

Title	Compiler Generation Method for ASIP Design Space Exploration
Author(s)	小林, 真輔
Citation	大阪大学, 2003, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/2287
rights	
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

Compiler Generation Method for ASIP Design Space Exploration

Doctoral Dissertation

by

Shinsuke Kobayashi Department of Informatics and Mathematical Science Graduate School of Engineering Science Osaka University

Contents

1	Introduction			1
	1.1	Applic	ation Specific Instruction-set Processor	2
		1.1.1	Benefits of ASIP	3
		1.1.2	Application Trends of Embedded Systems	4
		1.1.3	Problems of ASIP development	4
	1.2	2 ASIP Design Space Exploration Flow		
	1.3	Execut	ion Model of SoC with ASIP	6
		1.3.1	Interrupt Service Routine (ISR) Model	6
		1.3.2	Operating System (OS) Model	8
	1.4	Role of	f Compiler in ASIP	10
1.5 Compiler Retargetability		ler Retargetability	10	
	1.6	.6 Contribution of this Thesis		12
	1.7			12
2	Rela	Related Work		13
	2.1	Processor Generator		13
		2.1.1	Processor Core Generation based on Parameterized Generic	
			Processor Core	13
		2.1.2	Processor Core Generation based on Processor Specifica-	
			tion Language	15
		2.1.3	Comparison with Two Approaches	15
	2.2	Compiler Generator		
	2.3	Summary		

i

CONTENTS

3	Con	ıpiler G	eneration for ASIPs	21
	3.1	Introduction		
	3.2	PEAS-III		
		3.2.1	Organization of the PEAS-III system	22
		3.2.2	Flexible Hardware Model	23
		3.2.3	Micro-operation Level Processor Specification	24
	3.3	Proces	sor Model of PEAS-III	25
	3.4	Compi	iler Generation for PEAS-III	27
	3.5	Input Descriptions of the Compiler Generator		27
		3.5.1	Primitive operations used by resources	29
		3.5.2	Timing specifications of resources	29
		3.5.3	Storage units specifications for memory and register allo-	
			cation	30
		3.5.4	Instruction set specification including behavior of instruc-	
			tions and usage of resources	31
		3.5.5	Processor structure by resource connection graph	32
	3.6	Compi	iler Generation Flow	34
		3.6.1	Information Analysis	34
		3.6.2	Mapping Rule Generation	35
		3.6.3	Generation of Scheduling Information	42
	3.7	Summ	ary	42
1 Experiments		ts	45	
•	4 1 Experiment 1		ment 1	45
		4.1.1	Objective	45
		4.1.2	Target Processors	45
		4.1.3	Applications and Environment of the experiment	46
		4.1.4	Results	47
		4.1.5	Discussion	48
	4.2	Fyneriment 2		
		4.2.1	Objective	48
		4.2.2	Base Processor	48

ii

CONTENTS

5

	4.2.3	Applications and Architecture Candidates	49
	4.2.4	Results	49
	4.2.5	Discussion	52
4.3	Experi	ment 3	53
	4.3.1	Objective	53
	4.3.2	Target Application	53
	4.3.3	Target Processors	53
	4.3.4	Results	53
	4.3.5	Discussion	55
4.4	Case S	tudy	55
	4.4.1	Objective of Case Study	55
	4.4.2	Target Application: JPEG Codec	56
	4.4.3	Architecture Candidates	56
	4.4.4	Input Image	58
	4.4.5	DCT/IDCT Unit	59
	4.4.6	Additional Instructions	61
	4.4.7	Compiler Generation for Target Processors	67
	4.4.8	How to Estimate Design Quality	67
	4.4.9	Processor Organization	68
	4.4.10	Trade-offs Between Hardware Cost and Performance	69
	4.4.11	Trade-offs Between Hardware Cost and Power Consumption	72
	4.4.12	Design Time	74
	4.4.13	Discussion	74
4.5	Summ	ary	76
Disc	ussion		77
5.1	Compi	ler Retargetability	77
5.2	Code (Quality of the Generated Compiler	80
5.3	Requirements and Solutions for SoC Processors		
5.4	Design Productivity of SoC Processors		
5.5	Design	Space Exploration Using the Proposed Compiler Generator	82
	U		

CONTENTS

6	Con	clusion	and Future Work	83
	6.1 Conclusion			83
	6.2	Proture Work		
		6.2.1	Retargeting Algorithm for Special Architecture	84
		6.2.2	Simulator and Profiler	85
		6.2.3	VLIW extension	85
		6.2.4	Code Generation for Low Power Design	86
		6.2.5	OS Generation	86
Bi	bliogi	raphy		86
A	BNF	f of Arc	hitecture Description for the Proposed Compiler Genera-	
	tor			91
	A.1	Lexica	l Elements	91
	A.2	2 Grammer		
		A.2.1	Architecture Type Section	92
		A.2.2	Resource Class Declaration	92
		A.2.3	Structure Definition	93
		A.2.4	Storage Definition	94
		A.2.5	Instruction Definition	95
B	MIF	S-R300	0 Architecture Description for the Proposed Compiler Ger)-
	erat	or		99
Li	st of I	Major F	ublications of the Author	133

iv

Abstract

This thesis studies a compiler generation method for ASIPs (Application Specific Instruction-set Processor). In the ASIP development, it is an important issue that designers search for the architecture which matches target applications. This is called "design space exploration." In design space exploration, target processors are required to be evaluated in a short time. To evaluate architecture candidates, compiler plays an important role. When designers search for an optimal architecture of ASIP rapidly, the ASIP development system is one of the best solution.

PEAS-III (Practical Environment for ASIP development) [1] is an interactive ASIP design system. The PEAS-III system accepts the processor architecture description as input and generates a synthesizable HDL description of the target processor core, where user-defined instructions and interrupts can be easily implemented. The processor specification description includes: (1) architecture parameters such as pipeline stage counts and the number of delayed branch slots, (2) declaration of resources included in the processor such as ALUs and register files, (3) instruction format definitions, (4) micro-operation descriptions of instructions, and (5) interrupt definitions including cause conditions and micro-operation description of interrupts.

In this thesis, the compiler generation method for PEAS-III is proposed. The proposed compiler generation flow is as follows: (1) analysis of the target instruction set, and categorizing the instructions using the analysis result, (2) mapping rule generation for code emission, and (3) generation of scheduling information for code scheduling. In step (1), instructions are categorized into the following categories: (a) arithmetic, logical and compare operations such as addition, sub-traction and so on, (b) control instructions such as jump and branch, (c) load/store

instructions, (d) Compiler-Known-Functions for special instructions. In step (2), mapping rules for code emission are generated. Mapping rules produce relationships between internal representations of compiler and target instructions. In arithmetic, logical, and compare operations and their combinations, relationships between one instruction and one mapping rule can be made. However, in if-thenelse statements, function calls, and address calculation instructions, relationships between one instruction and one mapping rule cannot be made. In the proposed compiler generation method, the instruction for the case of multiple instructions to one mapping rule is automatically selected using instruction category. The control instructions and stack manipulation instructions can be selected using selection algorithm. In step (3), scheduling information is produced. When the instructions are scheduled, throughput and latency of the instruction are required. The proposed compiler generator calculates the throughput and the latency of the instruction group which uses the same resources when the instruction is executed.

Experimental results show that designers can efficiently evaluate numerous architecture candidates by means of execution cycles of applications, clock frequency, hardware cost of the processor core and power consumption when designers use the PEAS-III system. Therefore, designers can rapidly explore design space and explore trade-offs of designs by using the PEAS-III system. In addition, the JPEG Encoder case study shows that the proposed compiler generator improves the design time for the target compiler in a practical application.

Acknowledgments

I would like to express my gratitude to my adviser Prof. Masaharu Imai, Osaka University, for introducing me to this research area and guiding this work, for providing all facilities to carry it out, and for continuous support, help and encouragement.

The author also likes to express his thanks to Prof. Teruo Higashino, Prof. Hideo Matsuda and Prof. Yoshinori Takeuchi for reviewing this thesis, and to professors and staffs of the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University for their kind help.

I am extremely thankful to Prof. Jun Sato from Tsuruoka National College of Technology, Prof. Akira Kitajima from Osaka Electro-Communication University, Prof. Akichika Shiomi from Shizuoka University, and Mr. Nobuyuki Hikichi from Software Research Associates, Inc. Prof. Takumi Nakano from Toyota National College of Technology, Prof. Tsutomu Kimura from Toyota National College of Technology, Prof. Yoshimichi Honma from Nara National College of Technology, for their continuous support and encouragement, and many thanks to all members of the PEAS project for their kind assistance, especially, Dr. Makiko Itoh from Osaka University, currently she works for STARC (Semiconductor Technology Academic Research Center), Mr. Kentaro Mita from Osaka University, Dr. Keishi Sakanushi from Osaka University, and the members of the VLSI System Design Laboratory at Osaka University, especially, Ms. Akiko Mori, Ms. Ranko Morimoto, Mr. Norimasa Ohtsuki, Mr. Takafumi Morifuji, Mr. Jun-ichi Itoh, Mr. Yoshinori Jiyoudai, Mr. Katsuya Shinohara, Mr. Eiichiro Shigehara, Mr. Shigeaki Higaki, Mr. Shin'ichi Shibahara, Mr. Yoshiharu Watanabe, Mr. Tomohide Maeda, Mr. Naoki Morita, Mr. Yuichi Kurita, Mr. Teruaki

Sakata, Mr. Masaaki Abe, Mr. Toshiyuki Sasaki, Ms. Kyoko Ueda, Mr. Yukinori Yamane, Mr. Takuya Tokihisa, Mr. Koji Okuda, Mr. Youhei Ishimaru, Mr. Hiroaki Tanaka, Mr. Yoshio Okada, Mr. Yuki Kobayashi, and Mr. Noboru Yoneoka.

The author also thanks to professors and the members of Synthesis Corporation, especially, Prof. Isao Shirakawa from Osaka University, Dr. Toshiyuki Uegeki, Prof. Yukihiro Nakamura from Kyoto University, Prof. Koso Murakami from Osaka University, Prof. Kenji Taniguchi from Osaka University, Mr. Hideki Okamura, Mr. Toshihiro Yoshino, Prof. Takao Onoye from Osaka University, Prof. Toshihiro Masaki from Osaka University, Dr. Tomonori Izumi from Kyoto University, Dr. Hiroyuki Okuhata, Mr. Gen Fujita from Osaka University, Mr. Yukio Mitsuyama from Osaka University, and Mr. Masahide Hatanaka from Osaka University.

The author would like to thank professors and specialists for helpful discussions and encouragements, especially, Dr. Tokinori Kozawa from STARC, Prof. Toshiro Akino from Kinki University, Prof. Nagisa Ishiura from Kwansei Gakuin University, Dr. Hideki Yamauchi from Sanyo Electric Co. Ltd., Dr. Hiroyuki Tomiyama from Institute of Systems and Information Technologies / Kyushu, Dr. Morgan Hirosuke Miki from Sharp Corporation, Mr. Koji Miyanohana from Mitsubishi Electronic Co. Ltd., Mr. Takashi Okada from Hitachi, Ltd., and Mr. Tatsuo Watanabe from Sharp Corporation.

I would also like to express my thanks to all members of ACE Associated Compiler Expert bv., especially, Dr. Marnix Bindels, Dr. Bryan Olivier, Dr. Marcel Beemster, and members of Japan Novel Corporation, especially, Mr. Munemitsu Shioyama.

This work was partly supported by STARC, and one of tools was supported by Mentor Graphics higher education program.

Finally, I would like to thank my parents Shigeo and Sachie, and my brothers Naoki and Koji.

Chapter 1

Introduction

ITRS (International Technology Roadmap for Semiconductors) predicts that 90 % of SoCs (System-On-a-Chip) will include more than one instruction-set processor in 2005 [2, 3]. From these reports, instruction-set processors for embedded systems play an important role in the SoC design. Instruction-set processors have been developed and integrated by a lot of semiconductor companies, such as Intel Pentium processor, Motorola PowerPC, AMD Athron, and so on. These processors are used as CPUs (Central Processing Units) in personal computers. The primary requirement of CPUs for personal computers is high performance processing. Windows or Macintosh applications need to be executed on the processor faster and faster, people buy new PC that contains higher performance processor. Because the range of these applications faster. To execute every kind of applications, the hardware cost of the processor core and the development cost are very large.

On the other hand, consumer products such as set-top boxes, mobile terminals, entertainment machines and so on, also contain instruction-set processors. Requirements of embedded systems such as consumer products, are cost effective architecture and low power. Moreover, rapid technology change makes product life cycles short and makes time-to-market a critical issue for industries. Time required for design and verification is measured in months or years with high uncertainly. One of the solutions for this requirement is ASIP (Application Specific Instruction-set Processor) solution. In the ASIP design, designers consider the feature of application and select an instruction-set architecture. Because the architecture is suitable for application, ASIP can achieve not only low cost but also high performance and low power. Unfortunately, although the ASIP solution can achieve low cost, high performance and low power, development cost of ASIP is very large. The reason is that designers select an architecture from a lot of architecture candidates. Many designers decide such application specific processor architecture using their experiences. This approach, however, includes miss-decision, which means that they don't select suitable architecture for the target application. If the selected processor does not match the constraints of the target system, design time increases because they redesign architecture. Hence, evaluation of many architectures in a short time is a key issue for ASIP development. To achieve this task, the ASIP development environment that includes generation method both of processor and software development environment is needed. High abstraction level language reduces design and verification costs of ASIPs. Moreover, ITRS reported that software routinely accounted for 80 % of the embedded systems development cost. Hence, the software development environment plays an important role in the embedded system design, and compiler retarget technology, one of the key technologies of the software development environment generation, is indispensable. This chapter begins with a review and look at the trends of ASIP and retargetable compiler, and concludes with the organization of this thesis.

