2279	
Control	2016	
Other	115	
		8931
Total		63107

3.3.2.1 Hardware Evaluation

The hardware area of the H-BFP processor was estimated. The HDL model of DLX-Int, DLX-HBFP and DLX-FP were synthesized using a 0.14μm process. Synthesis results are summarized in Tab.3.1. The estimated delay, frequency, gate count for DLX-Int, DLX-HBFP and DLX-FP are shown. The total area of DLX-HBFP is about 63K gates with the maximum frequency at about 186.4MHz. The increased gate count is about 8.9Kgate while the DLX-FP increased about 25Kgate compared with DLX-Int.

The breakdown of gate count of synthesized DLX-HBFP is shown in Tab.3.2. In Tab.3.2, total gate count of DLX-Int are shown, and breakdown of incremental gate count is also de-

scribed. The gate count of H-BFP components is just 1.8Kgate. Additional gate count to embed H-BFP components into processor pipeline, which is the sum of pipeline register, multiplexer, control and other is about 7.1Kgates. On the other hand, increased gate count of DLX-FP more than 25Kgate. The floating-point coprocessor typically costs about 20-40Kgate[48][49]. This means that the ability of H-BFP can be added with little hardware and delay overhead.

3.3.2.2 Performance

To confirm the performance of the H-BFP processor/compiler, the size of assembly programs for H-BFP processor has been evaluated. DLX-HBFP was compared with DLX-FP. Table 3.3 shows the number of instructions to process each program for DLX-HBFP and DLX-FP. Comparing DLX-HBFP with DLX-FP, the number of instructions of DLX-HBFP is comparable to that of DLX-FP in 7 cases. This result indicates the H-BFP processor/compiler can process applications as efficient as usual processor/compiler supporting floating-point arithmetic.

Table 3.3: Comparison of the number of insns. among different implementations, N : the size of vector, M : the width and height of matrix, T : the number of taps

	DLX-HBFP [# of insns]	DLX-FP [# of insns]
n_real_updates	**14N+26**	14N+13
n_complex_updates	44N+30	36N+13
complex_multiply	**26N+15**	26N+7
convolution	**12N+20**	12N+12
dot_product	**12N+22**	12N+11
fir	12NT+30	10NT+9
matrix1	$12M^3+16M^2+7M+11$	$10M^3+14M^2+7M+4$
matrix2	$\mathbf{12M^3+25M^2+7M+12}$	$12M^3+23M^2+7M+4$
mat1x3	$\mathbf{12M^2+14M+11}$	$12M^2+12M+4$
fir2dim	$\mathbf{30M^2T+23M^2+5M+11}$	$30M^2T+24M^2+6M+5$

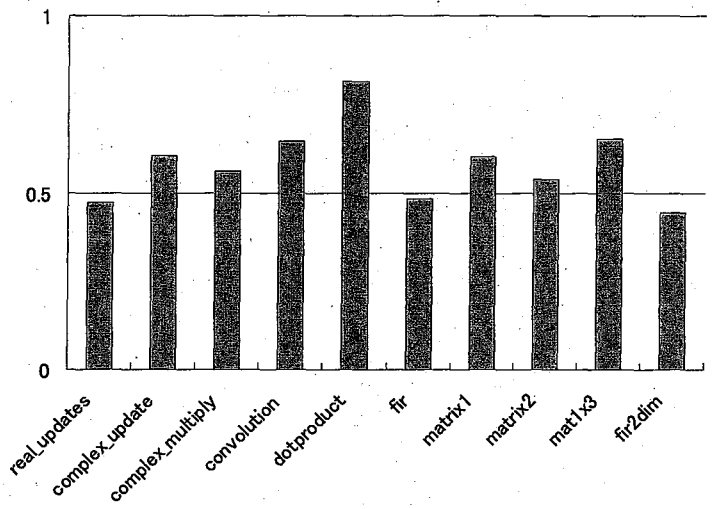


Figure 3.8: Relative Ratio of Execution Time of DLX-HBFP to DLX-FP

Fig.3.8 shows relative ratio of execution time of DLX-HBFP to DLX-FP. The execution time is defined as execution cycle count divided by the frequency. In Fig.3.8, relative ratio is ranged from about 0.5 up to 0.8. This is because that the execution cycles of DLX-HBFP is less than that of DLX-FP and the frequency of DLX-HBFP is higher than DLX-FP. Actually, the floating-point operations of DLX-FP take a larger number of cycles than usual floating-point unit. However, the number of dynamically executed instructions of DLX-HBFP is almost same as DLX-FP, and computatinal cost of H-BFP operations is smaller than floating-point operations. Hence, DLX-HBFP can prcocess applications more efficiently than DLX-FP.

3.3.2.3 Signal Processing Quality

C programs implemented by floating-point arithmetic in DSPstone benchmark has been modified for the DLX-HBFP. The HDL model of the DLX-HBFP with the object code generated by the compiler has been simulated on an HDL simulator. The white noise has been used as the input of the programs.

The signal processing quality of H-BFP implementation has been evaluated using an signal-to-noise ratio measure which is defined as

$$SNR = 10 \cdot \log_{10}[\frac{\sum_{n=0}^{N-1} R(n)^2}{\sum_{n=0}^{N-1}\{R(n) - S(n)\}^2}] \tag{3.1}$$

where N is the number of output samples of the application, $R(n)$ is the n th output of

Table 3.4: SNR of each programs run on DLX-HBFP and DLX-FP

	DLX-HBFP [dB]	DLX-FP [dB]
n_real_updates	83.13	146.67
n_complex_updates	81.66	148.33
complex_multiply	83.91	141.89
convolution	80.85	151.48
dot_product	82.68	137.00
fir	80.38	146.67
matrix1	81.01	146.51
matrix2	81.34	146.27
mat1x3	81.27	142.54
fir2dim	83.57	145.92

the double precision floating-point computation, and $S(n)$ is the n-th output obtained by HDL simulation of the H-BFP processor, respectively.

Tab.3.4 shows the SNR of DLX-HBFP and DLX-FP for each program. The first column shows the names of programs, the second and third columns shows the SNR of DLX-HBFP and DLX-FP, respectively. In Tab.3.4, SNRs of DLX-HBFP ranged from 80.85 to 83.91. SNRs of the DLX-FP score higher than those of the DLX-HBFP. This is because SNRs are depends on the bit width of mantissa. The bit width of mantissa of DLX-HBFP is 16 bits on the memory. On the other hand, the bit width of mantissa of DLX-FP is 23 bits because of single floating-point. According to the previous study, SNRs of H-BFP are expected up to 80dB, and the SNRs are enough to practical applications[33]. The results of Tab.3.4 is consistent with the previous work.

3.4 Summary

In this chapter, a processor supporting hierarchical block-floating-point arithmetic and software development method for the processor are proposed. In experiments, some applications has been implemented and simulated on the H-BFP processor. It is confirmed that the H-BFP processor can achieve high signal quality and low hardware cost. Using the proposed method, signal processing applications can be easily developed.

Chapter 4

Optimal Code Generation for Media Instructions

This chapter describes the optimal code generation method for media processors considering permutation instructions. This chapter is organized as follows: Section 4.1 describes SIMD instructions. Section 4.2 introduces a code selection method using tree parsing and dynamic programing [50]. Section 4.3 explains the Leupers's method [15]. Section 4.4 describes the proposed method. Section 4.5 shows experimental results. Section 4.6 concludes this chapter.

4.1 SIMD Instructions

n SIMD instructions, a value in a register is assumed to consist of several values. Fig. 4.1(a) shows a SIMD instruction that performs two additions on upper and lower parts of registers. LOAD/STORE instructions is also regarded as SIMD instructions. Fig. 4.1(b) shows an example of SIMD LOAD/STORE instructions. Usually, processors with SIMD instructions also have permutation instructions. permutation instructions transfer several values from a couple of registers into a register. permutation instructions are useful to execute SIMD instructions effectively because permutation instructions produce packed data type.

Fig. 4.2 shows an example of permutation instructions. First, a[i] and a[i+1], and b[i] and b[i+1] are loaded by a LOAD instruction. Since source values of additions are not located

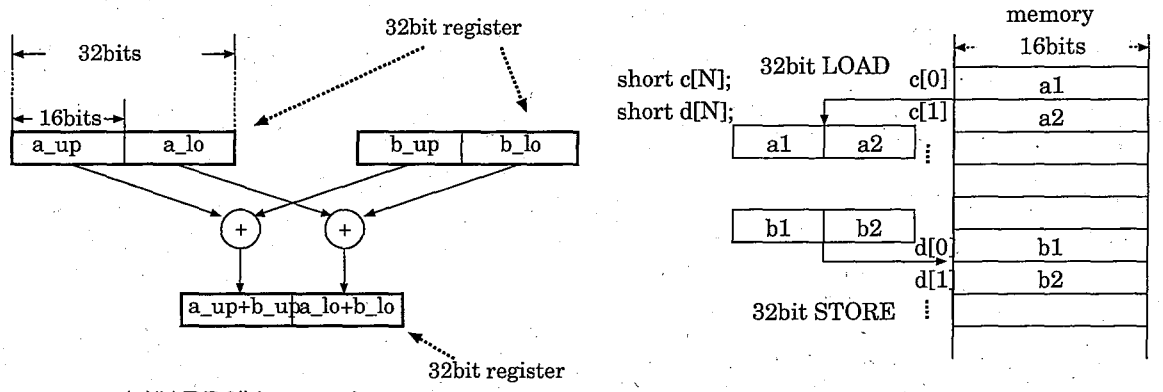


Figure 4.1: Example of SIMD instructions.

regularly, a SIMD instruction is not applied right after loading. However, replacing values by permutation instructions, SIMD instructions can be applied and the program is executed effectively.

4.2 Code selection

Code selection is usually implemented by using tree pattern matching and dynamic programming [50].

Let us assume a DAG $G = (V, E)$ representing a given basic block. Here $v \in V$ represents an IR level operation such as arithmetic, logical, load and store. $e \in E$ represents data dependency. A DAG is divided at its CSE(Common Sub Expression) into DFT(Data Flow Tree). Consequently, a set of DFT is got for a basic block.

In tree pattern matching and dynamic programming technique, an instruction set is modeled as a tree grammar. A tree grammar consists of a set of terminals, a set of nonterminals, a set of rules, a start symbol and a cost function for rules. Terminals represent operators in a DFT. Nonterminals represent hardware resources which can be stored data such as registers and memories. A cost function determines a cost, which is usually execution cycle of instruction corresponding the rule. Rules is used to represent behavior of instructions. For example, an ADD instruction which performs addition of two register contents, and stores the result to a

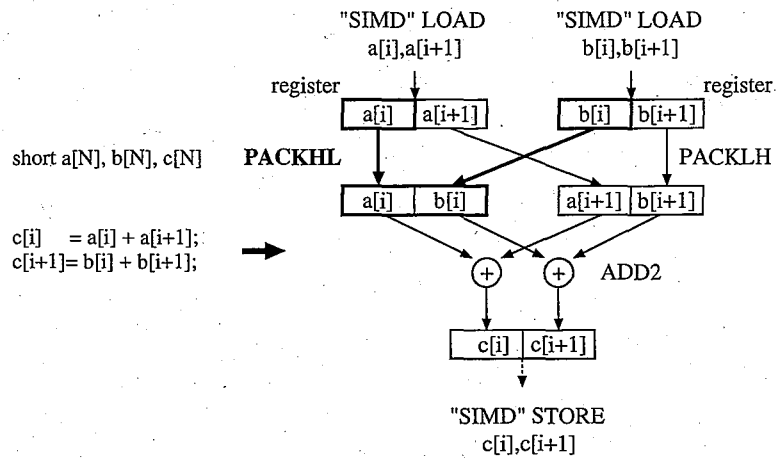


Figure 4.2: permutation instructions.

register represents as follows.

$$reg \rightarrow PLUS(reg, reg)$$

Code selection for a DFT is carried out by deriving the DFT which has minimal cost. In order to derive a tree which has minimal cost, dynamic programming is used. In a bottom–up traversal, all nodes v in DFT are labeled with a set of triples (n, p, c), where n is a nonterminal, p is a rule, and c is the cost for subtree which root is v. This means that node v can be reduced to n by applying rule p at cost c.

4.3 SIMD Instruction Formulation

In this chapter, formulation and solution in reference [15] are summarized.

4.3.1 Rules for SIMD instructions

A set of DFTs mentioned in section 4.2 is considered. The flow of this method is as follows; first, a set of rules is computed at each node in DFT by pattern matching. Next, a rule is selected from the set on condition that cost is minimum. For the sake of simplicity, we discuss the case of two data placed in a register. However, it is easy to extend this method to the case of three or more data placed in a register.

When a N–bit processor with SIMD instructions perform an operation on $\frac{N}{2}$–bit data, there are three options to execute the operation.

- Execute an instruction which performs on N–bit register
- Execute a SIMD instruction, where the operations perform on upper part of register
- Execute a SIMD instruction, where the operations perform on lower part of register

In the tree grammar, it is necessary to distinguish full registers as well as upper and lower subregisters. To represent the operation on upper and lower part of a register, additional non–terminals reg_hi and reg_lo are introduced. Using reg_hi and reg_lo, three operations mentioned above can be represented.

- Arithmetic and logical operations

  For example, 32–bit addition and upper and lower parts of SIMD addition are represented as follows.

$$\begin{aligned} reg &\rightarrow PLUS(reg, reg) \\ reg\_hi &\rightarrow PLUS(reg\_hi, reg\_hi) \\ reg\_lo &\rightarrow PLUS(reg\_lo, reg\_lo) \end{aligned}$$

  Other operations can be represented similarly to the example of addition.

- Loads and stores

  Similar to arithmetic and logical operations, 16–bit load operations are represented as follows.

$$\begin{aligned} reg &\rightarrow LOAD\_SHORT(addr) \\ reg\_hi &\rightarrow LOAD\_SHORT(addr) \\ reg\_lo &\rightarrow LOAD\_SHORT(addr) \end{aligned}$$

  16–bit store operations are represented as follows.

$$\begin{aligned} S &\rightarrow STORE\_SHORT(reg, addr) \\ S &\rightarrow STORE\_SHORT(reg\_hi, addr) \\ S &\rightarrow STORE\_SHORT(reg\_lo, addr) \end{aligned}$$

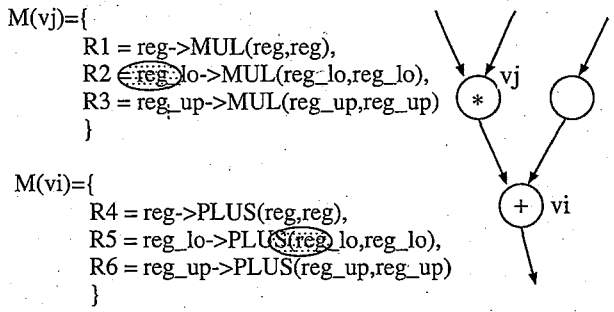


Figure 4.3: Consistency of nonterminals.

