Compiler Generation Method from ADL for ASIP Integrated Development Environment

Yusuke Hyodo, Kensuke Murata, Takuji Hieda[†], Keishi Sakanushi, Yoshinori Takeuchi, Masaharu Imai

Graduate School of Information and Science, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
Email:y-hyodoh, k-murata, sakanusi, takeuchi, imai@ist.osaka-u.ac.jp

In this paper, we propose a compiler generation method from architecture description language (ADL) of ASIP Integrated Development Environment. By using our proposed method the modification of compiler, due to changes in processor specification, becomes easier and the amount of description and design time can be reduced. In our experiments, we compared the description and design time of an ASIP using our proposed method and conventional method, which generates a compiler manually. The experimental results show that the proposed method can reduce both the amount of the description and the design time by approximately 80% as compared to conventional method.

I. INTRODUCTION

In recent years performance requirement of embedded systems and the design complexity of these systems have increased. These two factors lead to higher development cost and increase time to market. Embedded system are usual implemented by using application specific integrated circuit (ASIC) or general purpose processor (GPP). When an embedded system is designed by using ASIC, power consumption is low and processing speed is fast, but adding or modifying functionality require a large amount of cost because the system must be redesigned from the beginning. On the other hand functionality of embedded system based on GPP can be easily modified at the cost of increased power consumption and reduced processing speed. In order to have the processing power and low power consumption of an ASIC while maintaining the flexibility of a GPP, application specific instruction-set processor(ASIP) are becoming popular.

Many factors, such as extended instruction and internal functional unit, influence the performance of an ASIP. Therefore, to develop an ASIP which satisfies design constraints and performance requirements, design and evaluation of ASIP are iterated alternatively. To evaluate ASIP performance, application program development environments such as compiler, debugger, and simulator are required.

[†]at present, Ritsumeikan University

Whenever ASIP is redesigned, application program development environments corresponding to ASIP has to be modified. Recently, the processor design environment which can generate HDL and application program development environment simultaneously is proposed[1]. ASIP Meister [1], which is an ASIP design environment, can develop application program development environment as well as ASIP HDL from original architecture description language(ADL). Generated tools include assemble, compiler, debugger and simulator. ASIP Meister also generates RTL description of an ASIP both Verilog and VHDL. When developing ASIP using ASIP Meister, a behavior description and a micro-operation description are should be defined. The behavior description defines the functional meaning of instructions and the micro-operation description defines operations for every pipeline stage of each instruction. The behavior description generates software development tools, such as compiler and simulator, and the micro-operation description generates ASIP's RTL. When there is conflict between behavior description and micro-operation description the application compiled by the generated compiler will not run on the generated ASIP.

To avoid this situation, it is required that the behavior description is unified with higher abstraction specification description which is an input of ASIP Meister. Consequently, a method of generating a micro-operation description from a behavior description and generating a synthesizable HDL description is proposed by T.Shiro et al. [2]. In addition, a method of generating a debugger from behavior the description is proposed by A.Watanabe et al. [3].

However information required for generation of software development tools lacks specification description of ASIP. As a result, the generation of the application program development environment does not have the flexibility for developing an application program.

A new behavior description which includes information required for generating an application program development environment is proposed by K.Murata [4].

In this paper we will present a compiler generation method from the behavior description proposed by K.Murata [4]. In the proposed method, information required for compiler generation is extracted from the behavior description of ASIP Meister, and converted into code generation description of a compiler. This method reduces the amount of descriptions for a compiler generation and a design period, and eases the design load of the designer caused by the contradiction during descriptions.

The composition of this paper is as follows. Section 2 explains ASIP Meister and CoSy compiler development system [5]. In Sect. 3, the method of generating a compiler from ADL which is an input of ASIP Meister is described. In Sect. 4 we will present the experiment results for evaluation of our proposed method, and finally in Sect. 5 conclude this paper.

II. ASIP DEVELOPMENT ENVIRONMENT

This section introduces ASIP Meister [1] and CoSy [5] compiler development system.

A. *ASIP integrated development environment: ASIP Meister*

ASIP Meister is an ASIP design environment using a behavior description. Based on the inputted specification description of the target processor, ASIP Meister generates synthesizable HDL description of the processor and generates simulator, compiler, and assembler corresponding to the generated processor.

1. Processor specification description

   The specification description of ASIP Meister proposed by K.Murata [4] consists of the following items.