1.1 Application Specific Instruction-set Processor

ASIP (Application Specific Instruction-set Processor) is a programmable processor that is designed for a specific, well-defined class of applications. An ASIP is usually characterized by a small, well-defined instruction-set that is tuned to the critical inner loops of the application code. The following sections describe benefits of ASIP, application trends of embedded systems, and problems of ASIP development.



Cost-Performance

Figure 1.1: Advantage of ASIP solution

1.1.1 Benefits of ASIP

Figure 1.1 shows advantage of ASIP solution. The horizontal axis is cost-performance ratio and the vertical axis is flexibility. Off-the-shelf general purpose processor like Intel Pentium processor has high flexibility, but cost-performance ratio of the general purpose processor is low. On the contrary, although ASIC achieves high cost-performance, ASIC has lower flexibility. ASIP has higher flexibility than ASIC has, and achieves higher cost-performance than the general purpose processor. Hence, ASIP can be one of the key component of SoCs.

On the other hand, the cost of a SoC design is very expensive. Industry analysts indicate much of the rising cost of deep-submicron IC masks: The cost of a full mask set approaches \$1 million. As a result, it is difficult that designers change the SoC specification and redevelop chips. ASIP design methods permit painless workarounds for the design cost problem because ASIP has flexibility. Hence, flexibility is a key issue in developing SoC. Although ASIC cannot satisfy flexibility, ASIP can satisfy flexibility.

In addition, ASIP design methods increase designer productivity. RTL-based ASIC design routinely includes bugs because complexity of ASIC increases. An

ASIP based SoC design method significantly cuts risks of fatal logic bugs and permits graceful recovery when testers discover a bug. The reason is that designers develop software instead of hardware logic in complex function fields.

1.1.2 Application Trends of Embedded Systems

When the trends of ASIPs are examined, it is important to examine trends of the application requirements associated with embedded systems. The trends are as follows: (1) New wireless handsets and base stations need to support multiple mode. (2) The evolution of video coding standards are developing from JPEG, to MPEG1, MPEG2, MPEG4 and so on. Each standard evolution is accompanied by increase of significant complexity. As a result, many functions currently in hardware will be performed in software in order to accommodate this increased complexity and evolving standards. (3) Entertainment machines such as PlayStation 2, Game cube, Xbox and so on need high performance CG processing. Not only high quality graphics and presentation but also low price are required for entertainment applications.

1.1.3 Problems of ASIP development

However, there are still several problems in the ASIP development. First, designers must select an architecture from a lot of candidates when they develop ASIP, which is called "design space exploration." In addition, the SoC requirements allow much shorter time for time-to-market. Hence, designers do not have enough time to select an optimal architecture from a lot of designs. Secondly, development cost of hardware and software development environment is very large. Generally, the development cost of hardware and software development environment is several months or about a year. Therefore, reducing the development cost is a key issue in the ASIP design.



Figure 1.2: Design Space Exploration Flow.

1.2 ASIP Design Space Exploration Flow

ASIP design space exploration flow is shown in Fig. 1.2. The flow when designers search the design space of ASIP suitable for a target application is as follows: (1) Rapid Prototyping, (2) Evaluation, (3) RTL Design, (4) Logic Synthesis, and (5) Layout. In the first step, designers consider architecture and make prototype to evaluate the architecture. In the second step, architecture is evaluated using prototype made in previous step. To evaluate ASIP, software development environment such as compiler, simulator and assembler is needed. The reason is that the execution cycle when the target application is executed by ASIP is key factor to measure design quality. If the evaluation result does not fulfill the requirements of design constraints, designers return back to the previous step and consider another architecture candidates. If the evaluation result matches design constraints, designers write RTL model and proceed to the following design step. Of course, when fatal violation is occurred in final step, designers return back to the previous

step and redesign ASIP. To reduce iteration cost, prototyping and evaluation cost should be reduced. Generally, software environment development cost is on the order of several months and years. However, the development cost is too large to explore design space. Hence, software development environment, especially compiler, strongly needs to be developed rapidly.

1.3 Execution Model of SoC with ASIP

Execution models of embedded system with ASIP are categorized into two categories. One is interrupt service routine (ISR) model, and the other is operating system (OS) model. ISR model is used to realize multi-function or single-function system which execute a task at the same time. OS model is used to realize multifunction system which executes more than one task at the same time. The following sections explain execution models in detail.

1.3.1 Interrupt Service Routine (ISR) Model

In ISR model, function of system is designed using interrupt service routines. ISRs are located on memory map. Each ISR is executed when interrupt is occurred. Fig.1.3 shows an overview of ISR model. The system is started by reset interrupt. When reset interrupt is occurred, reset vector is executed and program jumps to boot routine. The boot routine processes stack allocation, global variable initialization, and so on. When the boot routine is finished, the program jumps to main routine. In main routine, variable initialization is executed. Then, the main routine waits interrupts. When an interrupt is occurred, the program jumps to interrupt vector and the program jumps to an ISR. ISR processes the function of system and return to the main routine.

The benefit of ISR model is simple organization. Hence, small embedded system applies ISR model. However, management of a lot of tasks using ISR model is difficult, because this model cannot manage task priority. If the target system needs real-time task management, OS model is a more suitable solution.



Program Memory

Figure 1.3: Overview of Interrupt Service Routine (ISR) Model.

1.3.2 Operating System (OS) Model

In OS model, tasks are managed using operating system. OS main routine executes tasks and switches contexts to avoid occupying resources of target system. Fig.1.4 shows overview of OS model. When reset interrupt is occurred, program jumps to boot routine. The boot routine executes stack allocation, global variables initialization, and so on. Then, the program jumps to program loader. The program loader loads the OS main routine, and locates to memory. When loader is finished, the program jumps to OS main routine. OS main routine executes tasks. When a task is switched, loader stores the context of task and loads new task data to memory. In each task, the priority of task and the sleep time of task can be set using system calls that depend on OS.

The benefit of OS model is that designers may not consider resource management and task management. Hence, development cost of application can be reduced, and portability of application in OS model is better than that in ISR model. However, designers must be familiar with OS and system calls to develop embedded system, especially real-time system.



Program Memory

Figure 1.4: Overview of Operating System (OS) Model.

1.4 Role of Compiler in ASIP

Previous section explains execution model when ASIP is used in SoC. In ISR model, applications of system are developed as ISR. In OS model, applications of system are developed as task. Designers develop ISR or task using high level language, such as C language, C++ language and so on, or assembly language. Using assembly language, designers can describe optimal applications for the target processor, but the development cost is too large to release products within a short design time. Although the code quality of application using high level language is not higher than the code quality of hand assembly code. However, portability of application using high level language is much better than that of hand assembly code.

Especially, in ASIP design space exploration, it is required that each task of systems is rapidly developed for target processors. Of course, application development time can be reduced when designers use compiler. In addition, when designers prepare hand assembly code for each processor, iteration cost is so large that total retargeting time for ASIPs is on the order months or years. As a result, compiler is very important for ASIP development, and compiler retargeting is a key issue to explore the best possible architecture.

1.5 Compiler Retargetability

For embedded processors, the interest in retargetable compilers is twofold:

- Retargetability allows the rapid set-up of a compiler to a newly designed processor. This can be an enormous boost for algorithm developers wishing to evaluate the efficiently of application code on different existing architectures.
- Retargetablility permits design space exploration. Processor designer is able to tune his architecture to run efficiently for a set of source applications in a particular domain, recompiling the application for each redesign of the architecture.

1.5. COMPILER RETARGETABILITY

From the interest, the retargetable compiler is needed by both algorithm developers and architecture designers. In today's retargetable compiler, several levels of retargetablility exists. In [4], they are generally categorized into three levels in compiler retargetability.

• Automatically retargetable level

The compiler includes a set of parameters that change the characteristics of target processors. Retargeting time is on the order of minutes and seconds, but compilers in this category mainly include parameterized compilers allowing a narrow range of target processor.

• Developer retargetable level

The compiler can be retargeted to a wide range of processor architectures, but this level compiler requires expertise with the compiler systems. Retargeting time is on the order of months and weeks. Therefore, the developer retargetable compiler does not satisfy the design time requirement when the compiler is used for design space exploration.

• User retargetable level

The designer is able to retarget the target processor even when changing its instruction-set specification. Retargeting time is on the order of days and hours.

Compilers in automatically retargetable level are mainly parameterized compilers which allow narrow variations of the target processor. The disadvantage in these compilers is the small range of targets which they support. Compilers in developer retargetable category supports a wide range of the target architectures. The disadvantage of these compilers is, however, large development time. As a result, the goal of the compiler generator for the ASIP development system should be user retargetable, because the user retarbetable level compiler widely permits the architecture design styles and set-up the application development environment rapidly.

1.6 Contribution of this Thesis

A compiler generation method for ASIPs is proposed in this thesis. The proposed compiler generator permits the design space exploration to find an optimal architecture from various range of architectures. The proposed compiler generator is the user-retargetable compiler generator, which uses both of the instruction-set information and the structural information. From this feature, designers can modify their design in a short design time, and the compiler generator keeps retarget range wider than that of the automatically retargetable compiler. The experimental results show that the modification cost of adding instructions to some processors and changing the resource features in it is so low that the developer can use this compiler generator in design space exploration.

1.7 Organization of this Thesis

The organization of the rest of this thesis is as follows: Chapter 2 describes surveys of ASIP development systems. Chapter 3 describes compiler generation method for the ASIP development system: PEAS-III, which has been developed in Osaka University. Chapter 4 describes experimental results using the proposed compiler generation method. Chapter 5 describes discussion of results presented in the previous chapters. Finally, Chapter 6 concludes this thesis and describes my future work.

Chapter 2

Related Work

In this chapter, ASIP development environments are surveyed. When designers develop processor core for embedded systems, they need to design processor core and software development environment suitable for the target application at the same time. ASIP development environments have been proposed to evaluate the processor organization and develop the processor core rapidly. The ASIP development environment includes generations of processor core descriptions and software development environment.

In the following sections, ASIP development environments that have been proposed is discussed.

2.1 Processor Generator

Conventional approaches to ASIP development can be classified into two kinds. One is based on "parameterized generic processor core," and the other is based on "processor specification language."

2.1.1 Processor Core Generation based on Parameterized Generic Processor Core

This category includes PEAS-I [5], Satsuki [6], MetaCore [7], CASTLE [8], and Xtensa [9].

PEAS-I is one of the system which utilizes ASIP optimization method. PEAS-I has the base processor called PEAS-I CPU. The PEAS-I CPU includes ALU, shifter, multiplier and divider. Users can specify the number of registers in the register file. The hardware algorithm of multiplier and divider are automatically selected using the result of target application profiling. Moreover, instructions are automatically reduced when the profiling result reports that the instructions are not needed. However, the pipeline stage cannot be changed.

Satsuki is similar to PEAS-I. In Satsuki, RISC processor, C compiler, assembler are generated from configuration file. In addition, the data and instruction width of RISC processor can be changed. Hence, designers can optimize CPU to reduce hardware cost and power consumption.

MetaCore is an application specific DSP development system. Basic and extended instruction set is prepared in MetaCore, and users can add custom instructions to the instruction set. The target architecture specification includes the netlist level description of the datapath structure and the behavioral description of instructions. From this specification, software development tools and HDL descriptions of the target processor are synthesized. However, execution units can be added to one pipeline stage, and changing the number of pipeline stages are not permitted.

In the CASTLE system, the target processor's datapath is described in block diagram. The CASTLE system generates VHDL descriptions of the processor that specifies the datapath. The feature of CASTLE includes: instantiation for functional units from a module library, automatic input signal conflict resolution by selector insertion, and generation of VLIW control word for the datapath. CASTLE, however, assumes a base VLIW architecture and cannot change pipeline stages.

Xtensa uses a customizable processor core. User-defined instructions described in Tensilica Instruction Extension Language (TIE) can be added to the base processor core. While Xtensa supports both processor generation and software development environment generation, user-defined instructions must be executed in restricted cycles. Designers can specify the behavior of new instructions and the structure of execution stage. However, the number of pipeline stages and

2.1. PROCESSOR GENERATOR

the structure of pipeline stages except for execution stage cannot be changed.

2.1.2 Processor Core Generation based on Processor Specification Language

In AIDL, designers specify operations of each pipeline stage, timing relations, and cause/effect relations among pipeline stages. Various kinds of processors including processors with out-of-order completion can be described in AIDL. However, it is difficult that designers modify the design because they have to consider various kinds of dependency in the inter-instruction behavior.

Hamabe, *et al.* proposed a description of clock based instruction behavior and pipeline stage information including the relationship between hardware units and the pipeline stage that contains their operations. Since designers must describe instruction behaviors considering pipeline registers, modification cost of this approach is larger than those of other approaches.

2.1.3 Comparison with Two Approaches

In this section, comparison with each approach is described. In the first approach, their processor models usually have basic instruction sets and a synthesizable ASIP description is generated by adding predefined or user defined instructions to the basic instruction set. Architectures of these processors ease to develop parameterized retargetable compiler, but in many cases have little flexibility on pipeline structure and instruction variations. Hence, the variety of architecture candidates by these systems is limited with respect to pipeline stage count, instruction format and micro-operation for each pipeline.

In the second approach, the variety of architecture can be described using specific languages. Therefore, a lot of architecture candidates can be designed and evaluated using this approach. However, generation of the target compiler from these languages is more difficult than that of the parameterized target processor model, because the range of the target architecture is too wide. Since the requirement of ASIPs includes wide range of architecture, the second approach is much superior to the first approach achieving the requirements of the SoC processor.

2.2 Compiler Generator

Several generation methods of software development tools for embedded systems have been proposed, and most of them utilize architecture description languages as their input. Architecture description languages are classified into three categories depending on the focus of processor specification: (1) the structure of the processor, (2) the instruction set of the processor, and (3) the structure and the instruction set of the processor.