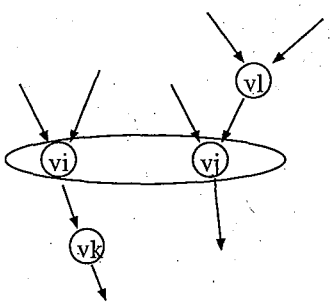


Figure 4.4: Schedulability.

- Common sub expressions

  The definition and the use of CSE are respectively represented as follows.

$$
\begin{aligned}
S &\rightarrow DEF\_SHORT\_CSE(reg) \\
S &\rightarrow DEF\_SHORT\_CSE(reg\_hi) \\
S &\rightarrow DEF\_SHORT\_CSE(reg\_lo)
\end{aligned}
$$

$$
\begin{aligned}
reg &\rightarrow USE\_SHORT\_CSE \\
reg\_hi &\rightarrow USE\_SHORT\_CSE \\
reg\_lo &\rightarrow USE\_SHORT\_CSE
\end{aligned}
$$

4.3.2 Constraints on selection of rules

In matching phase, a set of rule is annotated at each node. In the next phase, a rule is selected from the set, while the selection of rule have to be done under constraints as follows.

- Selection of single rule

  For each node v_i, exactly one rule has to be selected.

- Consistency of nonterminals

  Let v_j and v_k be children of v_i in a DFT. Here, a nonterminal which is left hand side of a rule is called target nonterminal. Each target nonterminal of the rule selected for v_j and v_k corresponded to argument of the rule selected for v_i has to be consist.

Fig. 4.3 shows an example of consistency of nonterminals. If R_2 is selected for v_i, R_5 have to be select for v_j.

- Common sub expressions

  Nonterminal of the rule selected for definition of CSE v_i and nonterminal of the rule selected for its use v_j must be identical.

- Node pairing

  When v_i is executed by a SIMD instruction, another node v_j which is executed by the identical SIMD instruction must be existed.

- Schedulability

  When we determine which nodes execute by SIMD instructions, data dependency between each pair should be considered. As shown in Fig. 4.4, if v_i and v_j are executed by an identical SIMD instruction, v_k and v_l cannot be execute at the same time.

4.3.3 ILP formulation

Let $V = \{v_1, ..., v_n\}$ be the set of DFG nodes, and let R_j be a set $M(v_i)$ of all rules matching v_i. Boolean solution variables x_{ij} is defined as follows:

$$x_{ij} = \begin{cases} 1, & \text{if } R_j \text{ is selected for } v_i \\ 0, & \text{other} \end{cases} \tag{4.1}$$

variables x_{ij} denotes which rule is selected for v_i from $M(v_i)$ after ILP is solved.

Let a pair of nodes (v_i, v_j) denote a SIMD pair if it holds below conditions.

- v_i and v_j can be executed in parallel. Namely, there is no path from v_i to v_j or from v_j to v_i in DFG.

- v_i and v_j represent same operation, and

- $M(v_i)$ contains a rule with target nonterminal reg_hi, and $M(v_j)$ contains a rule with target nonterminal reg_lo.

- If v_i and v_j are LOAD or STORE, which work on memory address p_i and p_j, then $p_i - p_j$ equal to the number of bytes occupied by the 16–bit value.

Boolean auxiliary variables y_{ij} is defined as follows:

$$y_{ij} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are executed at an identical SIMD instruction} \\ 0, & \text{other} \end{cases} \tag{4.2}$$

where variable y_{ij} denotes nodes which are executed by an identical SIMD instructions, and the result of the operation on v_i is stored to upper part of a destination register, the result of the operation on v_j is stored to lower part of a destination register.

Constraints described above are represent as follows.

- Selection of a single rule

  Since only one x_{ij} become 1 each v_i, this constraint represents as follows.

$$\forall v_i : \sum_{R_j \in M(v_i)} x_{ij} = 1 \tag{4.3}$$

- Consistency of target nonterminals

  Let $R_j \in M(v_i)$, $R_j = n_1 \rightarrow t(n_2, n_3)$ for a terminal t and nonterminals n_1, n_2, n_3, and let v_l and v_r be the left and right child of v_i. Let $M^N(v) \subseteq M(v)$ denote the subset of rules matching v that have N as the target nonterminal. If $R_j = n_1 \rightarrow t(n_2, n_3)$ is selected for v_i, then the rule chosen for v_l and v_r must have the target nonterminals n_2 and n_3 This constraint is represented as follows.

$$\forall v_i : \forall R_j \in M(v_i) \ : \ x_{ij} \leq \sum_{R_k \in M^{n_2}(v_l)} x_{lk} \tag{4.4}$$

$$\forall v_j : \forall R_j \in M(v_i) \ : \ x_{ij} \leq \sum_{R_k \in M^{n_3}(v_r)} x_{rk} \tag{4.5}$$

- Common subexpressions

Definitions of 16–bit CSEs and uses of 16–bit CSEs have been defined as follows.

$$\begin{aligned} R_1 = &\quad S &&\rightarrow DEF\_SHORT\_CSE(reg) \\ R_2 = &\quad S &&\rightarrow DEF\_SHORT\_CSE(reg\_hi) \\ R_3 = &\quad S &&\rightarrow DEF\_SHORT\_CSE(reg\_lo) \end{aligned}$$

$$\begin{aligned} R_4 = &\quad reg &&\rightarrow USE\_SHORT\_CSE \\ R_5 = &\quad reg\_hi &&\rightarrow USE\_SHORT\_CSE \\ R_6 = &\quad reg\_lo &&\rightarrow USE\_SHORT\_CSE \end{aligned}$$

Therefore, if v_i is definition of CSE and v_u is use of CSE, it is clear that $M(v_i) = \{R_1, R_2, R_3\}$ and $M(v_u) = \{R_4, R_5, R_6\}$. This constraint is represented as follows.

$$\forall v_i, v_u : x_{i1} = x_{u4},\ x_{i2} = x_{u5},\ x_{i3} = x_{u6} \tag{4.6}$$

- Node pairing

Let P denote the set of SIMD pairs. If $R_k \in M^{hi}(v_i)$ is selected for v_i, there must be v_j and $R_l \in M^{lo}(v_j)$ which holds $(v_i, v_j) \in P$. This condition is represented as follows.

$$\forall v_i : \sum_{R_k \in M^{hi}(v_l)} x_{ij} = \sum_{j:(v_i,v_j)\in P} y_{ij} \tag{4.7}$$

$$\forall v_i : \sum_{R_k \in M^{lo}(v_r)} x_{ij} = \sum_{j:(v_i,v_j)\in P} y_{ji} \tag{4.8}$$

- Schedulability

Let $X(v)$ denote a set of nodes that must be executed before v, and let $Y(v)$ denote a set of nodes that must be executed after v. If $(v_i, v_j) \in P$, then a set Z_{ij} defined below have to be empty.

$$Z_{ij} = P \cap (X(v_i) \times Y(v_j) \cup X(v_j) \times Y(v_i)) \tag{4.9}$$

This constraint is represented as follows.

$$\forall (v_i, v_j) \in P : \forall (v_k, v_l) \in Z_{ij} : y_{ij} + y_{kl} \leq 1 \tag{4.10}$$

- Objective function

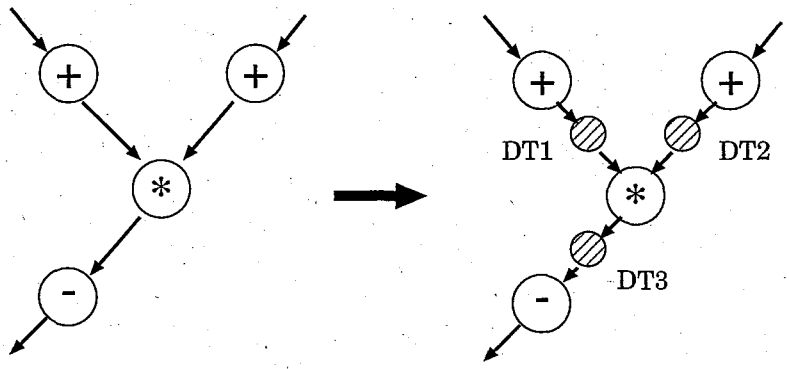


Figure 4.5: Nodes insertion for data transfers.

The optimization goal is to make the maximum use of SIMD instructions. Since target nonterminals of the rules for SIMD instructions are *reg_hi* or *reg_lo*, the objective function is represented as follows.

$$f = \sum_{v_i \in V} \Big(\sum_{R_k \in M^{hi}(v_i) \cup M^{lo}(v_i)} x_{ik} \Big) \tag{4.11}$$

4.4 SIMD Instruction Formulation with permutation Instructions

In this section, the proposed method is explained. The proposed method is extended from the Leupers's method [15]. Data transfer for SIMD instructions is considered in instruction selection of compiler. The following sections explain in detail.

4.4.1 IR and Rules for Data Packing and Moving

To represent data transfers on DFT, nodes that represent data transfer operations are introduced. Since candidates of data transfers appear between operations, nodes for data transfers are inserted between all operations. Fig. 4.5 shows nodes insertion for data transfer. DT1, DT2 and DT3 are added to DFT. Moreover, the rules of data transfer are also introduced. When a processor executes a permutation instruction, there are three conditions according to the locations where data exist.

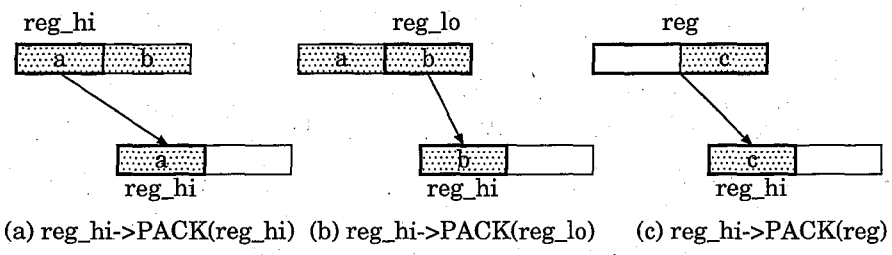


Figure 4.6: Rule of permutation instructions.

- Two values are located in a register. The value that would be packed is in the upper part of the register.
- Two values are located in a register. The value that would be packed is in the lower part of the register.
- A value is located in a register

These three conditions are shown in Fig.4.6. Fig. 4.6(a) shows a data transfer from upper part of a source register to upper part of a destination register. To represent permutation instructions, terminal $PERM$ is used. Fig. 4.6(a) represents the rule $reg\_hi \rightarrow PERM(reg\_hi)$. Similarly, Fig. 4.6(b) represents the rule $reg\_hi \rightarrow PERM(reg\_lo)$. Fig. 4.6(c) shows a data transfer from source register occupied by a value to upper part of a destination register. Fig. 4.6(c) represents the rule $reg\_hi \rightarrow PERM(reg)$. Data transfer to the lower part of destination register is represented as same as the case of data transfer to the upper part mentioned above. These conditions for permutation instructions are formulated as additional rules shown below.

$$
\begin{aligned}
reg\_lo &\rightarrow PERM(reg\_lo) \\
reg\_hi &\rightarrow PERM(reg\_lo) \\
reg\_lo &\rightarrow PERM(reg\_hi) \\
reg\_hi &\rightarrow PERM(reg\_hi) \\
reg\_lo &\rightarrow PERM(reg) \\
reg\_hi &\rightarrow PERM(reg),
\end{aligned}
$$

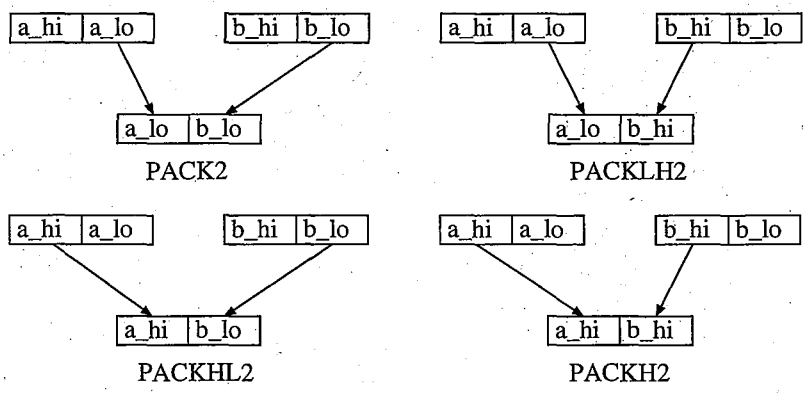


Figure 4.7: Example of permutation instructions.

where a permutation instruction consists of two rules : one has reg_hi as a target nonterminal, and the other has reg_lo as a target nonterminal.

For example, consider four permutation instructions which TMS320C62x [4] have shown in Fig. 4.7. Using the rules introduced above, permutation instructions are represented. PACKH2 consists of two data transfers, one is from upper part of source register to upper part of destination register, and the other is from upper part of source register to lower part of destination register. Former data flow is represented by $reg\_hi \rightarrow PERM(reg\_hi)$, and latter is represented by $reg\_lo \rightarrow PERM(reg\_hi)$, therefore, PACKH2 instruction can be represented by a pair of rules, $reg\_hi \rightarrow PERM(reg\_hi)$ and $reg\_lo \rightarrow PERM(reg\_hi)$.

Moreover, the rule for UNPACK which is a instruction that moves a value located upper or lower part of a register into a register is adopted. Those rules are represented as follows.

$$reg \rightarrow UNPACK(reg\_lo)$$
$$reg \rightarrow UNPACK(reg\_hi)$$

In addition, the rules indicates no operation called "NOMOVE" is introduced.

$$reg \rightarrow NOMOVE(reg)$$
$$reg\_lo \rightarrow NOMOVE(reg\_lo)$$
$$reg\_hi \rightarrow NOMOVE(reg\_hi)$$

These rules are selected when it is not necessary to move data.

$PERM$ and $UNPACK$ have some cost since actual instructions are executed if they are selected. However, $NOMOVE$ have no cost since that is corresponded to no actual instruction.