   - Architecture information

     Name of instruction-set architecture, word size of instruction-set, number of bits of the standard data types of programming language
   - Data type information

     Arbitrarily set up data types which are not the standard data type of programming language
   - Register information

     Bit width of register of processor, name of register, number of registers, role of the defined register
   - Memory information

     Bit width of memory, name of memory, type of the stored information, accessible bit width
   - Flag information

     Name of flag, position on status register, the conditions which the flag becomes true
   - Processor instruction information

     Name of instruction, instruction argument operation, processing behavior of instruction, operation flag, assembly language format

B. *CoSy compiler development system*

In CoSy compiler development system, code generation description (CGD) is generated from the input information. And, the optimized compiler for the target processor is generated from CGD.

The information to CoSy compiler development system consists of the following items.

- Architecture information

  Architecture information describes the fundamental information of a processor, such as instruction bit width, size of C data type, and available memory types.
- Register information

  Register information describes the information including the number of register files, the attribute of an available register, etc.
- Processor instruction information

  Instruction information describes the information corresponding to the intermediate representation of a compiler.

III. GENERATION METHOD OF COMPILER CODE GENERATION DESCRIPTION

This section proposes the method of generating code generation description(CGD) of CoSy compiler development system from this specification description of ASIP Meister. The outline of the proposed method is shown in Fig. 1.

A. *The outline of an automatic compiler code generation description generation system*

In order to generate CGD of CoSy compiler development system, the extraction of the information needed for compiler generation from each information defined by ASIP Meister's specification description is required. Figure 2 shows the relationship between the descriptions inputted into the ASIP Meister's specification description and CoSy compiler development system. Arrows in Fig. 2 represent the relation between ADL information of ASIP Meister and CGD information of CoSy compiler development system. A compiler is automatically generated by translating the ADL information of ASIP Meister into CGD of CoSy compiler development system.

By the proposed method, the three information, architecture information, register information, and instruction

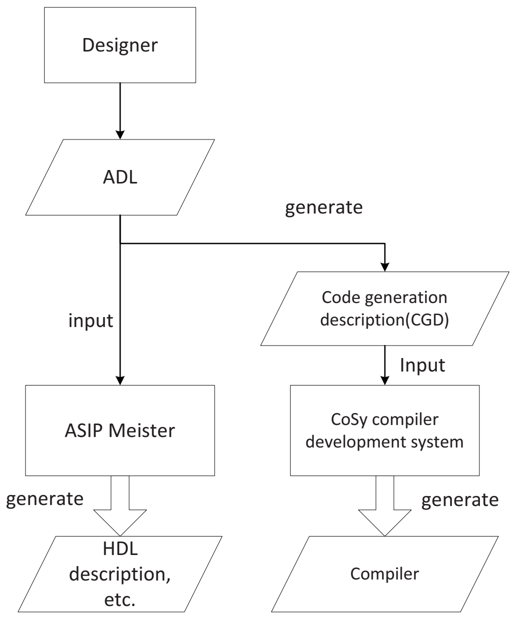

Fig. 1. The outline of the proposed method

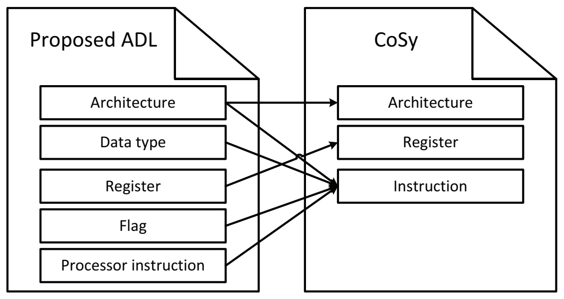

Fig. 2. The relation of the information on ASIP Meister and a CoSy compiler development system

information, which constitute description inputted into a CoSy compiler development system, are generated by considering the specification description of ASIP Meister as an input. Hereafter, a generation of architecture information, register information, and instruction information is described.

B. Generation of architecture information

The architecture information of CoSy compiler development system describes fundamental information on processors, such as bit width of instructions, size of C data types, and available memory types. All information required as the architecture information on CoSy compiler development system are included in the architecture information on ASIP Meister. Therefore, the required information is extracted from the architecture information, and generated and converted into input description of CoSy compiler development system.

C. Generation of register information

The register information on CoSy compiler development system describes register information, including the number of register files, the attribute of the register available, etc. . All information required as the register information on CoSy compiler development system is included in the register information on ASIP Meister, extracted required information from the register information, then generated and converted into input description of CoSy compiler development system.

D. Generation of instruction information

The instruction information on CoSy compiler development system describes information of an intermediate representation of compiler. Information required as the instruction information on CoSy compiler development system includes data type information, flag information and processor instruction information defined by ASIP Meister. Therefore, the information required from those information of ASIP Meister is extracted, and generated and converted into input description of CoSy compiler development system.