2.2.0.1 Description Language Focusing on the Structure of Processor

In the first class, binding and scheduling tasks are executed using the structural information of the processor. Therefore, yielded compilers can generate high-quality codes for the target processor. The MIMOLA system[10] is an example of this approach. The MIMOLA system generates a set of application program development tools including a compiler, for a target architecture. The target processor is specified using the same MIMOLA language. The compiler generated by MIMOLA is called MSSQ, which is used to analyze the target application and to make a data graph called i-tree. However, because designers must specify interconnections among hardware resources using a selector, it is not easy to modify the target machine description.

2.2.0.2 Description Language Focusing on the Instruction-set of Processor

The second class includes nML[11] and ISDL[12], which are examples of the instruction set architecture description language approach. Because these methods focus on the instruction set, modification of the instruction set is easier than using the method focusing on the structure of the processor. CBC compilers can be generated from a compiler description in nML. However, it is not possible to specify multi-cycle or multi-word length instructions in nML.

The ISDL system also generates a compiler assembler and simulator. In ISDL, constraints on parallelism are specified through illegal operation groupings. Hence, complex architectures which permit using instruction set parallelism can be described in ISDL. However, these methods do not have the ability to specify pipeline

execution information. Therefore, the compiler cannot generate efficient object codes for pipeline processors.

2.2.0.3 Description Language Focusing on the Structure and the Instructionset of Processor

The last class includes LISA[13], FlexWare[14], HMDES[15] and EXPRESSION [16] whose languages focus on both the structure and the behavior of the processor. Because these languages consider both the structure and the behavior, the architecture information used in instruction scheduling, such as pipeline execution information, can be described in these languages. When ASIPs are designed using HW/SW co-design methodology, area, performance, and power consumption are required to be evaluated. To evaluate the design quality, synthesizable HDL models and target compilers are needed. However, hardware resource information cannot be described in these languages.

LISA has been developed for processor architecture design. LISA inherits concepts from nML. Moreover, pipeline execution information can be described in LISA language. While an assembler and a cycle-accurate simulator can be generated using LISA, no result is reported that indicates compiler generator in LISA so far.

FlexWare contains the CODESYN compiler and the Insulin simulator for ASIPs. The simulator uses the VHDL simulation model of a generic parameterized machine. User-defined instructions can be described by the combination of generic instructions. Designers can specify execution cycles for each instruction, but cannot specify pipeline organization. Moreover, resource conflict information considering with pipeline execution is not described in FlexWare.

HMDES language is developed by IMPACT project. HMDES language has a structural/behavioral representation. Information is broken down into sections based on a high-level classification. HMDES, however, allows restricted architecture types. Moreover, to modify the architecture, designers may change a lot of sections. It is not suitable for design space exploration that the modification cost is too large.

EXPRESSION has a mixed-level approach to facilitate design space explo-

ration. Moreover, EXPRESSION provides support for reservation tables by extracting them from the structural description. However, synthesizable hardware description cannot be generated by EXPRESSION.

The PEAS-III system uses structural and behavioral information to generate target compilers and synthesizable HDL models. When ASIPs are designed using the PEAS-III system, FHM [17] is used for resources of ASIPs, which has many parameters such as bit width, implementation algorithm and so on. These parameters of resources affect the throughput and latency of resources. The proposed compiler generator produces the target compiler rapidly, when designers change the parameters of resources. Using the PEAS-III system, designers can efficiently evaluate numerous architectural candidates in terms of programs, clock frequency, hardware cost and power consumption of the processor core.

2.3 Summary

In this chapter, ASIP development environments have been discussed. The ASIP development environment includes generation of both processor and software development environment, such as compiler generation, instruction-set simulator generation, and so on. In processor generation, two methods have been proposed. One is the method based on parameterized processor core, and the other is the method based on processor specification languages. In the method based on parameterized processor core is prepared and designers specify the parameters of the processor core and add special purpose instructions to the base processor. One of the features of this approach is that the target compiler and other software development environments can be produced easily. However, the class of the target processor is limited. In the method based on processor core are described using the language. This approach supports much wider architecture class than the former approach. To generate the processor core, however, the number of pipeline stages or execution cycles are limited in this approach.

On the other hand, it has been proposed that the software development environment for ASIPs is produced from architecture specification languages. These

2.3. SUMMARY

methods are classified into three categories. In the first approach, the target compiler and simulator are generated from the structure of the processor core that is described using RT-level description. This approach supports various type of the architectures like heterogeneous register files, non-orthogonal datapath, and so on. It is, however, difficult to modify the architecture because abstraction level of the description is low. In the second approach, the target compiler and simulator are produced from instruction behavior. In this approach, designers can modify the architecture easily because the abstraction level of the description is higher than RT-level description, but the class of the target architecture is limited rather than the first one. In the third approach, the target compiler and simulator are generated from the structure of the processor and the behavior of the instructions. This approach supports larger class than the second one. Moreover, the modification cost is smaller than that of the first one.

In next chapter, the proposed compiler generation method is explained in more detail. The compiler generator based on the proposed generation method is a sub system of the PEAS-III system, which is one of the ASIP development system. PEAS-III can generate synthesizable HDL description and software development environment such as assembler and compiler using the architecture specification language.

Chapter 3

Compiler Generation for ASIPs

3.1 Introduction

There are two approaches for realizing application domain specific embedded systems. One is to use general purpose processors and ASICs (Application Specific Integrated Circuits), and the other is to use ASIPs (Application Specific Instruction set Processors). One of the advantages of the second approach is that better implementations can be realized by introducing cost-effective instructions suitable for specific applications. In the ASIP design, it is also important to search for a processor architecture that matches the target application. To achieve this goal, it is essential to estimate the design quality of architecture candidates that have different instruction sets, pipeline stage counts, and combinations of hardware resources. Here, design quality indicates area, performance, and power consumption of a design. Because there are many architectural parameters, there exist a huge number of processor architecture candidates, which makes it difficult to find an optimal architecture in a short design time. In this case, the target compiler plays an important role in estimating the design quality of processor candidates.

PEAS-III (Practical Environment for ASIP development) [1] is an interactive ASIP design system. The PEAS-III system accepts the processor architecture description as input and generates a synthesizable HDL description of the target processor core, where user-defined instructions and interrupts can be easily implemented. The processor specification description includes: (1) architecture

parameters such as pipeline stage counts, the number of delayed branch slots, (2) declaration of resources included in the processor, such as ALUs and register files, (3) instruction format definitions, (4) micro-operation descriptions of instructions, and (5) interrupt definitions including cause conditions and micro-operation description of interrupts. While a processor architect can design a processor in a few days using PEAS-III, development of a compiler for a target processor took several months.

This thesis proposes a compiler generation method for the PEAS-III system. Experimental results show that various compilers and synthesizable HDL descriptions can be generated from the same architectural description and designers can analyze trade-offs among hardware cost, performance and power by using PEAS-III.

The rest of this chapter is organized as follows. Section 3.2 explains the PEAS-III system which is the ASIP development environment. Processor Model of PEAS-III is explained in section 3.3. Section 3.4 explains the proposed compiler generation method. Section 3.5 presents input descriptions of the compiler generator. In section 3.6, compiler generation flow is explained. Finally, section 3.7 summarizes this chapter.

3.2 PEAS-III

3.2.1 Organization of the PEAS-III system

The organization of the PEAS-III system is shown in Fig. 3.1. The architecture specification is written on the PEAS-III input system. The designer selects resources from Flexible Hardware Model. The design quality is estimated from the architecture parameter and the selected resources. The hardware description of the processor core is produced by the HDL-generator. The HDL-generator analyzes the micro-operation description and makes the data flow graph of the target processor. Then, the target HDL is generated using the data flow graph. The software development environment generator including the compiler generator also produces the compiler and the assembler description. The proposed compiler generator

3.2. PEAS-III



Figure 3.1: Organization of the PEAS-III system.

ator extracts instruction set information and structural information from the input system, and generates the mapping rules for the target compiler and scheduling information.

3.2.2 Flexible Hardware Model

Flexible Hardware Model is parameterized resource model. The parameter includes bit width, interface type, hardware algorithm and so on. The abstraction level of description, such as behavior level, RT level, and gate level, is also included in the parameter. When a designer would like to change the characteristics of the resource, he only changes the parameter of FHM. FHM has the functions that are used in micro-operation description explained below. For example, ALU has addition, subtraction, logical-and, and logical-or functions. These functions are defined in each class. Hence, when a designer changes the parameters of resources, he does not have to change the other part of descriptions.

3.2.3 Micro-operation Level Processor Specification

The micro-operation level processor specification consists of six major steps as follows: (1) Design Goal and Architecture Parameter Setting, (2) Resource Declarations, (3) Instruction Format Definition, (4) Interrupt Condition Definitions, (5) Interface Declarations, (6) Micro-operation Descriptions of instructions and interrupts. The following sections explain each part briefly.

3.2.3.1 Design Goal and Architecture Parameter Setting

In this step, the designer specifies the design goal of area, clock frequency, execution cycle count and power consumption. In addition, architecture parameters for pipelined execution are specified. The architecture parameters include the following items: the number of pipeline stages, the number of delayed branch slots,

3.2.3.2 Resource Declaration

In the resource declaration step, Flexible Hardware Models are selected from FHM-DB, and instance names and parameter values for them are specified when the designer declares the resource instance. Moreover, since the estimation result of each resource instance is displayed on the GUI, called FHM Browser, the designer can select the resource considering the area, the delay and the power consumption of resources.

3.2.3.3 Instruction Format Definition

In this step, the instruction type including bit fields, field type, field name is defined. The instruction format including ope-code binary representation is defined using the instruction type. In the micro-operation description phase, the bit field name can be referred when the designer specifies the storage.

3.3. PROCESSOR MODEL OF PEAS-III

3.2.3.4 Interrupt Condition Definitions

Interrupt definitions include the interrupt conditions and the number of execution cycles of each interrupt.

3.2.3.5 Interface Declaration

The interface declaration includes the entity name, the direction of interface, bit width, and the attribute. The attribute of the interface includes clock, reset, instruction_memory_address_bus, instruction_memory_data_bus, data_memory_address_bus, data_memory_data_bus, and user_defined_port.

3.2.3.6 Micro-operation Descriptions of Instructions and Interrupts

In the micro-operation description step, the designer defines the behavior of each pipeline stage and interrupt behavior. Operations of the processor such as setting specific values to the special registers and jumping to the interrupt handler routine are described in the interrupt definition. The micro-operation consists of the three kinds of statements: (1) Operations that are executed by resources, for example, arithmetic and logic operation, register read/write are included in this category, (2) Data transfers between resources, and (3) Conditional execution of operations and data transfers.

3.3 Processor Model of PEAS-III

The processor model of PEAS-III is explained in this section. Figure 3.2 shows the processor model of PEAS-III. The processor model consists of resources, controller, and pipeline registers. The number of pipeline stage can be changed. The designer can select resources in each pipeline stage. The controller and pipeline register is generated by HDL generator [1]. The HDL generator makes data flow of each instruction from micro-operation description. Each data flow is merged and selectors that arbitrate resource conflict are inserted by HDL generator.



Figure 3.2: The processor model of PEAS-III.

3.4 Compiler Generation for PEAS-III

Figure 3.3 shows the relationship between the proposed compiler generator and generated compiler. The instruction information and structural information, which are inputs of the compiler generator, are produced from the PEAS-III input system. The proposed compiler generator makes mapping rules, resource usage, and storage specification for the target compiler. The target compiler produced by the proposed compiler generator executes the following steps: (1) Parsing the source code, (2) Machine independent optimization, (3) Syntax tree rewriting and pattern matching, (4) Register allocation and Spill code insertion, (5) Instruction scheduling, (6) Machine dependent optimization, and (7) Assembly code output. In steps (1) and (2), the compiler generator does not touch their processing for each design because these steps are independent of the target processor. In step (3), syntax tree rewriting and pattern matching are executed using the mapping rules, which are rewriting rules of the target processor. For example, in Fig. 3.3, when the target processor has three rules: (a) $reg_i \le mem_1$ (Load reg_i), (b) $reg_i \le reg_i + 1$ (Inc reg_i), and (c) $mem_2 \le reg_i$ (Store reg_i), syntax tree $mem_2 \le mem_1 + 1$ are rewritten using Load reg_i , Inc reg_i , and Store reg_i . Steps (4), (5) and (6) are executed to reduce the code size and execution cycles, respectively. Finally, the assembly code is emitted in step (7).

The following sections explain the architecture descriptions which are used in the proposed compiler generator, and the flow of the proposed compiler generation.

3.5 Input Descriptions of the Compiler Generator

The description used in the compiler generator includes the following information: (1) primitive operations used by resources, (2) timing specifications of resources, (3) storage-unit specifications for memory and register allocation, (4) instruction set specification including behavior of instructions and usage of resources, and (5) the processor structure by resource connection graph. The rest of this section describes these description in detail.


Figure 3.3: Relationship between the Proposed Compiler Generator and Generated Compiler.

```
addition {
     interface {
            input {
                       { type { int } width { 32 } }
                  а
                       { type { int } width { 32 } }
                  b
            }
            output {
                       { type {int} width {32} }
                  с
            }
      }
      behavior {
           c = a + b;
      }
}
```

Figure 3.4: Example of Primitive Operation Used By Resources.

3.5.1 Primitive operations used by resources

Resources contain particular primitive operations, which represent the behavior of resources. The primitive operations are described using sentences. The primitive operations are used in the timing specification of resources.

An example of primitive operation used by resources is shown in Fig. 3.4. The function "addition" has two input ports and one output port. The input and output port data type are 32 bits integers. The operation "+" is one of the primitive operations.