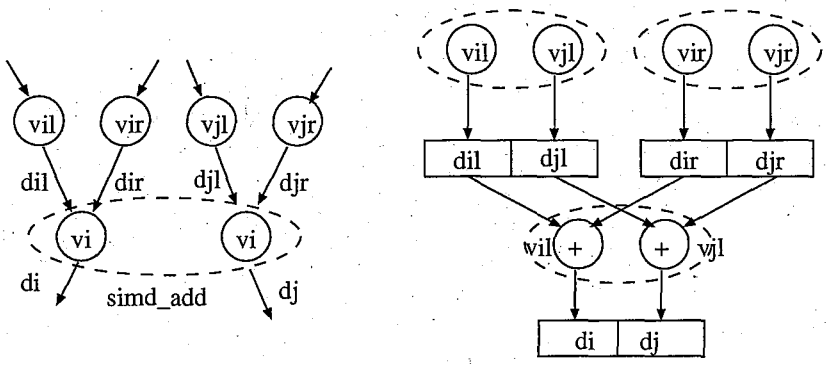


Figure 4.8: Identification of a register which source values located.

4.4.2 Constraints on selection of rules

According to additional DFT nodes and rules, the following constraints has to be considered.

- Node pairing for permutation instructions

  $PERM$, $UNPACK$ and $NOMOVE$ match DFT nodes for data transfer. Those rules must be selected under constraints shown below.

  - If $PERM$ is selected for v_i, another node v_j that is selected as PERM must exist, and they are executed an identical PERM instruction.
  - If $UNPACK$ is selected for v_i, there is no node executed with v_i.
  - If $NOMOVE$ is selected for v_i, even if a target nonterminal is *reg_hi* or *reg_lo*, v_i is not paired to other nodes because behavior of $NOMOVE$ does not depend on other part of a register. However, when SIMD instructions are executed successively, the nodes for data transfers between SIMD instructions must be selected as $NOMOVE$ and must be paired them.

- Identification of a register which source values located

  When a SIMD instruction is executed, left arguments have to be located in an identical register, and right arguments also have to be located in the source register. Fig. 4.8 shows an example of identification of registers. Each result of v_{i_l} and v_{j_l}, and v_{i_r} and v_{j_r} must be located in an identical register to perform v_i and v_j as a SIMD instruction.

4.4.3 ILP Formulation

In this section, ILP formulation for permutation instructions is explained.

- Node pairing for permutation

  Boolean auxiliary variables a_{ij} and b_{ij} are defined as follows:

$$a_{ij} = \begin{cases} 1, & v_i \text{ and } v_j \text{ are executed an identical } PERM \text{ instruction} \\ 0, & \text{other} \end{cases}$$

$$b_{ij} = \begin{cases} 1, & v_i \text{ and } v_j \text{ are stayed in an identical register} \\ 0, & \text{other} \end{cases}$$

  Let V_{MOVE} denote a set of nodes for data transfers, and let $M^N_{OP}(v)$ denote a subset of rules in $M^N(v)$ that have OP as the terminal OP. This constraint is represented as follows.

$$\forall v_i \in V_{MOVE} \ : \sum_{R_k \in M^{hi}_{PERM}(v_i)} x_{ik} = \sum_{j:(v_i,v_j)\in P} a_{ij} \tag{4.12}$$

$$\forall v_i \in V_{MOVE} \ : \sum_{R_k \in M^{lo}_{PERM}(v_i)} x_{ik} = \sum_{j:(v_j,v_i)\in P} a_{ji} \tag{4.13}$$

$$\forall v_i \in V_{MOVE} \ : \sum_{R_k \in M^{hi}_{NOMOVE}(v_i)} x_{ik} \geq \sum_{j:(v_i,v_j)\in P} b_{ij} \tag{4.14}$$

$$\forall v_i \in V_{MOVE} \ : \sum_{R_k \in M^{lo}_{NOMOVE}(v_i)} x_{ik} \geq \sum_{j:(v_j,v_i)\in P} b_{ji} \tag{4.15}$$

  According to the definition of y_{ij} , a_{ij} , and b_{ij}, following constraint is needed.

$$\forall y_{ij} \in V_{MOVE} : y_{ij} = a_{ij} + b_{ij} \tag{4.16}$$

- Identification of a register which source values located

  Let v_{i_l} and v_{i_r} be left and right children of v_i, v_{j_l} and v_{j_r} be left and right children of v_j. In order to execute a SIMD instruction for v_i and v_j, results of v_{i_l} and v_{j_l}, and v_{i_r} and v_{j_r} must be located in a register. When v_{i_l} and v_{j_l} are executed by an identical SIMD instruction, the results of v_{i_l} and v_{j_l} are stored to a register. Therefore, to execute a

SIMD instruction for v_i and v_j, v_{i_l} and v_{j_l}, and v_{i_r} and v_{j_r} must be executed by a SIMD instruction. y_{ij} denotes that SIMD instructions is executed for v_i and v_j. This constraint is represented as follows.

$$\forall(v_i, v_j) \in P, v_i \in V : y_{ij} \leq y_{i_l j_l} \tag{4.17}$$

$$\forall(v_i, v_j) \in P, v_i \in V : y_{ij} \leq y_{i_r j_r} \tag{4.18}$$

- Objective function

  The optimization goal is to minimize code size. Consider variables x_{ij}, y_{ij} for arithmetic, logical operation and load/store, y_{ij} corresponds to a SIMD instruction, and x_{ij} for the rule which has reg as a target nonterminal corresponds to an instruction. On the other hand, if variables x_{ij}, a_{ij}, b_{ij} represent data transfer operations, a_{ij} corresponds to a permutation instruction, x_{ij} for $UNPACK$ corresponds to a data transfer operation, and x_{ij}, b_{ij} for $NOMOVE$ corresponds to no instruction. Let P_{MOVE} denote a set of pairs of nodes for data transfer, and code size can be represented as follows.

$$\begin{aligned} f = & \sum_{v_i \in V - V_{MOVE}} \sum_{R_k \in M^{reg}(v_i)} x_{ik} + \sum_{(v_i, v_j) \in P - P_{MOVE}} y_{ij} \\ & + \sum_{v_i \in V_{MOVE}} \sum_{R_k \in M^{reg}_{UNPACK}(v_i)} x_{ik} + \sum_{(v_i, v_j) \in P_{MOVE}} a_{ij} \end{aligned} \tag{4.19}$$

4.5 Experimental results

The proposed formulation was implemented by using CoSy compiler development environment [51] on RedHat Linux 8.0. For evaluation, a DLX based processor which had DLX instruction set without floating point arithmetic operation, but had SIMD instructions, such as ADD2, MULT2, and several permutation instructions. ADD2 instruction performs two additions on 16–bit values, MULT2 instruction which two multiplications on 16–bit values, and a variety of permutation instructions are PACKL, PACKLH, PACKHL and PACKHH. To compare the quality of generated code, three compilers were used: (1) a compiler generated by the compiler generator of ASIP meister [52] (2) a compiler applied the Leupers's method based

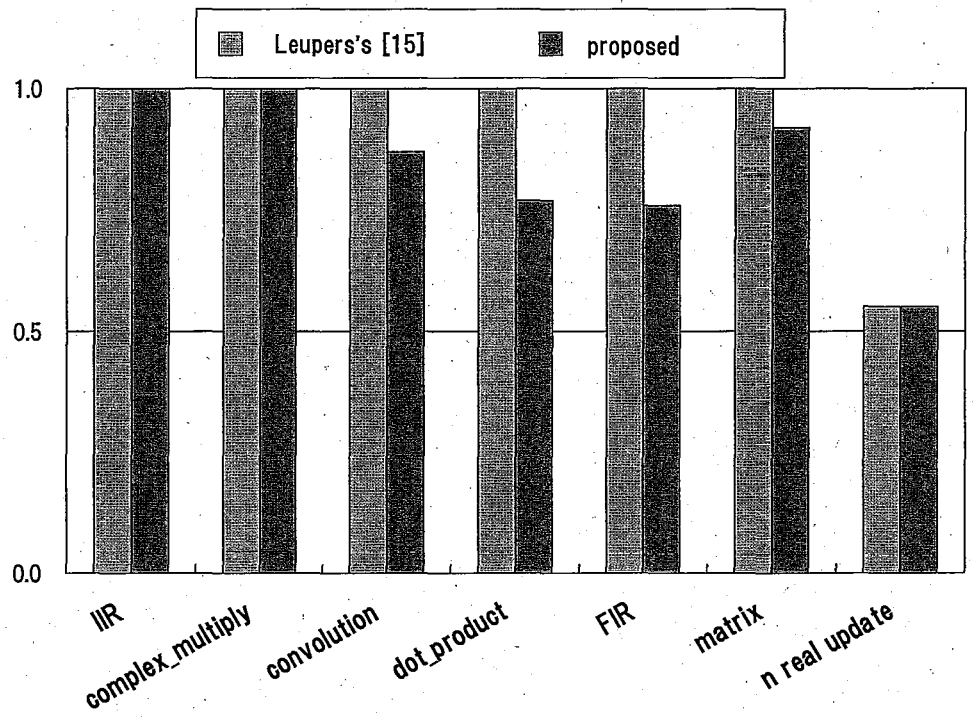


Figure 4.9: The ratio of generated code size.

on (1)'s compiler, and (3) a compiler applied the proposed method based on (1)'s compiler. Programs for evaluation which consists of iir_biquad_one_section, complex_multiply, convolution, dot_product, fir, matrix and n_real_updates were selected from DSPstone benchmark [47]. Original codes such as convolution, dot_product, fir, matrix, n_real_updates were unrolled easily extract parallel executions.

Table 4.1 shows generated code size and the number of execution cycles of each program compiled by each compiler. Figs. 4.9 shows the ratio of code size generated by (2) and (3) to generated by (1), and Fig. 10 shows the ratio of execution cycles of generated code. Table 4.2 shows the number of nodes of DFT, the number of variables and constraints in ILP and CPU time.

In Fig. 4.9 and Fig. 4.10 comparing the Leupers's method with no SIMD the Leupers's method was effective in only n_real_updates. However, the proposed method reduced code

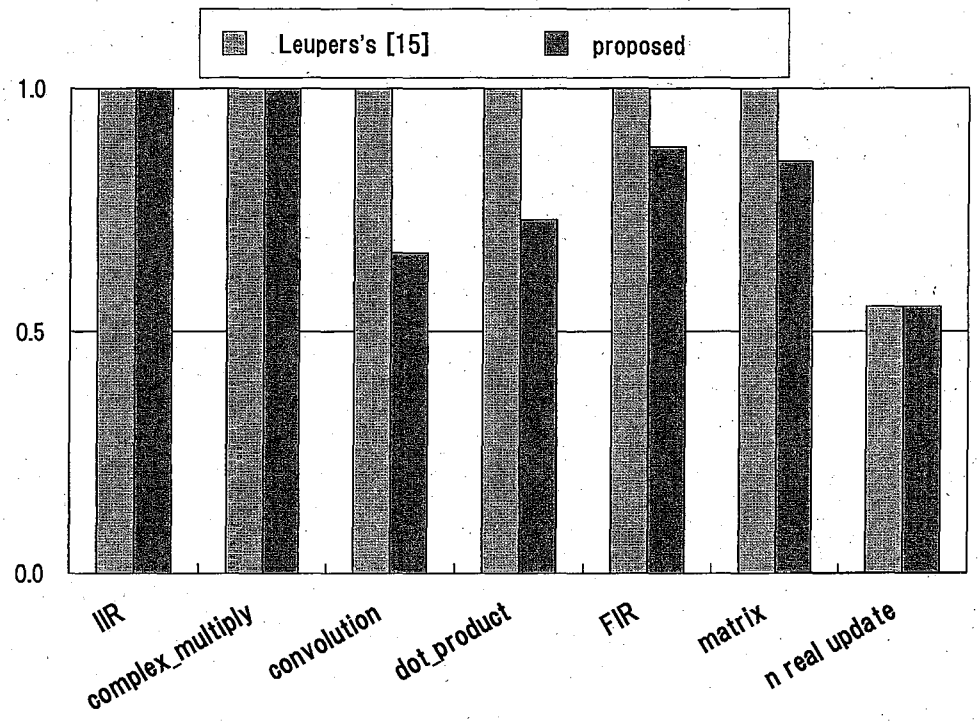


Figure 4.10: The ratio of execution cycles.

size and execution cycles in convolution, dot_product, FIR, matrix, and n_real_updates. The Leupers's method can select SIMD instructions the case where a sequence of instructions consists of SIMD instructions only because the Leupers's method does not consider data transfer. However, such conditions are not often filled. On the other hand, the proposed method inserts data transfer instructions when SIMD instructions can be applied by moving values, or unpacked data is required. Actually, in convolution, the proposed method selected a permutation instruction to adapt the location of values for SIMD multiplication instruction and select it.

In Table 4.2, comparing the Leupers's method and the proposed method, the proposed method takes much more time to solve ILP. This is because the proposed method has wider solution space than the Leupers's method. Therefore, the proposed method spends much time to get an optimum solution. However. the proposed method can select SIMD instructions effectively. The code size of the proposed method is smaller than that of the Leupers's method, and execution cycles of the proposed method is smaller than that of the Leupers's method.

Table 4.2 shows that the proposed method compiles 6 programs within a minite. However, the proposed method takes more than 5000 seconds to compile FIR. Because it is proved that

Table 4.1: Generated code size and execution cycles.

program	no SIMD optimization		Leuper's method		proposal method	
	code size	execution cycles	code size	execution cycles	code size	execution cyclyes
iir_biquad_N_section	132	420	132	420	132	420
complex multiply	126	562	126	562	126	562
convolution	62	784	62	784	54	514
dot product	57	162	57	162	44	118
FIR	88	828	88	828	67	365
matrix	137	5268	137	5268	127	4458
n real update	95	1162	53	634	53	634

the ILP belongs to NP complete problem [53], it is expected that the compilation time for large programs will be very long. The compilation time depends on not only the size of programs but also characteristics of programs. Characteristics of programs reflect the number of variables, the number of constraints and the solution space in an instance of ILP. Comparing the convolution and n real update in Table 4.2, the number of nodes and the number of constraints of n real update are smaller than convolution. However, the compilation time of n real update is longer than convolution. This indicates that the instance of ILP of convolution can be solved more easily than n real update. Fig. 4.11 and 4.12 show the program fragments of convolution and n real update. In Fig. 4.11 and 4.12, 4 additions and 4 multiplications can be found for each programs. However, only multiplications are executable in parallel in 4.11, while additions are also executable in parallel in 4.12. This means the constraints in the instance of ILP of convolution is tighter than n real updates. As a result, the optimum solution of convolution can be found in shorter time than n real update.