The instruction or the instruction group of a processor mapped to an intermediate representation is defined as an intermediate code. Depending on processor, the all instruction of the processor may not be mapped to an intermediate representation required for the compiler generation. For example, if a processor has neither multiplication instruction nor division instruction to map to intermediate representation, an intermediate representation for multiplication and division cannot be fulfilled.

Therefore, by the proposed method, the data of the specification description of ASIP Meister is converted to an intermediate code, and the intermediate code corresponding to intermediate representation which was not mapped is supplemented. Then, the intermediate code is generated and converted to an input description of CoSy compiler development system. The flow of the proposed method using an intermediate code is shown in Fig.3.

1. Generation of intermediate code

   From an instruction of the processor, the instruction matched an intermediate representation is generated as an intermediate code and stored in a list. The information which constitutes the generated intermediate code is shown as follows.

Identification number: ID given for every intermediate code is stored.

Format description: The description format of an intermediate code in the assembly is stored.

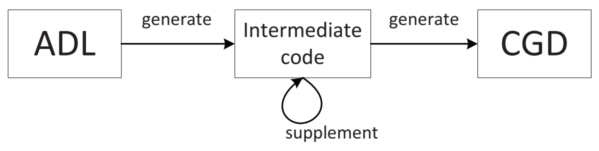

Fig. 3. The flow figure of the CGD generation from ADL using intermediate code

Argument information: The argument information of an intermediate code is stored.

Intermediate code list: The list of intermediate codes is stored.

Code quantity: The code quantity of an assembly is stored.

The number of execution cycles: The number of execution cycles of an intermediate code is stored.

The number of temporary registers: The number of the temporary registers used by an intermediate code list is stored.

2. Supplement of intermediate code

In generation of an intermediate code, it is necessary to generate the intermediate code corresponding to intermediate representation which was not mapped to an instruction or the intermediate code already generated by mapping of a processor.

Search of the intermediate code: Since the identification number is arbitrarily assigned to the intermediate code, it is required to check the intermediate code of the supplement based on the identification number. Then, the identification number which represents an intermediate code required for compiler generation is checked. When the intermediate code is not generated, it is necessary to supplement the intermediate code.

Supplement method: The method of supplementing an intermediate code is prepared beforehand inside a compiler generation system for every intermediate code, and then generates an intermediate code in accordance with the prepared supplement method.

For example, in a negate instruction, there are two supplement ways. The first one is to reverse a value and adding 1 to the reversed value. The second method is to subtract the value from 0. These two supplement ways is prepared beforehand. When supplementing a negate instruction, the method among these supplement ways which the number of execution cycles is smaller is chosen for supplement.

Supplement order: When an intermediate code is supplemented, the instruction using the supplement or the intermediate code may not exist, or the instruction has not been supplemented yet. In that case, the supplement of the intermediate code is interrupted, and it supplements the other intermediate code. Then, the supplement of intermediate code which has not been able to supplement occurs again. The above operation is iterated until supplement method is no longer performed. Generally, all intermediate codes required for compiler generation exist or can be supplemented. If intermediate codes occurred for compiler generation are insufficient, an error is raised because compiler generation cannot ne perform.

Subroutine-ized of intermediate code: The number of instructions and the number of execution cycles may become large depending on an intermediate code when it is supplemented. For example, in multiplication or division, the number of instructions and the number of execution cycles become large because a pseudo-multiply instruction and a pseudo-divide instruction are built by using an add instruction, a subtract instruction, a shift instruction, etc. for supplement.

Even if it directly supplements such instructions, the number of instructions and the number of execution cycles are still large. Therefore, readability and conservativeness are lost remarkably.

Then, the intermediate code which becomes large in the number of an instruction or the number of execution cycles from supplement method is subroutine-ized. In an intermediate code, a subroutine is called and an assembly file is directly outputted as a call place from an intermediate code. As a result, it becomes possible to maintain readability and conservativeness.

IV. EVALUATION EXPERIMENTS

The object of this experiment is to confirm that the proposed method that generates CGD from ADL can easily perform generation and specification convention of a compiler in a short time comparing to the conventional method that a designer describes both ADL and CGD. In this experiment, the compiler of target processor, Brownie Micro 16 [6], was designed by the proposed method and the conventional method. Then the amount of description, a design man day, and the ease of specification convention is compared.