3.5.2 Timing specifications of resources

The timing specification of resources includes throughput and latency information when functions of resources are used. The throughput and the latency are used for instruction scheduling. This information can be acquired from FHM-DBMS[18]. Hence, when the designer changes the resource parameters including implementation algorithms, specifications for resources are generated from FHM-DBMS.

```
ADDER {
      port {
           input \{ in1, in2 \}
           output { out1 }
      }
      function {
           addition {
                 interface {
                       in1 \{a\}
                       in2 {b}
                       out1 \{c\}
                  }
                  latency {1}
                  throughput {1}
            }
      }
}
```

Figure 3.5: Example of Timing Specification.

An example of timing specifications of resources is shown in Fig. 3.5. ADDER has the addition function. The latency and the throughput of the addition of ADDER are 1 cycle and 1 cycle, respectively. The latency and throughput sections are used when calculating the throughput and latency of instruction.

3.5.3 Storage units specifications for memory and register allocation

The specification for a storage unit consists of available flag, storage class, resource, size of storages, bit width, and data type. The specification for storage unit is used for memory and register allocation in the generated compiler. The available flag indicates that the storage can be allocated by the compiler. The storage class indicates the usage of storage such as data register, program counter, data memory, instruction memory, stack pointer and frame pointer. The resource

```
GPR {
    class { reg }
    resource { GPR }
    avail { T }
    number { 32 }
    width { 32 }
    data_type { any }
}
```

Figure 3.6: Example of Storage Unit Specification.

indicates hardware resource. The number indicates the number of storage. The width means data width. The data type means what kinds data type can be treated.

An example of storage unit description is shown in Fig. 3.6. The "GPR" belongs to the register class, and uses resource "GPR". Moreover, the available flag field is T, which means that the storage GPR can be allocated by the compiler. The number of storages which belong to GPR is 32, and bit width is 32. The GPR treats any data type. This means that the GPR has 32 general purpose registers, and each register has 32 bits.

3.5.4 Instruction set specification including behavior of instructions and usage of resources

The specifications for an instruction set include operand declaration, instruction format, usage of resources, and behavior of instruction. Operands of instruction are declared in operand field. Operands are declared using addressing modes. Table 3.1 shows addressing modes. First column shows addressing mode, and second column shows description of addressing mode. The "REG" is the storage instance which belongs to register class, and the "MEM" is the storage instance which belongs to memory class. Format of instruction is declared in format field. The format of instruction is used to make assembler file format. Resources and functions which are used by the instruction are described in functions field. Usage

CHAPTER 3. COMPILER GENERATION FOR ASIPS

Addressing Mode	Description
Register direct	REG
Register indirect	[REG,disp]
Memory direct	@MEM
Memory indirect	@[MEM,disp]
Immediate	#Imm

Table 3.1: Addressing Mode.

of resources is used in generation of the scheduling information to avoid resource conflict. Behavior of instruction is used for instruction mapping. Behavior is represented using combinations of operators included in "C" language such as "+", "-", "*" and so on.

An example of instruction description is shown in Fig. 3.7. The instruction "ST" has two operands. The operand "a" uses register-direct addressing mode using GPR register, and the operand "b" uses register-indirect addressing mode. The data type of both operands are INT32to0. This data type is user-defined data type. The function field describes resource and function usage of each pipeline stage. The behavior of instructions is described in the behavior field. The behavior of "ST" instruction includes data write and address increment.

3.5.5 Processor structure by resource connection graph

The structure of the processor is represented by a resource connection graph. Nodes in the resource connection graph correspond to the components in the processor, and the edges in the graph correspond to the resource connections. The processor structure is created from a micro-operation description [1]. Since a resource connection graph is generated, designers can concentrate on the instruction design.

An example of a processor structure description is shown in Fig. 3.8. The resource "ADDER0" belongs to the resource class ADDER. The ADDER0 is in the third pipeline stage, and connects to GPR.

```
ST {
     operand {
          GPR
                                     INT32to0
                                                   a;
          [GPR, disp]:DMEM
                                    INT32to0
                                                   b;
     }
     format {
          "ST" a ", " b
     }
     functions {
          stage(1) {
               PC.read
               IMEM.load_word
               PC.inc
               IR.read
          }
          stage(2) {
               GPR.read0(a)
               GPR.read1
          }
          stage(3) {
               ALU0.addition
          }
          stage(4) {
               DMEM.store(b)
          }
          stage(5) {
          }
     }
     behavior {
          *b = a;
          b = b + 4;
     }
}
```

Figure 3.7: Example of Instruction Set Description.

```
ADDER0 {
    class { ADDER }
    stage { 3 }
    connection {
        out1 {
            GPR.in4
        }
    }
}
```

Figure 3.8: Example of Processor Structure Description.

3.6 Compiler Generation Flow

In this section, compiler generation flow is explained. The generation flow is as follows: (1) analysis of the target instruction-set, and categorizing the instructions using the analysis result, (2) mapping rule generation for code emission, and (3) scheduling information generation for code scheduling. The following section explain each step.

3.6.1 Information Analysis

The proposed compiler generator analyzes the instruction set. The target instruction set must include the minimum set of instructions which can compile any source code in C language. This result is used in the step of mapping rule generation. The proposed compiler generator examines the following cases: (1) All operations, which can be written in C language, are included in the target instruction set, (2) Load and store instructions are included in the target instruction set, and (3) Control instructions are included in the target instruction.

3.6. COMPILER GENERATION FLOW

3.6.2 Mapping Rule Generation

Mapping rules are created by the proposed compiler generator. The proposed compiler generator classifies target instructions into several categories. From these categories, mapping rules are generated. Instructions can be classified into the following categories: (1) arithmetic and logical instructions, (2) control instructions, (3) load and store instructions, (4) stack manipulation instructions, and (5) special instructions. The rest of this section explains these categories.

(1) Arithmetic, Logical and Compare Instructions

The instructions whose behavior is written by using arithmetic, logical and compare operations, are categorized into arithmetic, logical and compare instructions, respectively. Moreover, the compare instructions are categorized into two categories. One involves instructions writing the result of comparison to register, and the other involves instructions writing the result of comparison to condition code. When the result of the compare instruction is written to register, the behavior of compare is that of relational operations, such as "less than," "greater than," and so on. When the condition code is issued, the behavior of comparison is subtraction and updating of the condition flags including zero flag, carry flag, negate flag, and overflow flag. The proposed compiler generator analyzes these instructions and generates the mapping rules.

(2) Control Instructions

The instructions, which have the effect of changing the value of the program counter, are categorized into control instructions. Control instructions include conditional branch, jump, and function call. The conditional branch is described using an "if" statement with condition. The jump instruction is described using an "if" statement without condition. The function call is described using an "if" statement without condition. The function call is described using an "if" statement and assignment of the value of the program counter to the link register or the stack. The proposed compiler generator categorizes and maps these instructions to the syntax tree of the compiler.

Condition code	Relational operations
Z == 1	==
Z == 0	! =
N == 0	>=
N == 1	<
N == 0 && Z ==	>
0	
N == 1 Z == 1	<=

Table 3.2: The assignment rules between condition code and relational operations.

Moreover, the proposed compiler generator checks the condition of "if" statement. When the relational operations are used in the condition, the compiler generator assigns conditional branch instructions to syntax tree using each relational operation. When the conditional code is used in the condition of branch, the proposed compiler generator assigns branch instructions to the syntax tree using the rules, which are explained in table 3.2. Table 3.2 shows the assignment rules from condition code to relational operations. In table 3.2, 'Z' denotes zero flag, 'N' denotes negate flag. '1' denotes true value and '0' denotes false value.

(3) Load and Store Instructions

Load and store instructions are instructions whose behaviors include data transfers from memory to register and vise versa. The proposed compiler generator checks the data type and storages including register files and memories when the mapping rule of load/store is generated. The algorithm of mapping rule generation is summarized as follows.

- 1. Load instructions that move data from memory to register are obtained from a target instruction set.
- 2. Store instructions that move data from register to memory are obtained from a target instruction set.

3.6. COMPILER GENERATION FLOW

3. Conditions that are used in rule selection for syntax tree rewriting are made from manipulating data type and storages.

(4) Stack Manipulation Instructions

In the compiler, memory space of parameter and local variable are accessed using stack pointer and frame pointer. When function calls are executed, parameter values are pushed to stack. The proposed compiler generator selects such stack manipulation instructions. The selection algorithm is as follows.

- 1. Load and store instructions are obtained from a target instruction set.
- 2. Instructions that can use the stack pointer and frame pointer are selected from load and store instructions.
- 3. Data width is checked and obtained from instructions which have been selected in the previous step.

These stack manipulation instructions are used as spill and reload instructions.

Fig.3.9 shows memory layout of stack frame. The stack frame consists of function parameters, return address, frame pointer, and local variables area. The stack manipulation instructions are used for storing data, and loading data when function is called. The function call instruction selected from control instructions includes the return address assignment to link register. The value of link register is stored to stack frame after the function has been called. Address calculation instructions for local variable area allocation is selected from addition instructions or subtraction instructions.

(5) Special Instructions

Special instructions such as complex multiply and accumulate, trap instructions and co-processor control instructions determine the characteristics of the processor. In the proposed compiler generator, special instructions are represented using Compiler-Known-Functions. These functions directly replace instructions instead of constructing a usual function call. The proposed compiler generator checks



Figure 3.9: Memory Layout of Stack Frame.

3.6. COMPILER GENERATION FLOW

```
ckf prototype {
     void complexMAC ( unsigned int , unsigned int );
}
```

```
CKF_complexMAC {
	operand {
		GPR UInt31to0 a;
		GPR UInt31to0 b;
	}
	... (snip)
		behavior {
		complexMAC ( a , b );
	}
}
```

Figure 3.10: Example of Special Instruction Definition using Compiler-Known-Function.

the data type and storages used in the Compiler-Known-Functions, and annotates them to the target compiler.

Figure 3.10 shows an example of special instruction definition in the proposed compiler generator. In the **ckf prototype** section, the user defines Compiler-Known-Functions to execute special instructions. In instruction definition, the user describes the behavior using Compiler-Known-Functions, which he defines in the ckf prototype section. The proposed compiler generator produces the rules, which are specified to emit special instructions when Compiler-Known-Functions are used in input C source code.

(1) Method specifying loop end ad-	(2) Method specifying the number
dress	of loop instructions
Loop co	unter size
Start address size	Start address size
(End address size)	The number of Instruction
	Instruction Buffer
The number of special reg	ister sets (for loop nesting)

Table 3.3: Parameters of ZOL.

(6) Zero Overhead Loop (ZOL)

Zero Overhead Loop instructions can be reduced loop overhead including compare instructions to check the end of loop, and jump instruction to return to the beginning of loop. ZOL is used by many commercial DSP architectures [19, 20, 21, 22]. ZOLs of commercial DSPs are classified into three category: (1) Method specifying loop end address, (2) Method specifying the number of loop instructions, (3) Method using continue instruction. However, taking into account of the number of instructions when ZOL of each category is executed, method (3) is not superior to methods (1) and (2). The instruction counts of each method is: (1) $m \times i + 2$, where the loop end address setting and the loop begin address are specified 1 instruction, and the number of loop body instructions is mand iteration is i times, (2) $m \times i + 2$, where the loop end address setting and the number of loop body instructions are specified 1 instruction, and the number of loop body instructions is m and iteration is i times, (3) $m \times i + i + 1$, where the continue instruction is 1 instruction, and the number of loop body instructions is m and iteration is *i* times. From computational costs, in the proposed compiler generation method, ZOL methods (1) and (2) are supported.

Fig.3.3 shows parameters of ZOL. The common parameters of each ZOL method are loop counter size and the number of special register sets. Moreover, the method (1) has the start address size, and the method (2) has the start address size, the number of instructions, and instruction buffer that is used as local cache.

```
SETEND {
    operand {
        label any a;
        EADDR any b;
    }
    format {
            "SETEND" a
    }
        ... (snip)
        behavior {
            b = end_set(a);
        }
}
```

Figure 3.11: Example of ZOL instruction (SETEND).

The input format to specify ZOL parameters is described using storage specification and instruction behavior specification. If the designer would like to use method (1), he describes loop counter, start address register in storage specification. If the designer would like to use method (2), he specifies instruction counter. Then, he describes the instruction using iter_set() which means setting iteration count, start_set() which means setting start address, loop_start() which means starting the loop, end_set() which means setting loop end address. The proposed compiler generator detects the ZOL instructions and registers, and outputs to ZOL internal representation. Mapping rule of ZOL internal representation is assigned pseudo-instructions. When ZOL instructions are defined, the compiler generator also produces the filter for ZOL instruction. The filter is used to change the instruction format from pseudo-instructions to real instructions. The reason why pseudo-instruction is used is that the number of instruction is not determined before assembly code is emitted.

3.6.3 Generation of Scheduling Information

The compiler generator produces scheduling information. The algorithm is shown in Fig. 3.12. The generation of the scheduling information involves the following 3 steps: instruction classification, resource tracing, and throughput and latency calculation.

In the instruction classification step, instructions are classified by resource usage in the instruction and, its throughput and latency. For example, if the "xor" function and addition function use the same resource such as ALU and these functions have the same throughput and latency, these instructions are classified into the same class. In the resource tracing step, to obtain connections of function interfaces between resources, the compiler generator traces the resource graph translated from the processor structure using a resource connection graph. In the throughput and latency calculation step, the throughput and latency of instructions are calculated using the throughput and latency of the resources. The maximum value of all pipeline stages determines throughput, which is the ratio of instructions per cycles. Latency is the total value of the resource latencies from the execution stage to the write-back stage.

3.7 Summary

In this chapter, the compiler generation method for PEAS-III is proposed. The PEAS-III system is one of the ASIP development systems. Designers describe processor specification using the PEAS-III input environment. The HDL generator and the proposed compiler generator get the description from the input environment. Then, the HDL generator output the target processor description with accessing FHM-DBMS. FHM is parameterized resource model. When designers would like to change the characteristics of resource, he only changes the parameters of resource. Moreover, FHM has estimation method which produces hardware cost, delay time, power consumption, throughput cycle, and latency cycle. From this estimation result, the user can select the best solution from a lot of candidates easily.