There are some points to be considered to shorten compilation time. The first point is that the input DFGs were not modified before ILP formulation in this experiments. Generally, there are several DFG representations for a specific program fragment. Processor independent optimization techniques such as sharing common sub expression, redundant expression elimination

Table 4.2: The number of DFT nodes, variables and constraints in ILP and CPU time.

program	Leupers's method				proposed method			
	# of nodes	# of var.	# of cons.	CPU time [sec]	# of nodes	# of var.	# of cons.	CPU time [sec]
iir_biquad_N_section	40	189	190	0.11	69	2304	7974	0.99
complex multiply	16	62	69	0.09	30	776	1789	0.18
convolution	34	149	174	0.09	60	2062	7504	1.99
dot product	18	67	88	0.08	32	704	1522	0.18
fir-	48	305	627	0.17	81	3660	20097	5679.00
matrix	6	21	25	0.12	10	92	101	3.79
n real update	28	129	137	0.12	51	2166	4557	22.72

```
y+=px[i+0]*ph[N-1-i+3];
y+=px[i+1]*ph[N-1-i+2];
y+=px[i+2]*ph[N-1-i+1];
y+=px[i+3]*ph[N-1-i+0];
```

Figure 4.11: convolution.

```
p_d[i+0]=p_c[i+0]+p_a[i+0]*p_b[i+0];
p_d[i+1]=p_c[i+1]+p_a[i+1]*p_b[i+1];
p_d[i+2]=p_c[i+2]+p_a[i+2]*p_b[i+2];
p_d[i+3]=p_c[i+3]+p_a[i+3]*p_b[i+3];
```

Figure 4.12: n real update.

reduce the number of nodes in DFGs. If the number of nodes in DFGs decreases, the compilation time can be shorten because the size of the instance of ILP becomes small. The second point is that the instance of ILP is solved without any modification. Redundant variables and constraints may be found in the instance of ILP which is derived by the proposed ILP formulation. If variables or constraints can be reduced by some analysis before solving the instance of ILP, ILP solver may solve the modified problem in shorter time comparing with the case to solve the original instance of problem.

4.6 Summary

In this chapter, a code selection method for SIMD instructions considering data transfer has proposed. In the proposed method, nodes for data transfer has been added to DAGs, and rules for data transfer has been introduced. Similar to the Leupers's method, code selection problem was formulated into ILP, and the problem was solved by using ILP solver. Experimental results show that the proposed method can generate more efficient codes than the Leupers's method, which use data transfer instructions to exploit SIMD instructions.

Chapter 5

Efficient Code Generation Algorithm for Media Instructions

This chapter describes the fast code generation method for media processors. The organization of this chapter is as follows: The way to find SIMD operations in high level language program is described in section 5.1. Permutation instruction generation based on MDDs is presented in section 5.2. Experimental results are shown in section 5.3. This chapter is concluded in section 5.4.

5.1 Generation of SIMD Instructions

The proposed code generation approach mainly consists of two parts. The first part is SIMD instruction generation, and the second part is permutation instruction generation.

In the first part, using tree matching in [50] and [15], a data flow graph (DFG) whose nodes are elements of SIMD operations is constructed. After the DFG construction, the DFG is divided into data flow trees (DFT), then operations are grouped into SIMD instructions. Finally, a DFG whose nodes are SIMD instructions is constructed. In this section, grouping of SIMD operations is explained. Pattern matching, DFG construction and DFT construction are similar to [1].

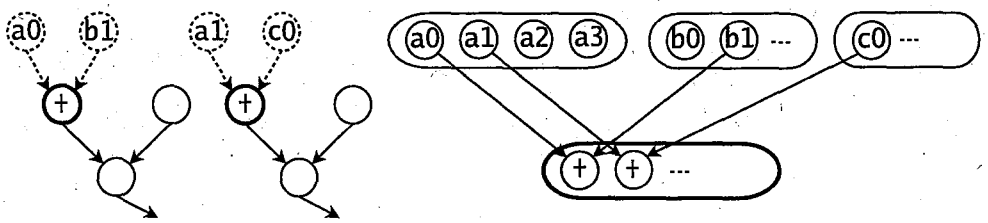


Figure 5.1: Operation grouping

5.1.1 Grouping SIMD Operations

Groups of operations performed by SIMD instructions are determined as follows. First, leaves of DFTs which have the same operations are selected. Then, if the number of the selected nodes is less than the number of operations that one SIMD instruction can perform, the selected nodes are grouped as a SIMD instruction. If not, the selected nodes are divided into smaller groups whose number of elements is less than the number of operations of a SIMD instruction. Finally, the nodes in the selected group are removed from the DFTs. This grouping is repeatedly processed until all nodes are removed from DFTs. In this process, nodes corresponding to load and store operations that can be executable by one SIMD instruction are restricted by its memory address. Since misaligned memory access is unavailable or cause large penalty cycles, consecutive and aligned memory access operations are grouped as SIMD instructions. Fig.5.1 shows an example of operation grouping. The load operations have been removed from DFTs as shown in Fig.5.1(a). The add operations, nodes with a plus operator, are grouped and added to the DFT in Fig.5.1(b). Then, the add nodes will be removed from Fig.5.1(a). After this grouping, the DFT in Fig5.1(b) is constructed. This process continues until all nodes are removed from DFTs.

5.1.2 Ordering SIMD Operations in Registers

The order of operations in a register is determined as follows. The load and store operations are uniquely ordered by the memory address accessed by operations, because the available group of memory access operation is limited by the memory address and alignment as mentioned in

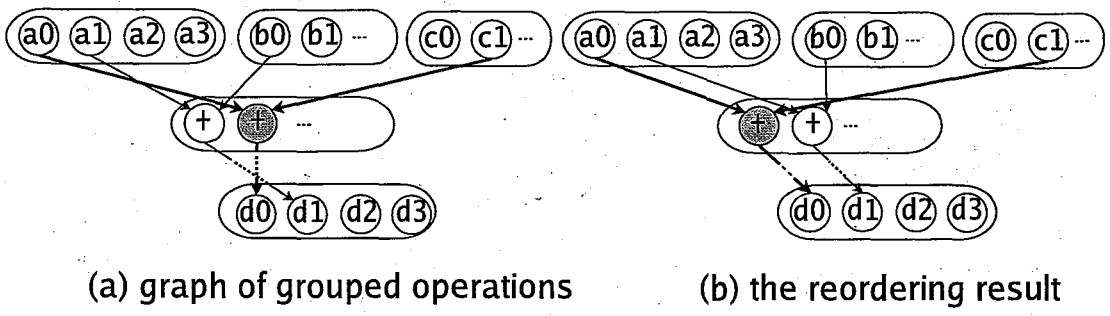


Figure 5.2: Operation ordering

section5.1.1. The order of operations except for load and store is determined by the order of load and store operations. The most frequently used position where sources and destinations are arranged is selected for each operation. Fig.5.2 shows the example of operation ordering. In Fig.5.2(a), a part of grouped graph is shown. The most left add operation has two sources `a1` and `b0`, one destination `d1`. The order in the grouped node of `a1` is the second element, `b0` is the first and `d1` is the second. Therefore, the most left add operation in Fig.5.2(a) is reordered to the second in the grouped node as shown in Fig.5.2(b). Similarly the second add operation is reordered to the most left in the grouped node.

5.2 Generation of Permutation Instructions

This section describes how the permutation instructions are generated. Hereafter, the contents of packed data are called *permutation* because they are naturally represented by permutations. The generation method of permutations consists of two steps. In the first step, it is examined whether the target permutation can be generated using given permutation instructions. The basic concept of the first step is to generate all permutations from source permutations using available permutation instructions. In the second step, the expression tree representing the construction process of the target permutations from input permutations is generated. The tree construction starts from the root, which is the node corresponding to the target permutations, and the tree grows from the root to the leaf by adding nodes representing permutation instructions. In the permutation instruction generation, Multi-valued Decision Diagram (MDD) [54] is

utilized to represent and manipulate the sets of permutations. Using MDD, a permutation operation can be manipulated not on a pair of permutations but on pairs of sets of permutations. This characteristic enables efficient generation of permutations. In the rest of this section, MDDs are introduced first. Then the permutation instruction generation algorithm is described.

5.2.1 Introduction of MDDs for Representation of a Set of Permutations

In this section, representation of the set of permutations using Multi-valued Decision Diagram(MDD) is introduced.

Consider the register which has n elements of packed data. Let $S = \{s_1, s_2, \ldots\}$ be the set of given sub-word data. Let $r \in S^n$ be the permutation representing the content of a register.

Let $R \in 2^{S^n}$ be the set of permutations. When a set of permutations R and a permutation r are given, a function $F_R : S^n \rightarrow \{0, 1\}$ is defined as follows :

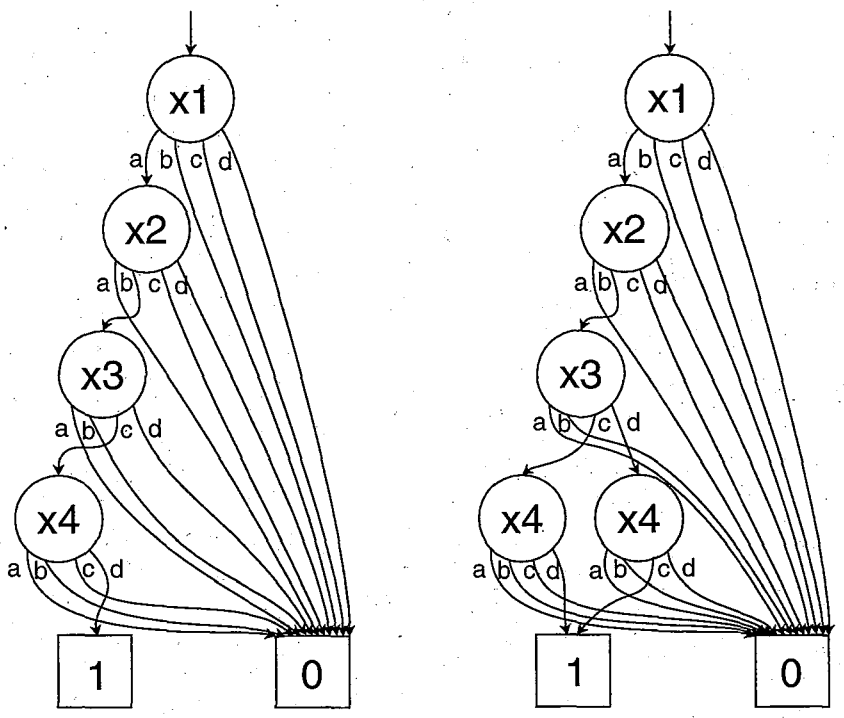
$$F_R(r) = \begin{cases} 1, & \text{if } r \in R \\ 0, & \text{if } r \notin R \end{cases} \tag{5.1}$$

According to the definition, the function F_R implicitly represents the set of permutations R.

Here, variables $x_1, .., x_n$ whose domain is S are introduced . Assume x_i to be the ith element of r which is the input of F_R. Using x_i, the equation(5.1) can be expressed as follows :

$$F_R(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } (x_1, \ldots, x_n) \in R \\ 0, & \text{if } (x_1, \ldots, x_n) \notin R \end{cases} \tag{5.2}$$

In the equation(5.2), F_R is defined as the multi-valued input, binary-valued output function. Such functions can be represented by Multi-valued Decision Diagram. Fig.5.3 shows two MDDs for {abcd} and {abcd,abdc}. In Fig.5.3(a), the only one path exists from the root to 1-terminal through the edges a,b,c,d. On the other hand, in Fig.5.3(b), there are two paths exist from the root to 1-terminal through a,b,c,d and a,b,d,c. Considering the sequences of the labeled symbols on edges as elements in a set, a set of permutations can be represented by an MDD. Moreover, some MDD manipulations correspond to operations on the sets of permutations. The logical-or operation on MDD corresponds to the union operation on the set of



(a) { abcd } (b) { abcd, abdc }

Figure 5.3: MDDs for { abcd } and { abcd,abdc }

permutations. Similarly, logical-and operation on MDD corresponds to the intersection operation. For example, the MDD shown in Fig.5.3(b) is constructed by the logical-or of MDDs representing $\{\texttt{abcd}\}$ and $\{\texttt{abdc}\}$, which corresponds to the union of $\{\texttt{abcd}\}$ and $\{\texttt{abdc}\}$.

5.2.2 Permutation Operation Manipulation on MDDs

Using MDDs, basic operations such as union and intersection can be applied to permutations. Similar to such basic operations, permutation operations can also be performed on MDDs.

Consider a permutation operation p which takes two permutations r_1, r_2, and returns a permutation r_3. Let $r_i(j)$ be the jth elements of a permutation r_i. Let $\sigma(k)$ be a function defined by $\sigma(k) = (i_k, j_k)$, $i_k \in \{1, 2\}$, $j_k \in \{1, \ldots, n\}$ for $k = 1, \ldots, n$. Let $q_\sigma(k)$ be the j_kth element of the i_kth input permutation of p_σ. Given a function σ, a permutation operation $p_\sigma(r_1, r_2)$ is

defined as follows :

$$\begin{aligned} p_\sigma(r_1, r_2) &= (q_\sigma(1), \ldots, q_\sigma(n)) \\ &= (r_{i_1}(j_1), \ldots, r_{i_n}(j_n)) \end{aligned} \tag{5.3}$$

Let P_σ be the permutation operation on sets of permutations R_1 and R_2. $P_\sigma(R_1, R_2)$ is defined as follows :

$$P_\sigma(R_1, R_2) = \bigcup_{(r_1, r_2): r_1 \in R_1, r_2 \in R_2} \{\ p_\sigma(r_1, r_2)\ \} \tag{5.4}$$

The result of $P_\sigma(R_1, R_2)$ is a set of permutations whose elements are results of p_σ on any pairs of elements of input sets.

The direct computation of equation(5.4) is hard when $|R_1|$ and $|R_2|$ are large. However, using MDDs, permutation operations on sets of permutations are effectively manipulated. In other words, there is no need to execute the permutation operation for each pair. The way to compute $P(R_1, R_2)$ using MDDs consists of three primitive manipulations.

1. For each R_i, make R_i' from R_i by adding all permutations whose elements at the position where the permutation operation works are the same as that of any of permutations in Ri.

   This computation on MDDs is simply implemented. Every node which corresponds to unused element is replaced with the union of its children as shown in Fig.5.4.

2. For each R_i', make R_i'' by reordering the elements of all permutations in R_i' to match with the order of the output of the permutation operation.

   This computation on MDDs is almost same as the conventional variable ordering technique for decision diagrams. The difference between this reordering and conventional variable ordering is that the level of variable is not changed in this reordering whereas it is changed in the conventional variable ordering. Fig.5.5 shows reordering of elements on MDDs.