TABLE I
COMPARISON OF THE AMOUNT OF THE DESCRIPTION (UNITS: BYTES)

	Amount of description	Reduction rates
Conventional method (ADL+CGD)	54,457	-
Proposed method (ADL)	12,440	77.30%

TABLE II
COMPARISON OF THE DESIGN TIME (UNITS: MINUTES)

	Design time	Reduction rates
Conventional method (ADL+CGD)	746	-
Proposed method (ADL)	125	83.24%

The comparison of the amount of description and the comparison of design time are shown in Table I and Table II. Accordingly, the proposed method can reduce both the amount of the description and the design time by approximately 80% to conventional method.

Moreover, in the case of adding five instructions in instruction-set and the case of adding 12 specific instructions, the amount of description and the design time are compared. The comparison results of the amount of description and the design time in adding instruction are shown in Table III, and Table IV. Furthermore, the comparison results of both method in adding specific instruction are shown in Table V and Table VI.

In Table III and Table IV, when instruction is added, the amount of a description is reduced by about 40%, and the design time is reduced by about 70%. In Table V and Table VI, when specific instruction is added, the amount of a description is reduced by about 50%, and the design time is reduced by about 70%.

Thus, it is confirmed that the proposed method can generate a compiler easier by a short time than the conventional method.

Additionally, a compiler is generated using the processor which has several instructions removed from BrownieMicro16 . As a result, it is confirmed that code generation description is generated correctly.

V. CONCLUSION

This paper proposed a method of compiler generation from ADL of ASIP Meister. The CGD generation system of CoSy compiler generation system from ASIP Meister's ADL is implemented. In the evaluation experiment, it is confirmed that the proposed method of generating a compiler from ADL can reduce both the amount of a description and the design time by approximately 80% in new design comparing to the conventional method that describes both ADL and CGD. Additionally, the amount of a description and the design time are compared when instruction is added in processor. Thus, it confirmed that the amount of a description is reduced by 40%, and the design time is reduced by 70%.

A future work is to supple optimization of processor instruction, the confirmation of correctness of a generation compiler, the extension of target processor architecture, and so on.

TABLE III
COMPARISON OF THE AMOUNT OF THE DESCRIPTION IN ADDING INSTRUCTION(UNITS: BYTES)

	Amount of description	Reduction rates
Conventional method (ADL+CGD)	2,702	-
Proposed method (ADL)	1,577	41.64%

TABLE IV
COMPARISON OF THE DESIGN TIME IN ADDING INSTRUCTION(UNITS: MINUTES)

	Design time	Reduction rates
Conventional method (ADL+CGD)	31	-
Proposed method (ADL)	10	67.74%

TABLE V
COMPARISON OF THE AMOUNT OF THE DESCRIPTION IN ADDING SPECIFIC INSTRUCTION(UNITS: BYTES)

	Amount of description	Reduction rates
Conventional method (ADL+CGD)	5,486	-
Proposed method (ADL)	2,912	46.92%

TABLE VI
COMPARISON OF THE DESIGN TIME IN ADDING SPECIFIC INSTRUCTION(UNITS: MINUTES)

	Design time	Reduction rates
Conventional method (ADL+CGD)	63	-
Proposed method (ADL)	18	71.43%

REFERENCES

[1] Masaharu IMAI, "ASIP Meister: A Configurable Processor Core Development System, " *Proceedings of ITI 3rd International Conference on Information & Communications Technology*, 2005.

[2] Takeshi SHIRO, Masaaki ABE, Keishi SAKANUSHI, Yoshinori TAKEUCHI, Masaharu IMAI, "Processor Generation Method from Instruction Behavior Description, " *DA Symposium 2006*, pp. 55-60, 2006.

[3] Aiko WATANABE, Takuji HIEDA, Keishi SAKANUSHI, Yoshinori TAKEUCHI, Masaharu IMAI, "Debug environ-

ment generation method for Software development in Hardware/Software Co-design, ” *DA Symposium 2007*, pp. 43-48, 2007.

[4] Kensuke MURATA, Takuji HIEDA, Keishi SAKANUSHI, Yoshinori TAKEUCHI, Masaharu IMAI, “Proposal of an Architecture Description Language for HDL Generation and Application Program Developing Environment, ” *DA Symposium 2011*, pp. 177-182, 2011.

[5] ACE Associated Compiler Experts, CoSy Compiler Development System: http://www.ace.nl/compiler/cosy.html.

[6] Hirofumi IWATO, Takuji HIEDA, Hiroaki TANAKA, Jun SATO, Keishi SAKANUSHI, Yoshinori TAKEUCHI, Masaharu IMAI, “A Highly Extensible Base Processor for Short-term ASIP Design, ” *IPSJ SIG Technical Reports*, Vol. 2007, No. 114, pp. 133-138, 2007.