3.7. SUMMARY

```
// Instructions are classified by resource and function.
while !( all instruction classes are calculated. )
{
     // "ready" is a set of write functions.
     ready = GetReadySet;
     while !( pipeline stages from execute stage
           to write storage stage are calculated.)
     {
           while !( all paths which are in
                 same pipeline stage are calculated. )
           {
                // A set of next resources
                nextReady = GetPredecessors(ready);
                // Get throughput and latency
                // which is used by this instruction.
                throughputTemp = GetThroughput(ready);
                latencyTemp += GetLatency(ready);
                if ( throughput > throughputTemp )
                 {
                      throughput = throughputTemp;
                 }
                ready = nextReady;
           }
     latency += latencyTemp;
     }
}
```

Figure 3.12: Algorithm of scheduling information generation.

The proposed compiler generation flow is as follows: (1) analysis of the target instruction-set, and categorizing the instructions using the analysis result, (2) mapping rule generation for code emission, and (3) scheduling information generation for code scheduling. In step (1), instructions are categorized into the following categories: (a) each arithmetic, logical and compare operation such as addition, subtraction and so on, (b) control instructions such as jump and branch, (c) load/store instructions, (d) Compiler-Known-Functions for special instructions. In step (2), mapping rules for code emission are generated. Mapping rules produce the relationships between internal representations of compiler and target instructions. In arithmetic, logical, and compare operations and their combinations, relationship between one instruction and one mapping rule can be made. However, in if-then-else statements, function call, and address calculation instructions, relationship one instruction and one mapping rule cannot be made. In the proposed compiler generation method, the instruction for the case of multiple instructions to one mapping rule is automatically selected using instruction category. The control instructions and stack manipulation instructions can be selected using selection algorithm explained in previous sections. In step (3), scheduling information is produced. When the instructions are scheduled, throughput and latency are required. The proposed compiler generator calculates the throughput and the latency of the instruction group which uses the same resources when the member instruction is executed.

Next chapter describes experiments to examine the proposed compiler generation method.

44

Chapter 4

Experiments

4.1 Experiment 1

4.1.1 Objective

The objective of this experiment is to evaluate the proposed compiler generator when it is used for many types of instruction sets processors.

4.1.2 Target Processors

The target processors are as follows:

(1) 32 bits RISC instruction set

(a)Architecture Type is Load/Store architecture, Harvard architecture, and pipeline architecture which has five pipeline stages. (b)Functional Units are load/store unit, ALU, multiplier, divider, shifter, and address calculation unit. (c)Addressing modes include direct register access, in-direct memory access. (d)Register file includes thirty two 32-bit registers.

(2) 16 bits CISC instruction set

(a)**Architecture Type** is Load/Store architecture, Harvard architecture, and pipeline architecture which has eight pipeline stages. (b)**Functional Units** are load/store

	Processor 1	Processor 2
Hardware Cost (K gates)	57.28	77.84
Performance (μs)	6.68	33.5
Power (mW)	30.8	75
Max Clock Fre- quency (MHz)	89.4	132.9

Table 4.1: Design Result of Processor 1 and Processor 2.

Table 4.2: Design Time of Processor 1 and Processor 2.

		Processor 1	Processor 2
Design	time	8	23 + 59
(hours)			

unit, ALU, multiplier, divider, shifter, address calculation unit, accumulator, and bit operation unit. (c)Addressing modes include direct register access, direct memory access, in-direct memory access, memory access with post-increment, and memory access with pre-decrement. (d)Register file has sixteen 8-bit registers and eight 16-bit registers. Moreover, eight 32-bit registers can be used. The 32-bit register overlaps two 16-bit registers, and 16-bit register overlaps two 8-bit registers.

4.1.3 Applications and Environment of the experiment

The FIR filter, a typical DSP application, was used in these experiments. Every processor was synthesized by a Synopsys Design Compiler using the 0.14 μ m CMOS standard cell library.



Figure 4.1: Amount of Descriptions (lines).

4.1.4 Results

Table 4.1 shows the design result of processor 1 and processor 2. The design result includes hardware cost, performance and dynamic power, when the processors executed FIR filter application.

Table 4.2 shows the design time for each processor. The design time for processor 1 was 8 hours, which includes the processor 1 architecture description in PEAS-III. The design time of processor 2 was 82 hours, which includes 23 hours for processor 2 architecture description with PEAS-III and 59 hours for designing the components for processor 2. From Table 4.2, the processors are designed in a short time, once the components of the processors have been designed.

Figure 4.1 shows the amount of description for the proposed compiler generator. "Described" denotes that designers describe this part. "Resource" denotes the description of the timing specification. Designers do not have to describe this part because it is obtained from FHM-DBMS. "Generated" denotes the lines that designers do not describe because this description is produced from micro operation description which is a part of PEAS-III machine description.

4.1.5 Discussion

In this experiment, the proposed compiler generator produced compilers for the target processors that have many types of instruction sets. Moreover, the target compilers were generated in a short design time. Using the PEAS-III system, designers can describe instruction sets with about 10 minutes per instruction. Hence, ASIPs and compilers are produced in reasonable time using PEAS-III. The lines of each description were about 3.5 K lines and 5 K lines, respectively. However, the lines described for processor 1 and processor 2 by designers were about 1.2 K and 1.7 K, respectively. This is because the timing specification is produced by FHM-DBMS, and the structural description is translated from micro operation description. Therefore, designers can describe processor specifications rapidly.

In this experiment, hardware cost of processor 1 was larger than that of processor 2, and performance of processor 1 was better than that of processor 2. The reason is as follows. In address calculation, many spill codes were generated in processor 2. 32-bit registers were used when the processor accessed the memory, but the number of 32-bit registers was not sufficient to store temporal values.

4.2 Experiment 2

4.2.1 Objective

The objective of this experiment is to evaluate the target processor using PEAS-III, when the configuration of processor core is changed.

4.2.2 Base Processor

The base processor used in this experiment was processor 1, which was the same processor as that in experiment 1.

4.2. EXPERIMENT 2

4.2.3 Applications and Architecture Candidates

DCT and FIR filter were used in these experiments. Multiply and shift instructions are used in DCT, and multiply and add instruction are commonly used in FIR filter. Therefore, the "MAC (Multiply and Accumulate)" and the "MSRA (Multiply and Shift Right Arithmetic)" instructions were added to the base processor in this experiment. Moreover, the size of the register file was changed among 8, 16 and 32 registers, because the size of the register file affects the area of the CPU core and execution cycles. In addition, in order to take trade-offs between hardware cost and performance into consideration, the number of pipeline stages was varied among 3, 4 and 5 stages.

4.2.4 Results

Figures 4.2 and 4.3 show trade-offs between hardware cost and performance in DCT and FIR filter. The horizontal axis in Fig. 4.2 and 4.3 indicates hardware cost of the processor core, and the vertical axis indicates the execution time of applications. In Figs. 4.2 and 4.3, "Base" denotes the processor core which has the processor 1 instruction set. "MAC" denotes the processor core where MAC instruction was added and "MAC and MSRA" denotes the processor core where MAC and MSRA instructions were added to the "Base" processor. As shown in Figs. 4.2 and 4.3, the trade-offs between hardware cost and performance existed, when the size of register file, the number of pipeline stages and the instruction set were changed.

Table 4.3 shows modification cost using the PEAS-III system. The time to design the base processor was eight hours. The modification cost of pipeline stages was half an hour. Moreover, adding each MAC and MSRA instruction takes half an hour. The total modification cost of all these experiments was only 4.1 hours.



Figure 4.2: Trade-offs Between Hardware Cost and Performance (DCT).



Figure 4.3: Trade-offs Between Hardware Cost and Performance (FIR Filter).

	cost (hour)
Base processor design	8
Pipeline stage count	0.5×2
MAC	0.5
MSRA	0.5
Size of Register File	0.3×2
Other	1.5
Total	12.1

Table 4.5: Mounication Co

4.2.5 Discussion

When designers select the processor architecture, they must consider trade-offs among hardware cost, performance and power consumption. Using the PEAS-III system, not only the processor HDL description but also its target compiler are generated when designers change the configuration of the processor core. Therefore, designers can explore the design space more efficiently by using the PEAS-III system rather than other systems including compiler generators, which are discussed in chapter 2. Moreover, the modification cost of these experiments is only a few hours, because resource features such as the size of the register file can be changed only by setting the parameters in the PEAS-III system. This indicates that the PEAS-III system enables rapid exploration of the design space.

In this experiment, MAC and MSRA instructions did not affect the performance of processors. This is because the maximum frequency of processors is reduced on adding the resources, with all reducing execution cycles. Using the PEAS-III system, not only execution cycles but also the maximum frequency can be evaluated.

4.3 Experiment 3

4.3.1 Objective

The objective of this experiment is to evaluate whether special instructions can be efficiently used by the target compiler. Furthermore, code quality is evaluated in this experiment.

4.3.2 Target Application

The target application was a complex coefficient FIR filter. Each complex data is organized as follows: (a) bit width was 32 bits, (b) real part of data was from the 16th bit to the 31st bit, (c) imaginary part was from the 0th bit to the 15th bit, and (d) the format of each part was a fixed-point number.

4.3.3 Target Processors

The base processor used in this experiment was processor 1, used in the experiment 1. Moreover, special instructions were added to the base processor. The special instructions were as follows: (1) CMULT calculates complex multiply and accumulate, (2) SETCPOS sets arithmetic point in the imaginary part, (3) SETR-POS sets arithmetic point in the real part, (4) ACMCLR sets accumulator value to zero, and (5) CLOAD moves accumulator value to general purpose registers. The FIR filter application was written by using compiler known functions, which operate each special instruction.

4.3.4 Results

Figure 4.4 shows a comparison of code quality between the code generated by a compiler and the code written by a designer. In Fig. 4.4, (a) denotes the code before instruction addition, (b) denotes the code after instruction addition, and (c) denotes the hand assembly code. The code sizes of (a), (b) and (c) were 624 bytes, 464 bytes, and 204 bytes respectively. Execution cycles of (a), (b) and (c) were 14593 cycles, 3665 cycles, and 2234 cycles, respectively. From Fig. 4.4, the



Figure 4.4: Code Quality Comparison Among (a) Code Before Instruction Addition, (b) Code After Instruction Addition, and (c) Hand Assembly Code.

code size of (b) was about 2.2 times larger than that of (c). The execution cycle of (b) was about 1.6 times larger than that of (c), and the execution cycle of (b) was about 3.5 times larger than that of (a).

4.3.5 Discussion

From experiment 3, special instructions such as CMULT and so on can be used in the generated compiler. Using special instructions, machine suitable codes can be generated. Moreover, comparing (a) and (b), special instructions play an important role in improving the performance of the processor, and designers evaluates the effect of special instructions using the proposed compiler generator. The code size of (b) was 2.2 times larger than that of (c), but the execution cycles of (b) were 1.6 times larger than that of (c). This is because loop optimizations such as loop invariant, loop strength reduction and so on effectively reduce the cost of iterations.

4.4 Case Study

4.4.1 Objective of Case Study

Objective of this case study is to evaluate effectiveness of ASIP design method and the proposed ASIP development environment. Particularly, it is evaluated that design space exploration time using the PEAS-III system when designers develop an application system used in real world. Target applications of ASIP include digital signal processing (DSP) such as JPEG, MPEG, network system, wireless communication system such as mobile phone. JPEG is one of the target applications of ASIP, and JPEG is used for a lot of systems such as digital camera, mobile phone with camera, and so on. Hence, JPEG is a good example to confirm effectiveness of ASIP design method and the proposed ASIP development environment.



Figure 4.5: JPEG Encoder Procedures based on the DCT.

4.4.2 Target Application: JPEG Codec

JPEG is a definition of a still-image compression algorithm established by the JPEG committee. Fig. 4.5 shows JPEG encoder procedures based on the DCT. In the encoding process, the input component's samples are grouped into 8×8 blocks, and each block is transformed by the DCT into a set of 64 values referred to as DCT coefficient. The first element is referred to as the DC coefficient and the other elements are referred to as the AC coefficients. Each of the 64 coefficients is then quantized using one of 64 corresponding values from a quantization table. After quantization, the DC coefficient and the 63 AC coefficients are prepared for Variable Length Coding (VLC) which compresses the DC and AC coefficients. In JPEG specification, one of two coding procedures can be used. One is Huffman encoding and the other is arithmetic coding.

4.4.3 Architecture Candidates

Several kinds of parameters are defined in JPEG specification. In this case study, 8 bit precision baseline algorithm was selected. Huffman coding was selected as VLC and VLD. In the following section, architecture candidates are described, and experimental results are explained.

4.4.3.1 DCT and IDCT

DCT and IDCT are designed using Chen DCT algorithm [23], which is one of the famous algorithm reducing multiplications and additions. Data flow of Chen DCT is shown in Fig. 4.6. Here, x(i) denotes element of input matrix, X(i) denotes



Figure 4.6: Data Flow of Chen DCT (1-dimensional 8 points).

transformed element. Ci_j and Si_j denote $\cos(\frac{i \times \pi}{j})$ and $\sin(\frac{i \times \pi}{j})$, respectively. Using Chen algorithm, multiplication times are reduced from 64 to 16, and addition times are reduced from 56 to 26 in 1 dimensional 8 points DCT. IDCT can be designed using inverse of DCT. Hence, multiplication and addition times in IDCT are reduced as much as those of DCT.

There are several approaches in DCT and IDCT design.

• Sequential Instructions Approach

Sequential instructions approach stands for software design. All of the algorithm is processed by software.

• DCT Instruction Approach

DCT instructions approach stands for hardware unit design. All of the algorithm is processed by hardware.