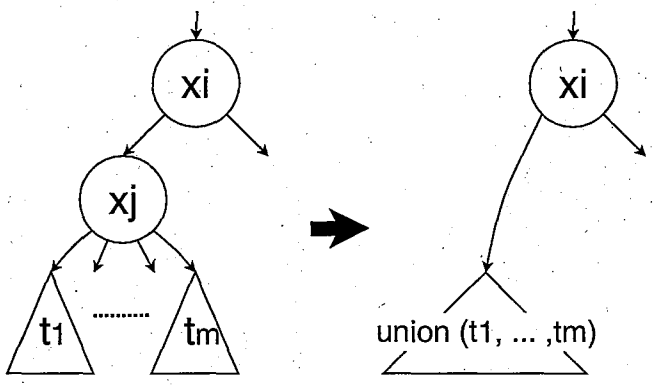


Figure 5.4: Adding permutations on MDDs

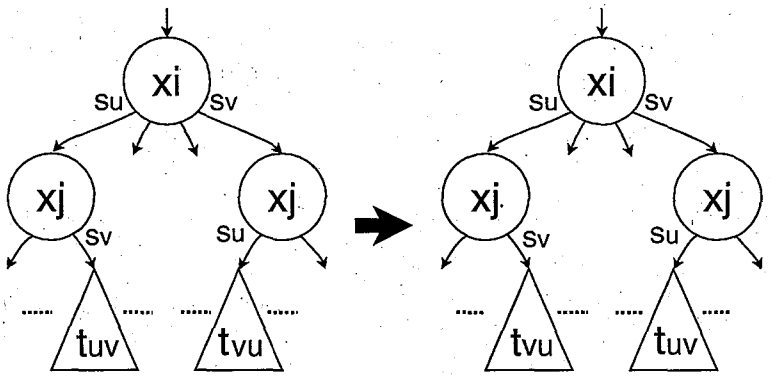


Figure 5.5: Reordering on MDDs

3. Finally, $P(R_1, R_2)$ is obtained by computing intersection of R_1'' and R_2''. The intersection operation corresponds to the logical-and operation on MDDs.

For the explanation of the permutation operation manipulation, consider a permutation operation p shown in Fig.5.6. Assume the input sets of permutations are $R_1 = \{\texttt{abcd}\}$ and $R_2 = \{\texttt{dcba}\}$. The elements of R_1' are all permutations matching $\texttt{ab}{*}{*}$. As a result, R_1' is obtained as follows :

$$\begin{aligned} R_1' = \{&\texttt{abaa}, \texttt{abba}, \texttt{abca}, \texttt{abda}, \\ &\texttt{abab}, \texttt{abbb}, \texttt{abcb}, \texttt{abdb}, \\ &\texttt{abac}, \texttt{abbc}, \texttt{abcc}, \texttt{abdc}, \\ &\texttt{abad}, \texttt{abbd}, \texttt{abcd}, \texttt{abdd}\} \end{aligned}$$

Similarly, the elements of R_2' are all permutations matching $\texttt{dc}{*}{*}$. In the second step of

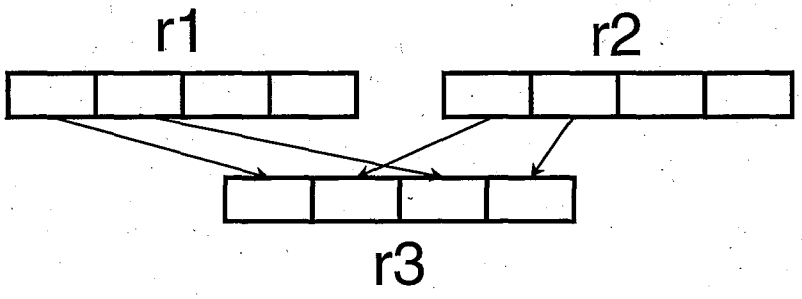


Figure 5.6: An example permutation instruction

the permutation operation manipulation, elements in R'_1 and R'_2 are reordered according to the permutation operation. The elements of R''_1 and R''_2 are all permutations matching `a*b*` and `*d*c` respectively. Finally, the intersection of R''_1 and R''_2 is computed. The intersection of R''_1 and R''_2 is the set of permutations matching both `a*b*` and `*d*c`. As a result, `{ adbc }` is obtained for this example.

In the example shown in this section, the length of permutation is 4, and the input sets of permutations have only one element. However, it is clear that these are not restrictions, because such parameters are independent of those manipulations.

5.2.3 Permutation Instruction Generation Algorithm

In this section, the permutation instruction generation algorithm is explained.

The inputs of the algorithm are the set of permutations R_0, a required permutation r_o and a set of permutation operations P. The output is an expression tree whose operations are $P \in \mathrm{P}$ and leaves are $r \in R_0$. The result of evaluation of the tree have to be r_o. Note that the subscript of R is used to distinguish among the variants of the set of permutations generated in this algorithm though R_i means the ith input of the permutation operation in the section3.

The permutation instruction generation algorithm consists of two sub-procedures.

1. Examine whether the required permutation can be generated using the given permutation operations.

2. Build the expression tree whose intermediate nodes are permutation operations, leaves

CanGeneratePermutation($R_0, \mathbf{P}, r_o$)

1: $i \leftarrow 0$
2: **while** $r_o \notin R_i$ **do**
3: $\quad \forall P_j \in \mathbf{P} : R_{i+1,j} \leftarrow P_j(R_i, R_i)$
4: $\quad R_{i+1} \leftarrow (\bigcup_j R_{i+1,j}) \cup R_i$
5: $\quad$ **if** $R_{i+1} = R_i$ **then**
6: $\quad\quad$ **return** false
7: $\quad$ **end if**
8: $\quad i \leftarrow i + 1$
9: **end while**
10: $n_{depth} \leftarrow i$
11: **return** true

Figure 5.7: Testing Target Permutation Generation

are input permutations.

The first sub-procedure *CanGeneratePermutation* is shown in Fig.5.7. The basic concept of *CanGeneratePermutation* is to generate all permutations from source permutations using available permutation operations until r_o is generated. The main process is the while loop in the lines from 2 to 9. The variable i is initialized to 0 and incremented for every iteration. R_i holds all permutations generated in $0, \ldots, i$ th iterations. In the lines 3 and 4, R_{i+1} is made from R_i by adding permutations generated by available permutation operations. In the line 5, R_{i+1} and R_i are compared. If R_{i+1} is equal to R_i, this sub-procedure will finish and return "false" since it means that no more permutations can be generated and the required permutation could not be generated by available permutation operations. Until the required permutation is generated or no other permutations can be generated, the while loop is executed repeatedly. When this sub-procedure finished, the number of iterations is obtained as a constant n_{depth}. The constant n_{depth} and the sets of permutations $R_1, \ldots, R_{n_{depth}}$ generated in this sub-procedure are also used in the second sub-procedure.

GetExpressionTree($R^{require}, i$)

1: **if** $R^{require} \cap R_0 \neq \phi$ **then**
2: **return** a leaf corresponds to $r \in R^{require} \cap R_0$
3: **end if**
4: **if** $R^{require} \cap R_i = \phi$ **then**
5: **return** nil
6: **end if**
7: **for all** $P_j \in \mathrm{P}$ **do**
8: $(R_j^{require,l}, R_j^{require,r}) \leftarrow P_j^{-1}(R^{require})$
9: $T_j^l \leftarrow$ *GetExpressionTree*($R_j^{require,l}, i-1$)
10: $T_j^r \leftarrow$ *GetExpressionTree*($R_j^{require,r}.i-1$)
11: **if** $T_j^l \neq nil$ and $T_j^r \neq nil$ **then**
12: $T_j \leftarrow$ a tree with P_j as root, subtrees are T_j^l and T_j^r
13: **else**
14: $T_j \leftarrow nil$
15: **end if**
16: **end for**
17: **if** $\forall T_j : T_j \neq nil$ **then**
18: **return** T_j such that the number of nodes is minimal
19: **else**
20: **return** nil
21: **end if**

Figure 5.8: Expression Tree Construction

The second sub-procedure *GetExpressionTree* is shown in Fig.5.8. The inputs are a set of permutations $R^{require}$ and an integer i. An expression tree representing the expression to generate one of the elements in $R^{require}$ is returned. The second input i indicates the depth of tree to be built. The depth of obtained tree will be less than equal to i. This sub-procedure constructs an expression tree recursively. At the start, *GetExpressionTree* is invoked with $i = n_{depth}$ and $R^{require} = \{r_o\}$. In Fig.5.8, the lines from 1 to 3, a leaf of permutation in $R^{require} \cap R_0$ is returned if $R^{require}$ includes any permutations in R_0. In the lines from 4 to 6, nil is returned if the condition is satisfied since the condition indicates that no required permutation is in R_i. In the lines from 7 to 16, for each permutation operation P_j, a tree whose root is P_j is constructed. P_k^{-1} is the inverse permutation operation. In the line 8, P_k^{-1} returns a pair of sets of permutations $(R_j^{require,l}, R_j^{require,r})$ which is the source of $R^{require}$. In the lines 9 and 10, *GetExpressionTree* is recursively invoked to build the left and right subtrees, T_k^l and T_k^r. In the lines from 11 to 15, If both T_k^l and T_k^r are not nil, a tree T_k whose root is P_k, and the subtrees are T_k^l and T_k^r is built. Finally, in the lines from 17 to 21, T_k which has minimal cost is returned. If any T_k is nil, nil is returned.

In *GetExpressionTree*, the function recursively called itself $2 \cdot |\mathrm{P}|$ times. Therefore, *GetExpressionTree* is called $(2 \cdot |\mathrm{P}|)^{n_{depth}}$ times in the worst case. However, the lines from 4 to 6 in *GetExpressionTree* check whether it is possible to generate necessary permutations, and redundant subtree construction is pruned if it is not possible. Since sub expression trees which are not the part of feasible expression trees are not searched, computation time of *GetExpressionTree* practically depends on the number of feasible expression trees. This gives a great reduction of computation time to search a desired expression tree. In *CanGeneratePermutation*, on the other hand, permutation operations are performed $n_{depth} \cdot |\mathrm{P}|$ times. It is reasonable since it is polynomial in both the number of permutation operations and the depth of the tree.

This permutation instruction generation algorithm generates a feasible expression tree with minimum number of permutation instruction from minimum depth of feasible expression trees. In general, the depth of expression tree of the best solution is not minimum, and common sub expression should be considered in the algorithm. Therefore, to find the best solution required much time. The proposed algorithm can find a good solution in reasonable time in this feature.

5.3 Experimental Results

In this section, the proposed method is evaluated.

5.3.1 Experimental setup

To confirm the effectiveness of SIMD and permutation instruction generation algorithm, the algorithm was implemented. Media embedded Processor, MeP [55] was used as the target processor. MeP is a configurable processor core. The base processor, MeP core, is a 32 bit RISC architecture and has no SIMD instructions. SIMD instruction capability can be added to MeP core by customizing configuration. There are several configuration options for MeP, such as embedding user designed logics, adding coprocessors, and so on. The configuration option used in these experiments was coprocessor option. A dual-issue coprocessor which has a 64 bit register file and supports 8-parallel byte, 4-parallel halfword, and 2-parallel word SIMD instructions , was added to MeP core. Fig. 5.9 shows the target architecture in this experiments. There are 5 components in the Fig. 5.9, MeP core, a coprocessor, a local memory, a global bus interface and a data memory access controller. The MeP core and the coprocessor share a local memory, and both MeP core and the coprocessor can directly access the local memory through 64 bit data bus. Data transfers among MeP core, the coprocessor and the local memory are available. The local memory consists of data cache, data RAM, instruction cache and instruction RAM. The MeP core and the coprocessor fetched instructions and processing data from the local memory. The data memory access controller manages data transfers between the local memory and external memories. The target processor communicates with other components through the global bus interface. The coprocessor has 2 SIMD pipeline data paths. Both data paths supports 2/4/8 way arithmetic operations such as addition, subtraction and multiplication. 6 permutation instructions were implemented in the coprocessor. These permutation instructions take data elements in the lower or upper part of register from 2 source registers, then, interleave and store them into 1 destination register. Fig.5.10 shows all permutation instructions of the coprocessor. The additional coprocessor works with MeP core in parallel. Therefore, the target processor behaves as a 3-way VLIW processor. If the bit width of the processing data is 8, the target processor can perform up to 9 operations simultaneously; The

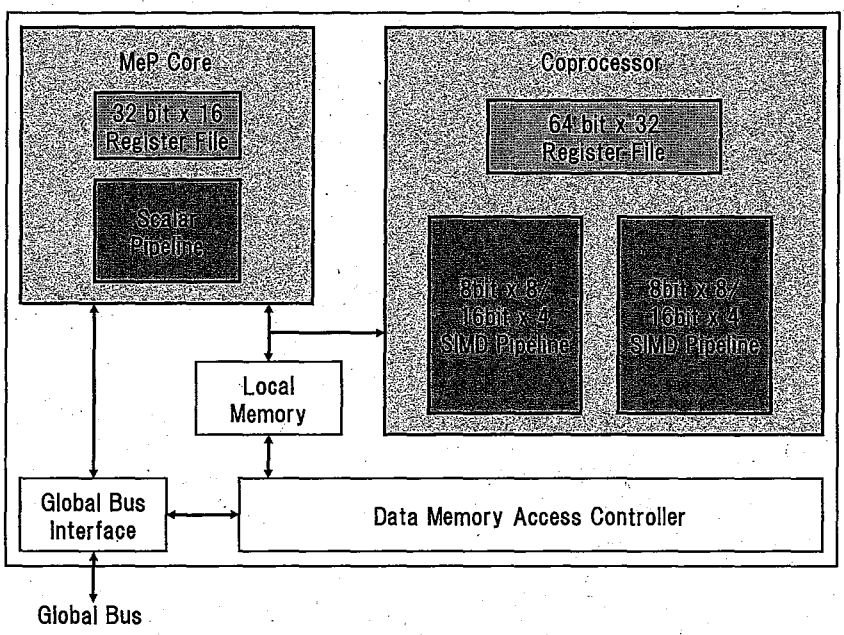


Figure 5.9: Target processor architecture

MeP core processes one operation, each SIMD pipeline in the coprocessor processes 4 operations. By the same token, if the bit width of the processing data is 16, the target processor can perform up to 17 operations.

In these experiments, 12 programs were used to evaluate the proposed method. To evaluate the ability to generate data reordering instruction sequence, following 4 programs which perform only data reordering were used:

- matrix transpose : matrix transposition
- bitreverse : bit reversed reordering
- reverse : reversed reordering
- shuffle : shuffle reordering

Other 8 programs were used to evaluate the entire SIMD instruction utilization technique. SIMD instructions could not be used to execute those programs without data reordering. Following 8 programs were selected as benchmarks:

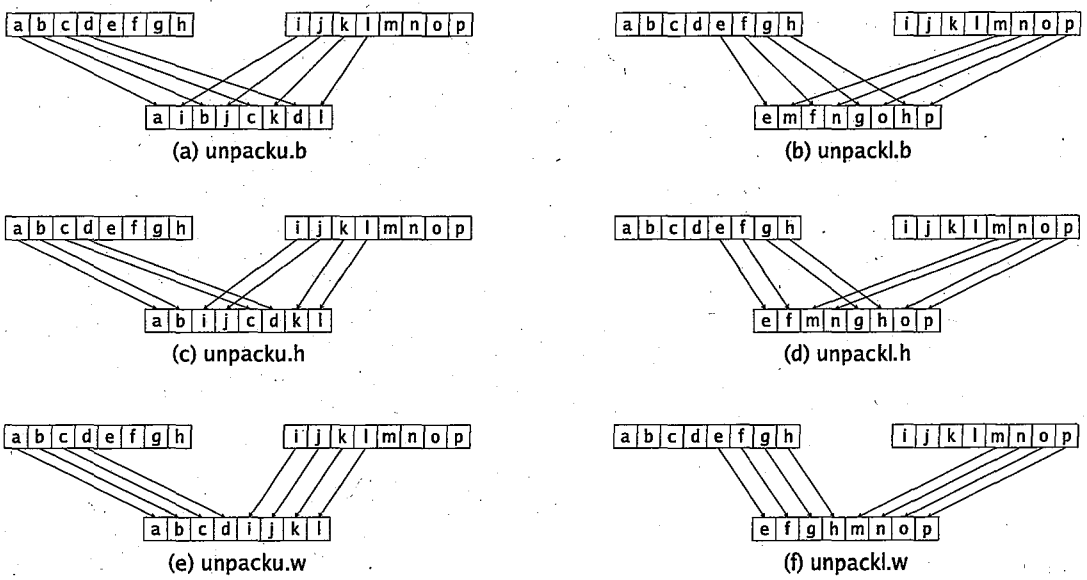


Figure 5.10: Permutation instructions of the target processor

- complex multiply : vector multiplication of complex numbers
- complex update : vector multiplication and addition of complex numbers
- convolution : 1 dimensional convolution
- dot product : inner product of two vectors
- matrix : multiplication of two matrices
- fft : Fast Fourier Transform of 16 complex numbers
- rgbgray : color conversion of an image from RGB to gray scale
- rgbcmyk : color conversion of an image from RGB to CMYK

5 of 8 programs, complex multiply, complex update, convolution, dot product and matrix are selected from DSPstone benchmark [47]. Other 3 programs, fft, rgbgray and rgbcmyk were coded from scratch.

These programs were suitable to confirm the ability to use SIMD and permutation instructions, because all programs includes operations which can be executed in parallel, and data reordering was necessary to process those operations by SIMD instructions. The size of a data

element was 16 bits and 8 bits for all programs. Therefore, 4 or 8 parallel SIMD instructions were used for 16 or 8 bits versions of programs.

The proposed method was implemented as a translator which translates plain C program to C program using built-in functions which are mapped to SIMD and permutation instructions. In our implementation, the compiler analyzed given C program, and unrolled loops in the given program. Then, proposed method was applied to the unrolled loop body. Finally, a C program with built-in functions of SIMD and permutation instructions were generated. For these experiments, an MDD package which performs several operations needed to realize the proposed method was used. MDDs were represented as shared ROMDD[56] in the MDD package, and any special technique to reduce the size of MDD such as edge negation was not used. To evaluate the proposed method, we also implemented the permutation instruction generation method based on backward tree and forward tree proposed in [1], and developed another translator with the method of [1] for comparison. The translator was the same as the translator with proposed method except for the permutation instruction generation method. In [1], not only generation method of permutation instruction sequences, but also method to generate SIMD instruction. However, we implemented only permutation instruction generation method of [1], because the main objective of these experiments is to evaluate method to generate permutation instruction sequences.

The SIMD instruction utilization methods were applied to application programs by the translators. Then, the output programs of translators were compiled by MeP C compiler, and simulated by a cycle accurate instruction set processor simulator (ISS). The compiler and ISS provided by Toshiba Corp. was used for compilation and simulation. These experiments were performed on RedHat Enterprise 3 operating system running on Intel Xeon 2.8 GHz processor with 2GB of memory.

In these experiments, the SIMD instruction generation methods were evaluated in terms of the number of instructions and execution cycles. The number of instructions in the main loops of programs was counted for each assembly code generated by compilers with and without SIMD instruction generation method. The number of execution cycles was measured by using ISS. This experiments assumed all processing data were located in the local memory, and the results of processing were also stored into the local memory. The number of execution cycles

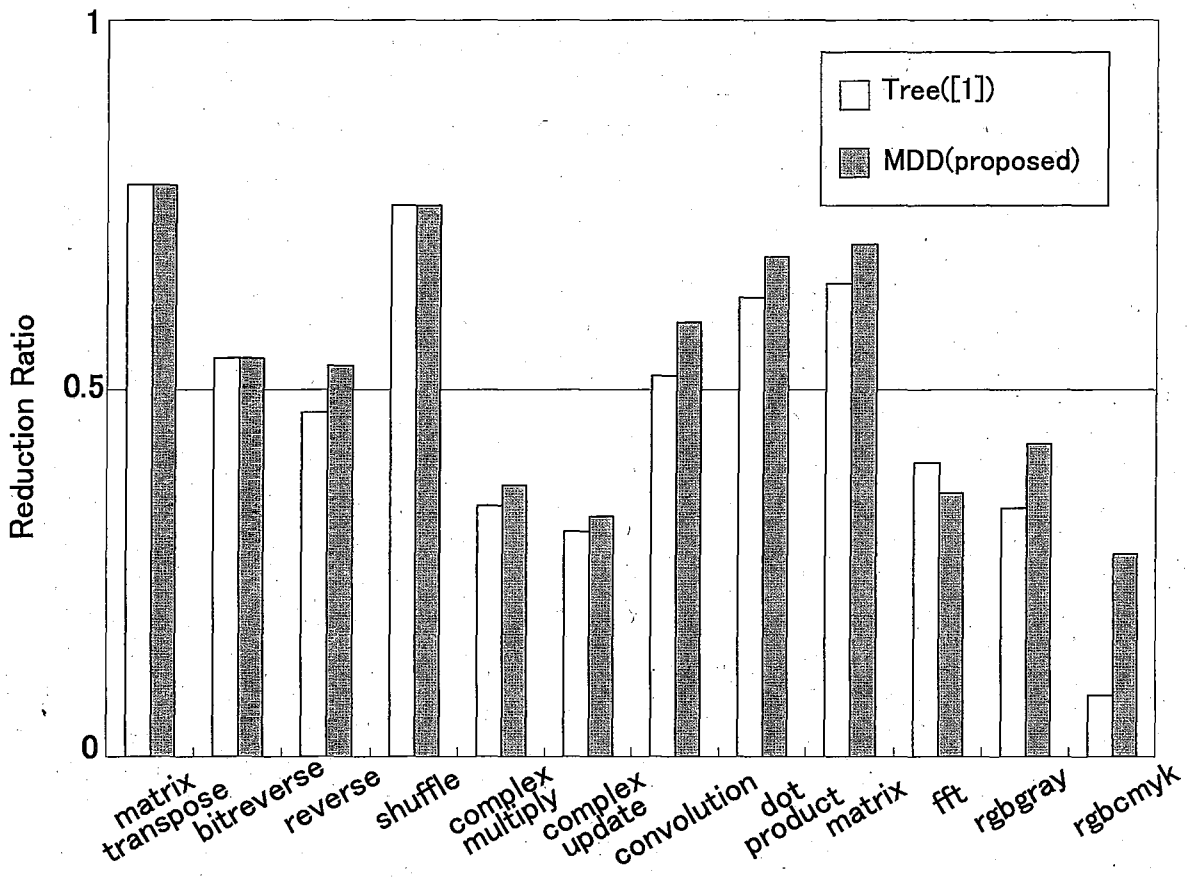


Figure 5.11: Code length reduction ratio.

The size of data element is 16 bits, 4 parallel SIMD instructions are used.

measured in these experiments was sum of the cycle count for loading data from the local memory, data processing and storing data into the local memory.

5.3.2 Results

Fig.5.11 and 5.12 show the reduction ratio in the number of instructions in the main loop of assembly code comparing the code generated by the compiler with the proposed method or the method of [1] to that without SIMD instruction. Comparing the proposed method to [1], the

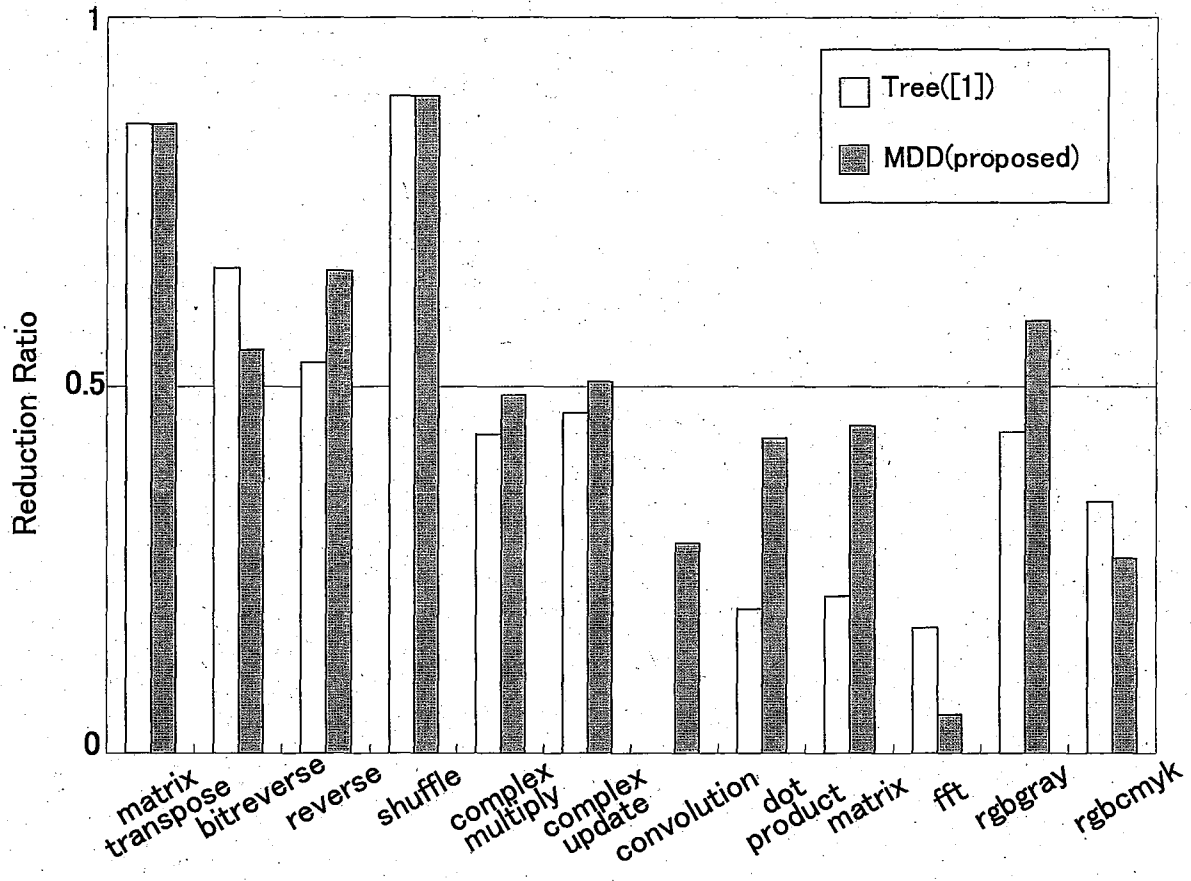


Figure 5.12: Code length reduction ratio.

The size of data element is 8 bits, 8 parallel SIMD instructions are used.

proposed method achieved the same or higher reduction ratio for most of the programs. Both methods successfully generated permutation instruction sequences for all programs. However, generated permutation instruction sequences for the same program were different for almost all programs. In case of data reordering programs (matrix transpose, bitreverse, reverse,shuffle) high reduction ratio was achieved by SIMD and permutation instructions. Without SIMD instructions, load and store instructions were generated for each data elements. However, with SIMD instructions, 4 or 8 data elements were loaded by one wide memory access instruction. Data elements were reordered by using permutation instructions, then reordered data elements were also stored by one wide memory access instruction. In case of other programs, not only memory access, but also data processing operations such as addition and multiplication were

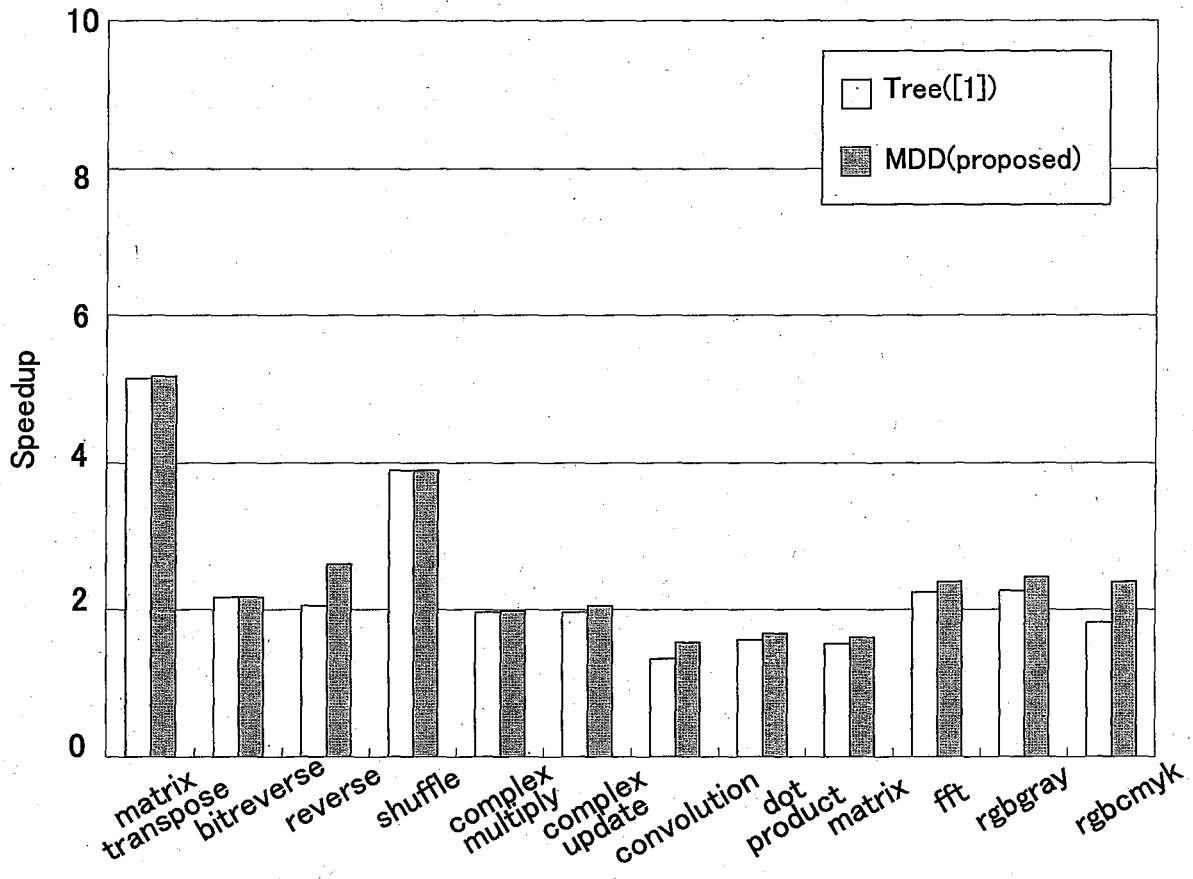


Figure 5.13: Speedup against without SIMD instructions.
The size of data element is 16 bits, 4 parallel SIMD instructions are used.

mapped to SIMD instructions.