• Butterfly Instructions Approach

Butterfly instructions approach stands for design using fine grain instructions. The part of the algorithm is processed by hardware, and the other part of the algorithm is processed by software.

Figure 4.7: C Source Code of Quantization.

These approaches have trade-offs between hardware cost and performance.

4.4.3.2 Quantization

In quantization design, several approaches exist, which is the same as DCT design. Fig. 4.7 shows the C source code of quantization. From Fig. 4.7, quantization divides the element by the element of quantization table. Hence, the performance of divider affects the execution cycles of quantization. In this case study, the algorithm of divider was changed.

4.4.4 Input Image

In this evaluation, a standard image (Fig. 4.8) was used as an input image. The image size was 256×256 pixels and the sampling factors of each component were as follows: horizontal sampling factors of Y, U, V were 4, 1, 1, and vertical

4.4. CASE STUDY



Figure 4.8: Sample Color Image (Lenna).

sampling factor were 4, 1, 1, respectively.

4.4.5 DCT/IDCT Unit

Fig. 4.9 shows the DCT/IDCT unit that processes 2 dimensional (2-D) 8 points DCT/IDCT. The input and output ports of DCT/IDCT unit consist of as follows: (a) input or output 32-bit data bus, (b) input port of 32-bit base address for data read/write, (c) 32-bit data address bus, (d) 1-bit calculation mode signal to change DCT execution or IDCT execution, (e) 1-bit start signal, and (f) 1-bit fin signal. Functional blocks consist of 8 blocks: 16-bit internal registers, ADD block1, ADD block2, MADD1, MADD2, MADD3, address unit, and controller. ADD block1, ADD block2, MADD1, MADD2, and MADD3 execute part of Chen DCT data flow illustrated in Fig. 4.6. ADD block1 has 4 input ports (in1, in2, in3, in4), and 4 output ports (out1, out2, out3, out4). Each adder calculates using the following equation: out1 = in1 + in2, out2 = in1 - in2, out3 = -in3 + in4, out4 = in1 - in2in3 + in4. ADD block2 has 8 input ports (in1, in2, in3, in4, in5, in6, in7, in8), and 8 output ports (out1, out2, out3, out4, out5, out6, out7, out8). Each adder calculates using the following equation: out1 = in1 + in8, out2 = in2 + in7, out3 = in3 + in6, out4 = in4 + in5, out5 = in4 - in5, out6 = in3 - in6,out7 = in2 - in7, out8 = in1 - in8. MADD1 has 2 input ports (in1, in2) and 2 output ports (out1, out2). MADD1 unit calculates using the following equation: $out1 = \cos(\frac{1\times\pi}{4}) \cdot in1 + \cos(\frac{1\times\pi}{4}) \cdot in2, out1 = \cos(\frac{1\times\pi}{4}) \cdot in1 - \cos(\frac{1\times\pi}{4}) \cdot in2.$



Figure 4.9: DCT/IDCT Unit.

MADD2 has 2 input ports (in1, in2) and 2 output ports (out1, out2). MADD2 unit calculates using the following equation: $out1 = \sin(\frac{1\times\pi}{8}) \cdot in1 + \cos(\frac{1\times\pi}{8}) \cdot in2$, $out2 = \sin(\frac{3\times\pi}{8}) \cdot in1 + \cos(\frac{3\times\pi}{8}) \cdot in2$, MADD3 has 4 input ports (in1, in2, in3, in4) and 4 output ports (out1, out2, out3, out4). MADD3 unit can change calculation mode to use the same unit twice in Chen DCT/IDCT calculation flow. MADD3 unit calculates using the following equation: $out1 = \sin(\frac{1\times\pi}{16}) \cdot in1 + \cos(\frac{1\times\pi}{16}) \cdot in4$, $out2 = \sin(\frac{5\times\pi}{16}) \cdot in2 + \cos(\frac{5\times\pi}{16}) \cdot in3$, $out3 = -\sin(\frac{3\times\pi}{16}) \cdot in2 + \cos(\frac{3\times\pi}{16}) \cdot in3$, $out4 = -\sin(\frac{7\times\pi}{16}) \cdot in1 + \cos(\frac{7\times\pi}{16}) \cdot in4$, or out1 = in1 + in4, out2 = in2 + in3, out3 = in2 - in3, out4 = in1 - in4. Each value is calculated in 16-bit fixed point arithmetic.

Fig. 4.10 shows the finite state machine of DCT/IDCT unit. The finite state machine consists of two part. One is 1-D Chen DCT calculation control part, the other is 2-D DCT calculation control part. In 2-D part, first step calculates row of matrix and second step calculates column of matrix. In each step, 1-D Chen DCT is executed 8 times. In 1-D part, the flow consists of data read, 4 steps execution illustrated in Fig. 4.6, and data write. The DCT/IDCT unit fetches data from the data memory to the internal registers. When the DCT/IDCT unit fetches data that is from row of matrix, one 16-bit value can be fetched using an address. In

4.4. CASE STUDY



Figure 4.10: Finite State Machine of DCT/IDCT Unit Controller.

column data of matrix, two 16-bit values can be fetched using an address. Hence, the number of memory accesses when the DCT/IDCT unit fetches from row of matrix is 8, and the number of memory accesses when the DCT/IDCT unit fetches from column of matrix is 4. From this feature, the number of memory accesses can be reduced when the DCT/IDCT unit is used. The reason why the DCT/IDCT unit has 32-bit data bus is that the data is allocated to the data memory which is the same memory of ASIP.

4.4.6 Additional Instructions

• DCT

DCT instruction executes the procedure of DCT. This instruction uses the DCT unit described in section 4.4.5. Instruction set specification for PEAS-III is described in Fig. 4.11. In application written in C language, DCT is described using function call. In PEAS-III specification, Compiler-Known-Function "dct" is defined, and the behavior of DCT instruction is defined using "dct" function. Micro-Operation description defines pipeline execution. DCT unit is executed at pipeline stage 4.

```
(a) Behavior Description for Compiler Generator
```

```
ckf prototype {
     void dct ( unsigned int , unsigned int );
}
```

```
DCT {
     operand {
          GPR UInt31to0 a;
          GPR UInt31to0 b;
     }
     format { "DCT" "," a "," b }
     function {
     stage(1) { PC.read IMEM.load_word
          PC.inc IR.read }
     stage(2) {GPR.read0 GPR.read1 }
     stage(3) { }
     stage(4) { DCT0.dct }
     stage(5) { }
     }
     behavior {
          dct ( a , b );
     }
}
```

(b) Micro-Operation Description

```
stage(1){ IR := IMEM[PC]; PC.inc();},
stage(2){ $op1 := GPR.read0(rs); $op2 := GPR.read1(rt);},
stage(3){},
stage(4){$dummy := DCT0.dct($op1,$op2);},
stage(5){}
```

(c) Bit Field 000000 rs rt 000000000 111111

Figure 4.11: DCT Instruction Specification of PEAS-III.

4.4. CASE STUDY

• MADD1

MADD1 instruction calculates the MADD1 block in Fig. 4.6. MADD1 instruction takes 2 operands as input and write back to the same operand registers. Instruction set specification for PEAS-III is described in Fig. 4.12. In application written in C language, MADD1 is described using function call. In PEAS-III specification, Compiler-Known-Function "madd1" is defined, and the behavior of MADD1 instruction is defined using "madd1" function. MADD1 unit is executed at pipeline stage 3.
```
(a) Behavior Description for Compiler Generator
```

```
ckf prototype {
     void madd1 ( unsigned int , unsigned int );
}
```

```
MADD1 {
     operand {
          GPR UInt15to0 a;
          GPR UInt15to0 b;
     }
     format { "MADD1" "," a "," b }
     function {
     stage(1) { PC.read IMEM.load_word
          PC.inc IR.read }
     stage(2) {GPR.read0 GPR.read1 }
     stage(3) {MADD1U0.madd }
     stage(4) \{ \}
     stage(5) {GPR.write0 GPR.write1 }
     }
     behavior {
          madd1 ( a , b );
     }
}
```

(b) Micro-Operation Description

```
stage(1){ IR := IMEM[PC]; PC.inc();},
stage(2){ $op1 := GPR.read0(rs); $op2 := GPR.read1(rt);},
stage(3){($result1, $result2) := MADD1U0.madd($op1, $op2);},
stage(4){ },
stage(5){GPR.write0($result1, rs); GPR.write1($result2, rt); }
```

(c) Bit Field 000000 rs rt 000000000 011110

Figure 4.12: MADD1 Instruction Specification of PEAS-III.

• MADD2

MADD2 instruction calculates the MADD2 block in Fig. 4.6. MADD2 instruction takes 2 operands as input and write back to the same operand registers. Instruction set specification for PEAS-III is described in Fig. 4.13. In application written in C language, MADD2 is described using function call. In PEAS-III specification, Compiler-Known-Function "madd2" is defined, and the behavior of MADD2 instruction is defined using "madd2" function. MADD2 unit is executed at pipeline stage 3.

```
(a) Behavior Description for Compiler Generator
```

```
ckf prototype {
    void madd2 ( unsigned int , unsigned int );
}
```

```
MADD2 {
     operand {
          GPR UInt15to0 a;
          GPR UInt15to0 b;
     }
     format { "MADD2" "," a "," b }
     function {
     stage(1) { PC.read IMEM.load_word
          PC.inc IR.read }
     stage(2) {GPR.read0 GPR.read1 }
     stage(3) {MADD2U0.madd }
     stage(4) { }
     stage(5) {GPR.write0 GPR.write1}
     }
     behavior {
          madd2 ( a , b );
     }
}
```

(b) Micro-Operation Description

```
stage(1){ IR := IMEM[PC]; PC.inc();},
stage(2){ $op1 := GPR.read0(rs); $op2 := GPR.read1(rt);},
stage(3){($result1, $result2) := MADD2U0.madd($op1, $op2);},
stage(4){ },
stage(5){GPR.write0($result1, rs); GPR.write1($result2, rt);}
```

(c) Bit Field 000000 rs rt 000000000 011111

Figure 4.13: MADD2 Instruction Specification of PEAS-III.

4.4.7 Compiler Generation for Target Processors

The target compiler is generated using processor specification partly represented in previous section. The target compiler produced by the proposed compiler generator executes the following steps: (1) Parsing the source code, (2) Machine independent optimization, (3) Syntax tree rewriting and pattern matching, (4) Register allocation and Spill code insertion, (5) Instruction scheduling, (6) Machine dependent optimization, and (7) Output assembly code. When special instructions such as DCT, MADD1 and so on are added to the processor specification, the proposed compiler generation method produces the following information: (a) function prototypes for C parser, (b) mapping rules for special instructions, and (c) instruction throughput and latency table for instruction scheduling. When parser reads the special instructions written in target application, the generated compiler makes CKF internal representation for compiler. When back-end of compiler generates assembler, target instruction is emitted using mapping rule for CKF. For example, DCT function is read by the compiler and the internal representation "xirCKF" is generated, which means that extended internal representation "CKF". The "xir-CKF" has attributes that include operands and CKF ID. The mapping rule for "xirCKF" specifies assembly format which is specified in format section. For instance, in DCT instruction in Fig. 4.11, the mapping rule of DCT instruction includes instruction string "DCT" and the operand order of DCT instruction "a" and "b". Furthermore, instruction latency and throughput are calculated using resource usage described in function section of instruction behavior specification. Resource throughput and latency can be obtained from FHM-DBMS. The proposed compiler generator traces the resource connection graph and calculates instruction throughput and latency.

4.4.8 How to Estimate Design Quality

Hardware Cost and maximum clock frequency were estimated using Synopsys Design Compiler. Input of Design Compiler was synthesizable HDL generated by PEAS-III. 0.14 μ m CMOS standard cell library (voltage 1.5 V) was used for logic synthesis. Execution cycle was estimated using Synopsys Scirocco that is a cycle-

	Multiplier	Divider	Area	Max	Exec Cy-	Power
				Freq.	cles	
			(K	(MHz)	(M	(mW /
			gates)		cycles)	MHz)
1. Normal	seq(32)	seq(34)	39.43	151	61.28	2.40
2. Normal	seq(32)	array	52.1	22.5	51.19	2.44
3. Normal	array	seq(34)	57.59	44.5	44.54	2.48
4. Normal	array	array	70.19	43.3	34.45	2.53
5. Butterfly	seq(32)	seq(34)	57.3	149	53.57	2.48
6. Butterfly	seq(32)	array	70.0	23.0	43.48	2.52
7. Butterfly	array	seq(34)	75.5	44.5	43.52	2.56
8. Butterfly	array	array	88.0	23.0	33.43	2.61
9. DCT	seq(32)	seq(34)	71.17	151	39.62	2.49
10. DCT	seq(32)	array	89.35	22.4	29.53	2.54
11. DCT	array	seq(34)	83.86	43.3	36.25	2.58
12. DCT	array	array	101.93	43.3	26.17	2.62

Table 4.4: Processor Cores and Their Execution Cycles of JPEG Application.

Library: 0.14 CMOS Standard Cell Library.

based HDL simulator. Dynamic power was estimated by gate-level simulation using Mentor Graphics ModelSim and Synopsys Power Compiler.

4.4.9 Processor Organization

Processor organization in this case study is shown in Table. 4.4. Normal denotes base instruction set that is sub set of MIPS-R3000 instruction set. Butterfly denotes instruction set added MADD1, and MADD2 instructions. DCT denotes instruction set added DCT instruction. The hardware algorithm of multiplier is sequential type that executes 32 cycles and array type that executes 1 cycle. On the other hand, the hardware algorithm of divider is sequential type that executes 34 cycles, and array type that executes 1 cycle.



Figure 4.14: Trade-offs Between Hardware cost and Execution Cycles When JPEG Encoder was Executed.