Fig.5.13 and 5.14 show the speedup by the SIMD and permutation instruction utilization. Since the proposed method achieved higher code reduction ratio than the method of [1] as shown in Fig.5.11 and 5.12, the proposed method achieved higher speedup than [1] in most of the programs. In the case of fft of 8bits version, speedup of [1] was lower than the proposed method while the code reduction ratio of [1] was higher than the proposed method. This is because instruction level parallelism in the assembly code generated by the proposed method is higher than that of the method of [1]. The final assembly code generated by the proposed method became more efficient by instruction scheduling performed by the MeP C Compiler. In the case of 16 bits version of programs(Fig. 5.13), speedups were achieved from about 1.7 up

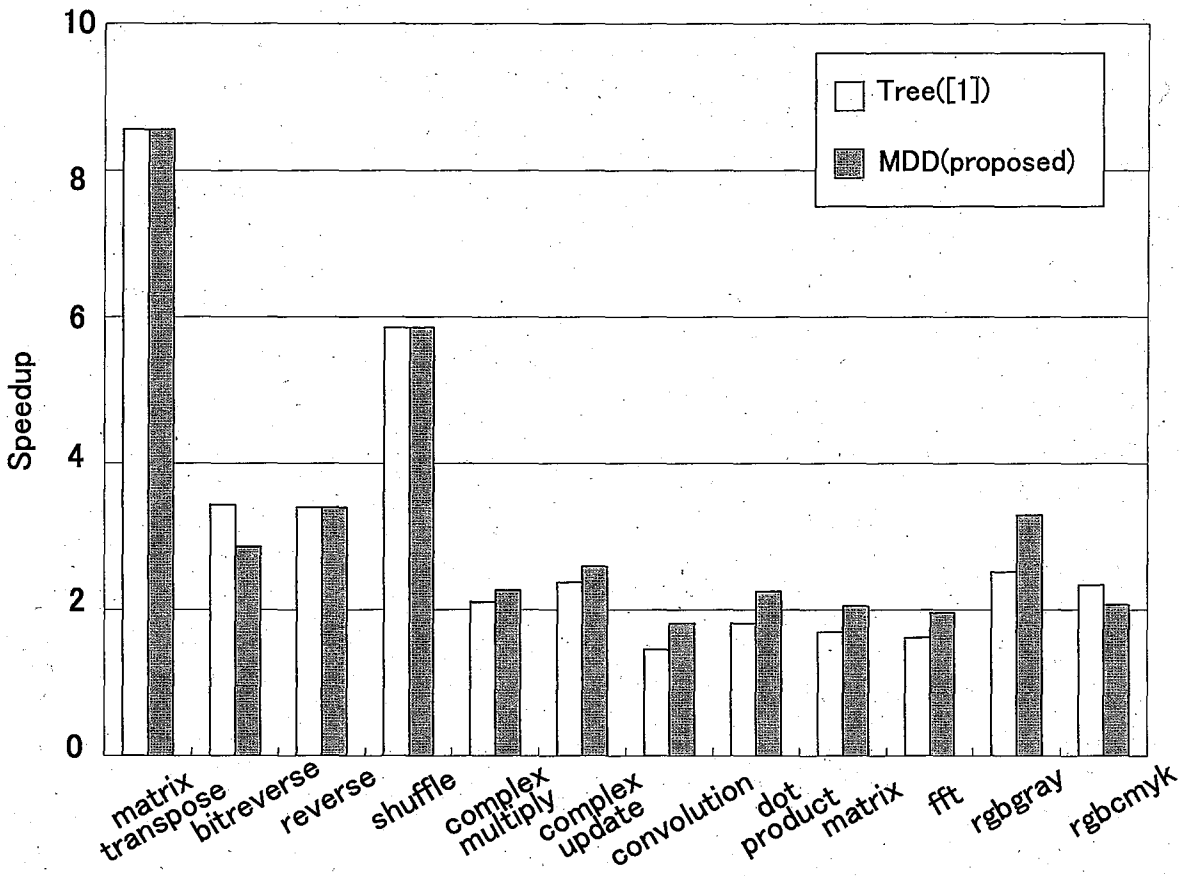


Figure 5.14: Speedup against without SIMD instructions.
The size of data element is 8 bits, 8 parallel SIMD instructions are used.

to about 5 times faster than without SIMD and permutation instructions. In the case of 8 bits version(Fig. 5.14), speedups were achieved from about 1.8 up to about 8.5 times faster than without SIMD and permutation instructions. The target processor behaves as a RISC processor when SIMD instructions are not used, but behaves as a 3-way VLIW processor when SIMD instructions are used. Therefore, the instruction level parallelism was also contributed to speedup. High speedup ratio exceeding SIMD parallelization factor was obtained for matrix transpose of 8 and 16 bits versions, as a result of both data level and instruction level parallelism. In the data reordering programs of 8 bits version, matrix transpose was about 2.0 times faster, and bitrerverse, reverse, shuffle were about 1.5 times faster than 16 bits version. In the case of 8 bits version of programs other than data reordering, 6 of 8 programs were about 1.25 times faster

than 16 bits version. However, in the case of fft and rgbcmyk, 8 bits version was slower than 16 bits version. This is because a large number of permutation instructions was generated as the number of data elements increase in the case of fft and rgbcmyk. The run-time overhead caused by permutation instructions greater than the execution cycles saved by the SIMD instructions. Table 5.1 shows the breakdown of generated instructions focused on the number of permutation instructions. "4 SIMD" and "8 SIMD" mean the size of a data elements is 16 bits and 8 bits respectively. The number of permutation instructions, total number of generated instructions and the percentage of permutations instructions for each program are shown in Table 5.1. In 4 SIMD, the ratio of permutation instructions was ranged from 22 up to 85 %. The number of generated permutaion instructions was small in the cases of shuffle, dot product and matrix. About a half of instructions was permutation instruction in the cases of matrix transpose, bitreverse, complex multiply and complex update. Highly ratio of permutation instructions could be found in the cases of reverse, fft, rgbgray and rgbcmyk. The reason why a large number of permutation instructions were generated was that the complex data permutations were required to perform those programs using SIMD instructions. For example, rgbgray takes an input vector composed of three colors, red, green and blue, and computes an output vector whose elements are the results of $output[i] = input[3 * i] + (input[3 * i + 1] >> 1) + (input[3 * i + 2] >> 2)$. In this case, 6 load instructions, 2 store instructions, 8 SIMD arithmetic instructions and 30 permutation instructions were generated. Fig. 5.15 shows the data flow of the generated code for this program. As shown in Fig. 5.15, permutation was very complex. Moreover, it took 5 permutation instructions for one permutation. Though the number of permutation instructions was large, total number of instructions was reduced, and writing such complex permutation instruction sequence by hand was too difficult for application programmers. In 8 SIMD, the ratio of permutation instructions was higher than 4 SIMD. The ratio of permutation instructions in some programs such as convolution, dot product, matrix and fft was much higher than the case of 4 SIMD. The reasons include the complex permutation,as well as unoptimized data permutation of the intermediate results. The proposed method determines the order of data in registers based on inputs and outputs of DFGs. This scheme often generates data permutation which require many permutation instructions. Better schemes or optimization techniques to determine the order of data in registers such as [57] would reduce the number of permutation

Table 5.1: Breakdown of generated instructions

	4 SIMD			8 SIMD		
	# of insn.		Ratio of	# of insn.		Ratio of
	# of perm.	Total	perm. [%]	# of perm.	Total	perm. [%]
matrix transpose	32	68	47	24	44	55
bitreverse	32	66	48	32	65	49
reverse	12	16	75	12	14	86
shuffle	4	14	29	2	6	33
complex multiply	52	96	54	48	78	62
complex update	64	129	50	58	95	71
convolution	9	23	39	35	40	88
dot product	4	18	22	23	32	72
matrix	4	17	24	23	31	74
fft	109	173	63	177	219	81
rgbgray	30	46	65	24	33	73
rgbcmyk	118	139	85	117	141	83

instructions.

In these experiments, the quality of the generated code cannot be clearly explained because the optimum solution cannot be obtained due to the high computational complexity of the optimum code generation for media instructions. However, there is potential for improvement of the performance of the target processor. The method to determine the order of data in registers have room for improvement as mentioned above. Much permutation instructions were generated by the current method as shown in Table 5.1. Permutation instructions would be reduced by improving this method. In addition, the quality of code would be improved by considering some architecture features such as multiple instruction issue and data path pipeline in code generation. Because the target processor in these experiments has one RISC core and 2 SIMD core, and supports 4 and 8 way SIMD instructions. Ideally, speedups could be achieved up to 9/17 times when using 4/8 way SIMD instructions. However, the speedups

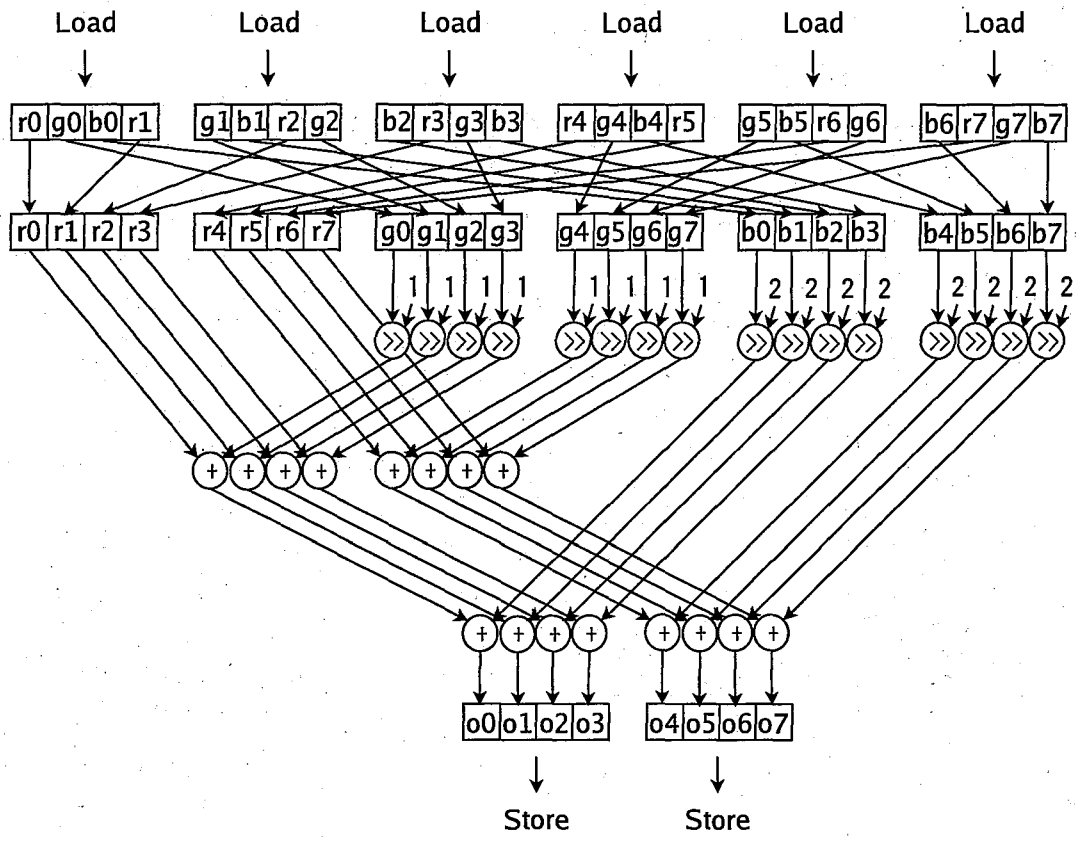


Figure 5.15: Permutation in rgbgray

shown in these experiments were averagely about 2 time. This was caused by not only much permutation instructions, but also unconsidered architecture features. Code generation with instruction scheduling considering VLIW and pipelined architecture would improve the quality of code. Because the generation of application specific instructions was the main topic in this study, other code generation and optimization techniques such as instruction scheduling and register allocation were out of the scope. Code generation with other code optimization techniques is future work.

Table 5.2 shows the compilation time of [1] and the proposed method for each program. "4 SIMD" and "8 SIMD" mean the size of a data elements is 16 bits and 8 bits respectively. In the 4 SIMD, the compilation time of [1] was shorter than the proposed method. Both methods compiled most of the programs within 10 seconds. fft and rgbcmyk took compilation time

Table 5.2: Comparison of compilation time between [1] and proposed method

	4 SIMD		8 SIMD	
	Tree([7]) [sec.]	MDD [sec.]	Tree([7]) [sec.]	MDD [sec.]
matrix transpose	7.47	8.51	2800	202
bitreverse	7.36	8.44	2700	17.7
reverse	0.65	0.89	409	37.3
shuffle	1.23	1.2	732	0.75
complex multiply	7.89	9.73	3380	271.3
complex update	11.9	14	4040	294
convolution	0.99	1.45	2550	40.0
dot product	0.74	1.04	2220	2.87
matrix	0.72	1.03	2220	2.77
fft	14.7	18	7410	1100
rgbgray	2.5	3.58	1060	290
rgbcmyk	24.3	34.3	8060	2320

more than 10 seconds, this is because the necessary data reordering was complex and appeared several time. On the contrary, in the 8 SIMD, the compilation time of the proposed method was shorter than that of [1]. [1] took more than 1000 seconds to compile each program for the most of the programs. On the other hand, it took less than 100 seconds to compile 6 programs for each program by the proposed method. In the case of fft and rgbcmyk, it took more than 1000 seconds. However, the compilation time was about a quarter of that of [1].

Table 5.3 shows the number of permutations(# perm) in $R_{i,j}$ and R_i generated in *CanGeneratePermutation()* of the proposed method, and the number of nodes of MDDs(# node) representing $R_{i,j}$ and R_i. The input sets of permutations was $R_0 = \{\text{abcdefgh}\}$. *CanGeneratePermutation()* was performed to generate as many permutations as possible. All possible permutations were generated after the main while-loop of *CanGeneratePermutation()* was performed 5 times. The numbers of permutations in $R_{i,j}$ and R_i become large as the value of i increases.

However, the most large size of MDD was R_3, and the size of MDD become small as the value of i increases. For all $R_{i,j}$ and R_i except for R_3, the number of MDD nodes was less than 1000 even if the number of permutations was more than 8000000. The sets of permutations were represented efficiently by MDDs.

5.4 Summary

In this chapter, a code generation technique for SIMD and permutation instructions are presented. Utilization of permutation instructions is essential for exploitation of SIMD instructions. In the presented algorithm, the packed data in registers are represented and manipulated by MDDs. Utilizing MDDs, permutation instructions can be generated efficiently. The experimental results show the permutation instruction generation algorithm can generate SIMD and permutation instructions, and reduce the number of instructions and speedup the execution of programs.

Table 5.3: Permutation count and MDD node count of sets of permutations generated by *CangeneratePermutation()*

$R_0 = \{$ `abcdefgh` $\}$, the number of permutation instructions, $|\mathbf{P}|$, is 6.

	# perm.	# nodes		# perm.	# nodes
$R_{1,0}$	1	10	$R_{4,0}$	8667135	86
$R_{1,1}$	1	10	$R_{4,1}$	8667135	79
$R_{1,2}$	1	10	$R_{4,2}$	8667135	24
$R_{1,3}$	1	10	$R_{4,3}$	8667135	86
$R_{1,4}$	1	10	$R_{4,4}$	8667135	79
$R_{1,5}$	1	10	$R_{4,5}$	8667135	24
R_1	7	33	R_1	14407168	454
$R_{2,0}$	36	69	$R_{5,0}$	16777216	1
$R_{2,1}$	36	57	$R_{5,1}$	16777216	1
$R_{2,2}$	36	24	$R_{5,2}$	16777216	1
$R_{2,3}$	36	69	$R_{5,3}$	16777216	1
$R_{2,4}$	36	57	$R_{5,4}$	16777216	1
$R_{2,5}$	36	24	$R_{5,5}$	16777216	1
R_2	188	205	R_5	16777216	1
$R_{3,0}$	7744	569			
$R_{3,1}$	7744	553			
$R_{3,2}$	7744	60			
$R_{3,3}$	7744	616			
$R_{3,4}$	7744	602			
$R_{3,5}$	7744	62			
R_3	34000	1795			

Chapter 6

Conclusion and Future Work

This chapter concludes this thesis. Then, the future direction of compilers for application specific instruction-set processors is discussed.

6.1 Conclusion

Compilation methods for any kinds of processors are crucial to ease software development effort. Compilers have to provide suitable programming models to describe applications and translate high-level programs into low-level assembly code. Emerging processors with new architecture features demand for new compilation methodology and optimization techniques.

Block-floating-point processors is specialized to perform arithmetic operations by block-floating-point manner. In spite of the advantage of block-floating-point arithmetic in performance and hardware area, block-floating-point arithmetic has been rarely employed in embedded systems because of the difficulty in programming. A challenge in code generation for block-floating-point processors is to bridge the gap between the programming model of block-floating-point and the model of general programming language.

A programming scheme and code generation method for block-floating-point processors is presented in this thesis. Compiler intrinsic functions are introduced to describe block-floating-point operations. Floating-point programs are easily translated into block-floating-point programs by using compiler intrinsic functions. Experimental results showed that the proposed compilation method successfully generates assembly code for block-floating-point processors.

The generated assembly code fulfilled the performance requirements for block-floating-point processors. The proposed method provides an easy way to use block-floating-point arithmetic to application developers.

On the other hand, the importance of media processors and its code optimization techniques have been increasing by the spread of media applications. A challenge in compilation for media processors is to exploit data level parallelism in application programs. Traditional parallelizing techniques for super computers based on vectorization of loops are not suitable to fully exploit the advantage of media instruction set. Utilization of SIMD instructions together with data permutation instruction maximizes performance of media processors. In this thesis, the code optimization problem for SIMD instructions considering permutation instructions is mathematically formulated into Integer Linear Programming problem. It is showed that the optimal assembly code is obtained by solving the formulated problem. Heuristic code generation for SIMD instructions is also proposed in this thesis. This method enables to generate assembly code with high degree of SIMD parallelism up to 8. The proposed method achieved speedup ratio up to about 8.5.Significant performance improvement in is shown by this method.

6.2 Future Work

The future work includes following items.

6.2.1 Automatic ASIP Design Space Exploration

Current ASIP design tools provide the interface to customize processors more easily than RTL design tools. However, the progress of semiconductor process technology is involving demands on higher productivity in electronic system design. ASIPs should not be manually customized but automatically customized in the next generation of ASIP design technology. Compiler technologies such as processor dependent and independent program transformation and optimization, retargettable code generation and optimization are indispensable to automate ASIP design.

6.2.2 Compilation Techniques for Low Power

As a concern to environmental issues including the warming of the earth grows, electronic systems are required to be ecological. To meet demands on low power for embedded systems, both hardware and software techniques to reduce power consumption are crucial. Compilation technologies for low power such as data localization in memory hierarchy, software controlled hardware activation, low energy usage of registers has been studied. There exists a lot of low power techniques by compilers, however, the relationships among different low power techniques or integration of several techniques have not been studied well. The mutual relation among some low power techniques is considered to be future work.

6.2.3 Compilation Techniques for Multi Processor SoC

Performance requirement for embedded systems is expected to be higher than current embedded systems. Current embedded systems have up to about 10 processor cores. However, more and more processor cores up to 100 or 1000 cores will be integrated in the next generation of embedded systems. Known compiler technologies for super computing which target a computing system consisting of up to 1000 or 10000 processing elements may be applicable to the next generation of multi processor embedded systems. However, the characteristics of the multi processor embedded systems are not well-known. Compiler technology for multi processor embedded systems will be a hot topic in the future compiler research.

Bibliography

[1] A. Kudriavtsev and P. Kogge, "Generation of Permutations for SIMD Processors," Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp.147 – 156, Jun. 2005.

[2] Texas Instruments, TMS320C1x User's Guide, 1991.

[3] Texas Instruments, TMS320C20x User's Guide, 1999.

[4] Texas Instruments, TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, 2007.

[5] Philips Semiconductors, PNX1300 Series Media Processors - Data Book, 2002.

[6] J.V. Praet, D. Lanneer, W. Geurts, and G. Goossens, "Processor Modeling and Code Selection for Retargetable Compilation," ACM Trans. on Design Automation of Electronic Systems, vol.6, no.3, pp.277–307, 2001.

[7] G. Coossens, D. Lanneer, W. Geurts, and J.V. Parat, "Design of ASIPs in Multi-Processor SoCs using the Chess/Checkers Retargetable Tool Suite," Proc. 2006 International Symposium on System-on-Chip, pp.1–4, 2006.

[8] CoWare, "Processor Designer." http://www.coware.com/products/processordesigner.php, 2007.

[9] A. Hoffmann, H. Meyr, and R. Leupers, Architecture Exploration for Embedded Processors with LISA, Kluwer Academic Publishers, 2003.

[10] R.E. Gonzalez, "Xtensa: a configurable and extensible processor," IEEE Micro, vol.20, no.2, pp.60–70, 2000.

[11] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid Configuration and Instruction Selection for an ASIP: A Case Study," DATE '03: Proc. of the Conference on Design, Automation and Test in Europe, pp.802–807, 2003.

[12] P. Marwedel and G. Goossens, Code Generation for Embedded Processors, Kluwer Academic Publishers, 1995.

[13] C. Liem, Retargetable Compilers for Embedded Core Processors, Kluwer Academic Publishers, 1997.

[14] R. Leupers, Retargetable Code Generation for Digital Signal Processors, Kluwer Academic Publishers, 1997.

[15] R. Leupers, Code Optimization Techniques for Embedded Processors, Kluwer Academic Publishers, 2000.

[16] T. Aamodt and P. Chow, "Embedded ISA Support for Enhanced Floating-Point to Fixed-Point ANSI C Compilation," 3rd International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), pp.128–137, November 2000.

[17] K.I. Kum, J. Kang, and W. Sung, "A floating-point to integer C converter with shift reduction for fixed-point digital signal processors," ICASSP '99: Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1999., pp.2163–2166, 1999.

[18] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," EURASIP J. Appl. Signal Process., vol.2006, no.1, pp.77–77, uary.

[19] R.J. Fisher, General-Purpose SIMD within a Register: Parallel Processing on Consumer Microprocessors, Ph.D. thesis, Purdue University, 2003.

[20] S. Kyo, S. Okazaki, and I. Kuroda, "An Extended C Language and a SIMD Compiler for Efficient Implementation of Image Filters on Media Extended Micro-Processor," Proc. of Acivs 2003(Adcanced Concepts for Intelligent Vision Systems)), pp.234–241, 2003.

[21] "Embedded C." http://www.embedded-c.org, 2007.

[22] J. Xiong, J. Johnson, R.W. Johnson, and D. Padua, "SPL: A Language and Compiler for DSP Algorithms," Programming Languages Design and Implementation (PLDI), pp.298–308, 2001.

[23] Texas Instruments, TMS320C64x DSP Library Programmer's Reference, 2003.

[24] "Intel®Math Kernel Library." http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, 2007.

[25] M.S. Lam, R. Sethi, J.D. Ullman, and A.V. Aho, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2006.

[26] K. Kennedy and J.R. Allen, Optimizing Compilers for Modern Architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc., 2002.

[27] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, "Instruction Selection using Binate Covering for Code Size Optimization," ICCAD '95: Proc. of the 1995 IEEE/ACM International Conference on Computer-Aided Design, pp.393–399, 1995.

[28] G. Araujo and S. Malik, "Code Generation for Fixed-point DSPs," ACM Trans. on Design Automation of Electronic Systtems, vol.3, no.2, pp.136–161, 1998.

[29] C.H. Gebotys, "An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy," ISSS '97: Proc. of the 10th International Symposium on System Synthesis, pp.41–47, 1997.

[30] R. Leupers and P. Marwedel, "Time-Constrained Code Compaction for DSPs," ISSS '95: Proc. of the 8th International Symposium on System Synthesis, pp.54–59, 1995.

[31] K. Ralev and P. Bauer, "Implementation Options for Block Floating Point Digital Filters," ICASSP '97: Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1997., pp.2197–2200, 1997.

[32] A. Mitra and M. Chakraborty, "The NLMS Algorithm in Block FLoating Point Format," IEEE Signal Processing Letters, pp.301–304, 2004.

[33] S. Kobayashi and G. Fettweis, "A Hierarchical Block-Floating-Point Arithmetic," Journal of VLSI Signal Processing, vol.24, no.1, pp.19–30, 2000.

[34] S. Kobayashi and G. Fettweis, "A New Approach for Block-floating-point Arithmetic," ICASSP '99: Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1999., pp.2009–2012, 1999.

[35] S. Kobayashi, I. Kozuka, and T. Kino, "Rapid Application Software Developement on A Block-Floating-Point DSP," Proc. 2003 International Signal Processing Conference, 2003.

[36] D. Elam and C. Lovescu, A Block Floating Point Implementationfor an N-Point FFT on the TMS320C55x DSP. Texas Instruments, 2003.

[37] A.J.C. Bik, M. Girkar, P.M. Grey, and X. Tian, "Automatic Intra-Register Vectorization for the Intel®Architecture," International Journal of Parallel Programming, vol.30, no.2, pp.65 – 98, Apr. 2002.

[38] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," Proc. of the Conference on Programming Language Design and Implementation, pp.145–156, Jun. 2000.

[39] A.E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp.82–93, Jun. 2004.

[40] P. Wu, A.E. Eichenberger, and A. Wang, "Efficient SIMD Code Generation for Runtime Alignment and Length Conversion," CGO '05: Proc. of the International Symposium on Code Generation and Optimization, pp.153–164, 2005.

[41] P. Wu, A.E. Eichenberger, A. Wang, and P. Zhao, "An Integrated Simdization Framework Using Virtual Vectors," ICS '05: Proc. of the 19th Annual International Conference on Supercomputing, pp.169–178, 2005.

[42] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of Interleaved Data for SIMD," PLDI '06: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.132–143, 2006.

[43] S. Larsen, R. Rabbah, and S. Amarasinghe, "Exploiting Vector Parallelism in Software Pipelined Loops," MICRO 38: Proc. of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pp.119–129, 2005.

[44] M. Imai, "ASIP Meister: A Configurable Processor Core Development System," Proc. ITI 3rd International Conference on Information & Communications Technology (ICICT), 2005.

[45] P.M. Sailer and D.R. Kaeli, The DLX Instruction Set Architecture Handbook, Morgan Kaufmann Publishers, Inc., 1996.

[46] ANSI/IEEE Standard 754, IEEE Standard for Binay Floating Point Arithmetic, 1985.

[47] V. Zivojnovic, J. Martinez, C. Schlger, and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology," Internatinal Conference on Signal Processing Applications and Technology, Oct. 1994.

[48] Tensilica, "Xtensa Processor Floating Point Unit." http:// www.tensilica.com/products/ x7_floa–ting_point.htm, 2007.

[49] GB3 Digital Systems, "ARM7 Floating-Point Co-Processor," 2005.

[50] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," ACM Trans. on Programming Languages and Systems, vol.11, no.4, pp.491 – 516, Oct. 1989.

[51] ACE, "CoSy compiler development system." http://www.ace.nl/compiler/cosy.html, 2007.

[52] S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai, "A Compiler Generation Method for HW/SW Codesign Based on Configurable Processors," IEICE Trans. on Fundamentals of Electronics, Communication and Computer Sciences, vol.E85-A, no.12, pp.2586–2595, 2002.

[53] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guid to the Theory of NP-Completness, W H Freeman & Co (Sd), 1979.

[54] S.M. Arvind Srinivasan Timothy Kam and R.K. Brayton, "Algorithms for Discrete Function Manipulation," Proceedings of the IEEE International Conference on Computer-Aided Design, pp.92–95, Nov. 1990.

[55] T. Miyamori, J. Tanabe, Y. Taniguchi, K. Furukawa, T. Kozakaya, H. Nakai, Y. Miyamoto, K. Maeda, and M. Matsui, "Development of Image Recognition Processor Based on Configurable Processor," Journal of Robotics and Mechatronics, vol.17, no.4, pp.437–446, 2005.

[56] D.M. Miller and R. Drechsler, "Implementing a Multiple-Valued Decision Diagram Package," Internatinal Symposium on Multi-Valued Logic, pp.52–57, May. 1998.

[57] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for simd devices," PLDI '06: Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, pp.118–131, ACM Press, 2006.