4.4.10 Trade-offs Between Hardware Cost and Performance

Fig. 4.14 shows trade-offs between hardware cost and execution cycles when JPEG encoder has been executed. Horizontal axis is hardware cost, and vertical axis is execution cycles. The number of each plot point in Fig. 4.14 corresponds to each processor in Table 4.4. From Fig. 4.14, the trade-off between hardware cost and execution cycles exists when instructions are added and the hardware algorithms are changed.

Figs. 4.15 and 4.16 show trade-offs between hardware cost and execution time when JPEG encoder has been executed. Horizontal axis is hardware cost, and vertical axis is execution time. In Fig. 4.15, execution time was calculated using execution cycles and clock frequency that was 66 MHz, and in Fig. 4.16, execution time was calculated using execution cycles and clock frequency that was 40 MHz. As shown in these figures, the number of architecture candidates was changed because the max clock frequency of each architecture candidate ranges between about 20 MHz and 150 MHz. These results show that designers have



Figure 4.15: Trade-offs Between Hardware Cost and Execution Time When JPEG Encoder was Executed. (66 MHz)

to consider not only the execution cycles of an application, but also the clock frequency when architecture candidates are selected. In Fig. 4.15, when a design constraint is that hardware cost is under 60 K gates, the processor No. 5 in Table 4.4 is selected as the optimal architecture.



Figure 4.16: Trade-offs Between Hardware Cost and Execution Time When JPEG Encoder was Executed. (40 MHz)

4.4.11 Trade-offs Between Hardware Cost and Power Consumption

Figs. 4.17 and 4.18 show trade-offs between hardware cost and power consumption when JPEG encoder has been executed. The horizontal axis is hardware cost, and the vertical axis is dynamic power. In Fig. 4.17, JPEG Encoder was executed within 0.5 second, and in Fig. 4.18, JPEG Encoder was executed within 1 second. In Fig. 4.17, the frequency of processor 1 was about 120 MHz, the frequency of processor 9 was about 90 MHz. Hence, the dynamic power of processor 1 in Fig. 4.17 was about 290 mW, and the dynamic power of processor 9 was about 190 mW. If design constraint of power consumption is 200 mW, the processor 9 can be selected, but if design constraint of power consumption is 300 mW, processor 1 can be selected because the hardware cost of processor 1 is smaller than that of processor 9.

Furthermore, if design constraint of execution time is within 1 second, the trade-off between hardware cost and power consumption is Fig. 4.18. In Fig. 4.18, processors 5 and 7 cannot be architecture candidates.



Figure 4.17: Trade-offs Between Hardware Cost and Power Consumption When JPEG Encoder was Executed Within 0.5 Second.



Figure 4.18: Trade-offs Between Hardware Cost and Power Consumption When JPEG Encoder was Executed Within 1 Second.

	Time (hour)
C source code design	130
DCT unit design	60
Total	190
Base processor design	12
Registration of DCT unit and	1
Convolution blocks to FHM-	
DBMS	
Instruction addition	1
Hardware algorithm selection	0.1
Others	150
Total	164.1

Table	4.5:	Design	Time.
raoio	1.2.	Dosign	THUC.

4.4.12 Design Time

The design time of the case study is shown in Table 4.5. From Table 4.5, about ten hours were spent using the PEAS-III system. Here, the reason why the hard-ware algorithm selection time is short is only changing FHM parameters to select hardware algorithm. From this result, the hardware description and the target compiler can be designed in a short design time. 130 hours were spent designing JPEG codec using C source code. 60 hours were spent DCT unit design. Others include debug time and simulation time and synthesizing time to evaluate the processor core. It seems that the time of JPEG codec application design and DCT unit design is as long as other environments.

4.4.13 Discussion

The experimental result shows that architecture candidates are changed when clock frequency or time constraint are changed. From this result, designers must consider not only the execution cycles of a target processor but also the max frequency of a target processor and power consumption. For example, in Fig. 4.17,

4.4. CASE STUDY

processor 5 can be an architecture candidate. However, in Fig. 4.18, processor 5 is not an architecture candidate because processor 3 can achieve low power and the same hardware cost. In the PEAS-III design, software development environment and designed processor's HDL descriptions are generated at the same time. Hence, designers can consider the execution cycles of application, the clock frequency of processor, hardware cost and power consumption efficiently.

When an application such as DSP application is designed using ASIPs, designers consider trade-offs among hardware cost, performance and power consumption. Generally, it is said that the design time of hardware description, compiler and assembler require several months or at least several weeks. However, it is too long to meet a requirement of the design time in design space exploration. On the other hand, when designers use other ASIP development systems that have been explained in section 1, either software development environment or hardware description is produced in a short time, but the other part, for example processor cores for software development environment, must be developed by themselves. The advantage of the PEAS-III system is that compiler, assembler and hardware description are generated at the same time. Furthermore, the modification cost of the design is low, and hardware modules such as DCT unit can be reused easily, because designers only select modules from FHM-DBMS as resources. Using the PEAS-III system, designers can evaluate processors and select an optimal architecture in a short design time.

The architecture candidates described in section 4.4.3 were selected from the feature of C source code or data flow. Although a lot of candidates can be considered, several architecture candidates that were expected to improve processor performance were designed to evaluate the potential of PEAS-III design method in this case study. Generally, architecture candidates selection is very difficult. Hence, the profiling environment to select architecture candidates and architecture selection method are needed to reduce design cost and to get better solution.

In table 4.5, the time of others includes debug time and simulation time of target processor. To reduce this part, a source code debugger and a faster simulator are desirable.

4.5 Summary

In this chapter, experiments using the proposed compiler generation method were explained. In experiment 1, development time and the amount of description were evaluated using two architectures. In experiment 2, 27 architectures were evaluated using FIR filter and DCT. In experiment 3, the proposed compiler generator was evaluated using a real application: JPEG encoder. Experimental results show that designers can efficiently evaluate numerous architecture candidates by means of execution cycles of applications, clock frequency and hardware cost of the processor core when they use the PEAS-III system. Therefore, designers can rapidly explore design space and explore trade-offs of designs by using the PEAS-III system.

Next chapter describes discussion of the result which was explained in this section.

Chapter 5

Discussion

In this chapter, feasibility of the proposed compiler generation method and impact of design productivity of SoC processor are discussed. The following sections discuss compiler retargetability, code quality of the generated compiler, design productivity of SoC processor, and design space exploration using the proposed compiler generation method.

5.1 Compiler Retargetability

In chapter 1, compiler retargetability has been discussed. Automatically retargetable compiler includes a set of parameters that changes the characteristics of base processor. The method for compiler generation using parameterized generic processor core such as PEAS-I, Satsuki, Xtensa and so on is automatically retargetable. These systems can easily produce the target compiler, because complexity of compiler generation is not high. However, the range of the supported processor's class is narrow. The number of registers and special instructions execute can be configured using these methods. However, the pipeline stage number, bit width of instruction or data, and instructions reduction cannot be configured using this methods.

Developer retargetable compiler can be retargeted to a wide range of processor architectures. The range includes not only the range of automatically retargetable compiler but also the pipeline stage number, bit width of instruction or data, special instructions that cannot execute in certain cycles and instructions reduction can be configured using this methods. In addition, the processor that has complex datapath can be included in the range, but spill code for the processor that has complex datapath is very difficult. However, this level compiler retarget requires expertise with the compiler systems. For example, GCC [24] is one of the developer retargetable compiler. GCC can be used for a lot of architecture such as Intel Pentium processor, IBM Power PC, MIPS architecture, ARM and so on. GCC can be retargeted to a lot of architecture, but GCC requires expertise of the compiler system. GCC users need to understand what is RTL which is an internal representation of GCC. It is difficult that designers who are not compiler experts understand RTL, because RTL is defined in order to represent high-level language such as C, C++, Java and so on. All processor designers do not have the expertise of compiler. In addition, the retargeting time is on the order of months and weeks. Hence, this type compiler is not suitable for ASIP design space exploration.

User retargetable compiler can be retargeted to the target processor by changing its instruction-set specification. Compiler generator explained in chapter 2 and the proposed compiler generator in this thesis are user retargetable. The range of configuration consists of the number of registers, special instructions in certain/uncertain cycles execution, the pipeline stage number, bit width of instruction or data, and instructions reduction. Moreover, the retargeting time is on the order of hours and days. Hence, this type compiler is suitable for design space exploration. The proposed compiler generator produces the target compiler using instruction-set specification and structure specification of processor. Designers that do not have the expertise of compiler can describe the processor specification and generate the target compiler.

When you see the aim of the compiler generation, the generation methods are categorized into two categories: (1) compiler and other software tools generation oriented method, (2) processor generation oriented method. ISDL, HMDES, EXPRESSION, LISA, FlexWare can be in the first category. Since the first category aims compiler and other tools generation, hardware resource model is not included. Hence, it is difficult to generate the synthesizable hardware description. Moreover, compiler oriented specification such as peep hole optimization

5.1. COMPILER RETARGETABILITY



Figure 5.1: Design Productivity of JPEG Encoder ASIP (From Case Study).

rules can be described in several methods in the first category. These features are suitable for compiler developers, but it is not suitable for all processor designers because not all of them are familiar with compilation techniques.

Fig. 5.1 shows design productivity of JPEG Encoder ASIP. Traditional design stands for the design using developer retargetable compiler such as GCC and RTL processor description. Proposed design methodology stands for the design using PEAS-III. When designers use the PEAS-III environment, processor and compiler can be designed within several days. In traditional design methodology, the retarget time of developer retargetable compiler is at least several weeks even if the compiler experts retarget it. Moreover, processor core must be developed individually. If compiler generation methods based on compiler oriented specification language can produce the target compiler rapidly, the design time of the processor core are not included. Hence, the proposed design methodology improves design productivity of ASIP significantly.

5.2 Code Quality of the Generated Compiler

The code quality of the generated compiler has been examined in chapter 3. In the embedded processor, code quality is one of the important factor, because memory space is limited and achieving high performance is required. The generated code size is about twice and the execution time is about 1.5 times larger than these by hand assembly code. Generally, the execution cycles using the generated code is about from 1.2 to 2 times larger than that by hand assembly code. Hence, the execution cycle using the generated code is feasible. The generated code size is, however, twice larger than that by hand assembly code. The generated compiler executes the loop specific optimizations such as loop invariant, loop unrolling and so on. Hence, the generated processor can execute the generated code 1.5 times better than that by hand assembly code. When you describe the target application assembly code such as JPEG, and MPEG, design time is on the order of months or years. Design time is, however, on the order of weeks when you use the target compiler. Hence, it is feasible to use the generated compiler when designers search an optimal architecture from a lot of candidates.

Moreover, in the proposed compiler generation method, general optimization algorithms such as dead code elimination, loop invariant and so on can be included for each target processor. These techniques are commonly used in compilers developed by compiler experts. Hence, general optimization algorithms are out of my study.

5.3 Requirements and Solutions for SoC Processors

In chapter 1, the application trends have been discussed. The trends include (1) new wireless handsets and base stations which need to support multiple mode, (2) the continued evolution of video coding standards from JPEG to MPEG1, MPEG2 and MPEG4, (3) entertainment and other embedded system which connect the Internet. These applications require upper compatibility to support legacy software or hardware. Therefore, the requirements of the SoC processor include not only high cost-performance and low power but also flexibility. Hence, one of the key

5.4. DESIGN PRODUCTIVITY OF SOC PROCESSORS

issues for SoC is ASIPs. There are, however, a lot of constraints when designers develop ASIP. Designers search the best solution from architecture candidates. When designers use the PEAS-III system, design space exploration can be in a short time, because the target compiler and the target processor are generated using the same processor specification. Moreover, designers use the generated description to develop SoC which includes ASIPs seamlessly.

5.4 Design Productivity of SoC Processors

ITRS [2, 3] predicts that the complexity and cost of design and verification of MPU products have rapidly increased to the point where thousands of engineeryears are devoted to a single design, yet processors reach market with hundreds bugs [3]. Moreover, to achieve the requirements of the SoC processor, designers search an optimal architecture from a lot of architecture candidates in a short design time. Hence, the time when designers decide the architecture is restricted. Therefore, the ASIP development environment is strongly needed in the SoC design.

PEAS-III has the well-defined parameterized model and the processor architecture specification language. Using the processor architecture specification language, the target processor description and the target compiler are generated. Generally, the development cost of the target processor and the target compiler, several months or a year are devoted to a single design. When designers would like to consider about the architecture, it is too long to develop both the target processor and the target compiler. The proposed compiler generation method enables design productivity increase from thirty to one hundred times, which is confirmed by experimental results in chapter 4. Designers can develop the target processor and the target compiler in a short design time using the PEAS-III design method.

5.5 Design Space Exploration Using the Proposed Compiler Generator

In case study of chapter 4, ASIP architecture for JPEG encoder was designed using PEAS-III. In PEAS-III, designers describe the architecture specification. Then, synthesizable HDL is produced by HDL generator, and the target compiler is produced by the proposed compiler generator using the same architecture specification. This feature can reduce iteration cost of design space exploration, because the target compiler is produced when designers make prototype of ASIPs. From experimental results, the design time of the target architecture is about 12 hours, which means that the target processor and the target compiler can be produced within order of days. In addition, RISC and CISC architecture can be supported, and special instructions such as DCT instruction can be supported by the proposed compiler generator. These results are sufficient to meet the requirements of the user retargetable compiler. Hence, it is one of the best solution that the proposed compiler generation method is used in ASIP design space exploration.

Chapter 6

Conclusion and Future Work

In this chapter, the conclusion of this thesis and the future work of this study are described.

6.1 Conclusion

In this thesis, the processor architecture and the compiler generator for embedded systems were proposed. In chapter 2, ASIP development environments have been discussed. The ASIP development environment includes generation of both processor and software development environment, such as compiler generation, instruction-set simulator generation, and so on. Several methods that the software development environment for ASIPs is produced from architecture specification languages have been proposed. These methods are classified into three categories. In the first approach, the target compiler and simulator are generated from the structure of the processor core that is described using RT-level description. This approach supports various type of the architectures like heterogeneous register files, non-orthogonal datapath, and so on. It is, however, difficult to modify the architecture because abstraction level of the description is low. In the second approach, the target compiler and simulator are produced from instruction behavior. In this approach, designers can modify the architecture easily because the abstraction level of the description is higher than RT-level description, but the class of the target architecture is limited rather than the first one. In the third approach, the target compiler and simulator are generated from the structure of the processor and the behavior of the instructions. This approach supports larger class than the second one. Moreover, the modification cost is smaller than that of the first one.

In chapter 3, a compiler generation method for ASIPs was proposed, the compiler generator was implemented for one of ASIP development system: PEAS-III, and the PEAS-III system is evaluated using case studies of DSP applications. The target compiler is produced by the proposed compiler generator using architecture specification. The architecture specification includes the following information: (1) primitive operations used by resources, (2) timing specifications of resources, (3) storage-unit specifications for memory and register allocation, (4) instruction set specification including behavior of instructions and usage of resources, and (5) the processor structure by resource connection graph. Mapping rule and scheduling information are generated using the architecture specification. Mapping rule includes arithmetic, control, load/store, spill/reload, and special hardware instructions. The proposed compiler generator analyzes the instruction-set specification, and decides each mapping rule for emitting the instructions. Experimental results show that designers can efficiently evaluate numerous architecture candidates by means of execution cycles of applications, clock frequency, hardware cost of the processor core and power consumption when they use the PEAS-III system. Therefore, designers can rapidly explore design space and explore trade-offs of designs by using the PEAS-III system. In addition, the case study shows that the proposed compiler generator can be used for a real application and improve the design time for the target compiler.

6.2 Future Work

The future work includes the following items.

6.2.1 Retargeting Algorithm for Special Architecture

DSPs have the special architecture such as SIMD, for particular applications. It is difficult that the compiler exploits these functions because these functions can-

6.2. FUTURE WORK

not be described in C language. Although the proposed compiler generator can use these functions by using Compiler-Known-Functions, designers modify the source code to use Compiler-Known-Functions. However, these kinds of retargeting is expected to be automatic.

In addition, compiler generation for the processors that have complex datapath is expected. These processors can reduce hardware cost and execution cycles for particular domain applications. It is useful that compiler generation method can exploit such processors.

6.2.2 Simulator and Profiler

To evaluate the target application or the target processor, the simulator and the profiler for the target processor are required. To retarget application specific architecture automatically, simulator and profiler generation are very important. If the features of target applications can be obtained from profiling report, architecture modification candidates can be listed. Simulator can calculates the execution cycle when target application is executed. Moreover, frequency of resources or frequency of instructions can be analyzed by using simulator. From this result, efficient instruction candidates for the target application can be reported. Furthermore, power consumption can be analyzed using frequency of resources and data type for instructions.

For example, in JPEG encoder case study in chapter 4, instructions are added to initial design. If simulator and profiler can be produced automatically, DCT instruction or butterfly instructions are reported from profiling result automatically. As a result, simulator and profiler boost ASIP modification rapidly.

6.2.3 VLIW extension

The proposed compiler generator can generate the target compiler for scalar processor. However, VLIW extension of the proposed compiler generator is needed, because VLIW processor will be used for high performance ASIPs. The configurable VILW model has been proposed in [25]. This VLIW model extends from the PEAS-III processor model. Operation dispatch model is added to the PEAS- III processor model in order to configure the number of VLIW slot, operation dispatch policy. Because the configurable VLIW model is based on the PEAS-III processor model, the target processor and the target compiler can be generated using this model. In compiler generation, instruction issue method using the proposed dispatch policy is needed.

6.2.4 Code Generation for Low Power Design

Market trends are favoring high-performance and low power systems: such as long battery life mobile phone, digital steel camera, and other mobile equipments. Gated clock and voltage control drastically reduce power consumption. More-over, low power techniques for instruction-set processor have been proposed. For example, instruction encoding is one of low power techniques [26]. To reduce instruction bus energy, instruction is encoded and frequency of data switching is reduced. This technique achieves about 75 % instruction bus transition reduction, which means that this technique can reduce power consumption of instruction bus significantly. In code generation, low power techniques are required.

6.2.5 OS Generation

Since complexity of application increase rapidly, OS generation method is an important issue for ASIP SoC. The reason is that the system development using ISR model which explained in chapter 1 is difficult. OS generation method was proposed by L. Gauthier *et.al.* [27]. OS consists of three types of components: API's, communication/system services, and device driver services. This generation method can produce from Colif specification, which defines communication in a hierarchical network of modules and behavior codes. Code size of generated OS is optimized and response time of service call is optimized for target applications.

Bibliography

- M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai, "PEAS-III: An ASIP design environment," Proceedings of 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD2000), pp. 430–436, Sept. 2000.
- [2] International Technology Roadmap for Semiconductors, "International technology roadmap for semiconductors 2001: Design." http://public.itrs.net, 2001.
- [3] International Technology Roadmap for Semiconductors, "International technology roadmap for semiconductors 2001: System drivers." http://public.itrs.net, 2001.
- [4] C. Liem, "Retargetable Compilers for Embedded Core Processors," Kluwer Academic Publishers, Dordrecht, 1997.
- [5] J. Sato, A. Y. Alomary, Y. Honma, T. Nakano, A. Shiomi, N. Hikichi, and M. Imai, "PEAS-I: A Hardware/Software Codesign System for ASIP Development," IEICE Trans. Fundamentals, vol. E77-A, no. 3, pp. 483–491, Mar. 1994.
- [6] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura, "Satsuki: An Integrated Processor Synthesis and Compiler Generation System," IEICE Trans. Inf. & Syst., vol. E79-D, no. 10, pp. 1373– 1381, Oct. 1996.

- [7] J.-H. Yang, B.-W. Kim, S.-J. Nam, J.-H. Cho, S.-W. Seo, C.-H. Ryu, *et al.*, "MetaCore: An Application Specific DSP Development System," 35th Design Automation Conference, pp. 800–803, 1998.
- [8] R. Camposano and J. Wilberg, "Embedded System Design," Design Automation for Embedded Systems, vol. 1, no. 1-2, pp. 5–50, Jan. 1996.
- [9] Tensilica, "Xtensa." http://www.tensilica.com.
- [10] R. Leupers and P. Marwedel, "Retargetable Code Generation Based on Structural Processor Descriptions," Design Automation for Embedded Systems, vol. 3, no. 1, pp. 75–108, Jan. 1998.
- [11] A. Fauth, "Beyond tool-specific machine descriptions," Code Generation for Embedded Processors, pp. 138–152, Kluwer Academic Publishers, 1995.
- [12] G. Hadjiyiannis, P. Russo, and S. Devadas, "A methodology for accurate performance evaluation in architecture exploration," 36th Design Automation Conference, pp. 927–932, June 1999.
- [13] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA Machine Description Language for Cycle-Accurate Models of Programmable DSP Architecture," 36th Design Automation Conference, pp. 933–938, 1999.
- [14] P. G. Paulin, C. Liem, T. C. May, and S. Sutawala, "Flexware: A flexible firmware development environment for embedded systems," Code Generation for Embedded Processors, pp. 65–84, 1995.
- [15] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "HMDES Version 2.0 Specification," Technical Report IMPACT-96-3, University of Illinois, 1996.
- [16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," Design And Test Conference 99, pp. 485– 490, March 1999.

- [17] M. Imai, A. Shiomi, Y. Takeuchi, and J. Sato, "Hardware/Software Codesign in the Deep Submicron Era," International Workshop on Logic and Architectural Synthesis '96, pp. 236–248, Dec. 1996.
- [18] T. Morifuji, Y. Takeuchi, J. Sato, and M. Imai, "Flexible hardware model: Implementation and effectiveness," Proc. of Synthesis And System Integration of Mixed Technologies 97, pp. 83–89, Dec. 1997.
- [19] MOTOROLA, Inc., "MSC8101 Programmer's Quick Reference." http://www.motorola.com, 2001.
- [20] Lucent Technologies., "DSP16410B and DSP16410C Degital Signal Processor, Programmer's Quick Reference Guide." http://www.lucent.com, 2001.
- [21] Analog Devices, Inc., "ADSP-21160, SHARC DSP Instruction-set Reference." http://www.analog.com, 1999.
- [22] TEXAS INSTRUMENTS, "TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set." http://www.ti.com, 2001.
- [23] W. H. Chen, C. H. Smith, and S. C. Fralick, "A fast computational algorithm for the discrete cosine transform," IEEE Trans. Commun., pp. 1004–1009, 1977.
- [24] R. M. Stallman, "Using Porting GNU CC," Free Software Foundation, Inc., http://www.gnu.org, 1995.
- [25] K. Okuda, S. Kobayashi, Y. Takeuchi, and M. Imai, "Proposal of an Architecture Model and a Simulator Generator for Configurable VLIW Processor," IPSJ Symposium Series, vol. 2002, pp. 161–166, July 2002. (in japanese).
- [26] L. Benini, G. D. Micheli, E. Macii, D. Sciuto, and C. Silvano, "Address Bus Encoding Techniques for System-level Power Optimization," Proceedings of Design Automation and Test in Europe (DATE'98), March 1998.

BIBLIOGRAPHY

[27] L. Gauthier, S. Yoo, and A. Jerraya, "Application-Specific Operating Systems Generation and Targeting for Embedded SoCs," Proceedings of the Workshop on Symthesis And System Integration of MIxed Technologies, vol. 2001, pp. 57–60, Oct. 2001.

Appendix A

BNF of Architecture Description for the Proposed Compiler Generator

A.1 Lexical Elements

<alphabets and="" numbers=""></alphabets>	::=	<let< th=""><th>ter> {</th><th><lette< th=""><th>er> •</th><th><numbe:< th=""><th>r> ".</th><th>_"}</th><th></th></numbe:<></th></lette<></th></let<>	ter> {	<lette< th=""><th>er> •</th><th><numbe:< th=""><th>r> ".</th><th>_"}</th><th></th></numbe:<></th></lette<>	er> •	<numbe:< th=""><th>r> ".</th><th>_"}</th><th></th></numbe:<>	r> ".	_"}	
<alphabets <="" td=""><td>••-</td><td>-Tet</td><td>11 1-44</td><td></td><td></td><td></td><td></td><td></td><td></td></alphabets>	••-	-Tet	11 1-44						
<alphabets small=""></alphabets>	::=	<sma.< td=""><td>11 1eti</td><td>cer> {</td><td><smail< td=""><td>LIETT</td><td>er> }</td><td></td><td></td></smail<></td></sma.<>	11 1eti	cer> {	<smail< td=""><td>LIETT</td><td>er> }</td><td></td><td></td></smail<>	LIETT	er> }		
<alphabets capital=""></alphabets>	::=	<cap:< td=""><td>ital le</td><td>etter></td><td>{ <car< td=""><td>pital .</td><td>letter</td><td>> }</td><td></td></car<></td></cap:<>	ital le	etter>	{ <car< td=""><td>pital .</td><td>letter</td><td>> }</td><td></td></car<>	pital .	letter	> }	
<alphabets string=""></alphabets>	::=	<a⊥1< td=""><td>alphal</td><td>oets> {</td><td><all< td=""><td>alphal</td><td>oets></td><td>}</td><td></td></all<></td></a⊥1<>	alphal	oets> {	<all< td=""><td>alphal</td><td>oets></td><td>}</td><td></td></all<>	alphal	oets>	}	
<all alphabets=""></all>	::=	<let< td=""><td colspan="6"><ietter> <number> "_" <blank></blank></number></ietter></td><td></td></let<>	<ietter> <number> "_" <blank></blank></number></ietter>						
<letter></letter>	::=	"a"	"b"	"C"	"d"	"e"	"f"	"g"	
		"h"	"i"	"j"	"k"	"1"	"m"	"n"	Í
		"0"	"p"	"q"	"r"	"s"	"t"	"u"	ĺ
		"v"	"w"	"x"	"У"	"z"	ĺ		
		"A"	"B"	"C"	"D"	"E"	"F"	"G"	
		"H"	"I"	ידי	"K"	"L"	"M"	"N"	i i
		"0"	"P"	"0"	"R"	"S"	"T"	нтти	
		"V"	"₩"	"X"	۳Y۳	"7."	-		1
<small letter=""></small>	••=	"a"	"h"	"c"	"đ"	"o"	# f # .	"a"	ł
<pre><capital letter=""></capital></pre>		" <u>7</u> "	"B"	"C"	ית"		 	9	i I
Capital (CCCC)	••-	л "U"	1 J 1 J T N	С нтн	1121	ند 11 T H	1 I.V.II		i
		"0"			T.	ц 1.01	11	1	1
e e			. Р. 11.71		".R."	. 2.		-0-	I.
		. v		"X"	"Y"	""."			
<natural number=""></natural>	::=	<non< td=""><td>zero n</td><td>number></td><td>• { <s:< td=""><td>ingle-o</td><td>ligit i</td><td>number></td><td>• }</td></s:<></td></non<>	zero n	number>	• { <s:< td=""><td>ingle-o</td><td>ligit i</td><td>number></td><td>• }</td></s:<>	ingle-o	ligit i	number>	• }
<nonnegative number=""></nonnegative>	::=	"0"	<nati< td=""><td>ıral nu</td><td>mber></td><td></td><td></td><td></td><td></td></nati<>	ıral nu	mber>				
<all number=""></all>	::=	"0"	["-"	'] <na< td=""><td>itural</td><td>numbei</td><td>c> .</td><td></td><td></td></na<>	itural	numbei	c> .		
<single-digit number=""></single-digit>	::=	"1"	"2"	"3"	"4"	"5"			
•		"6"	"7"	"8"	"9"	"0"			
<non number="" zero=""></non>	::=	"1"	"2"	"3"	"4"	"5"			
		"6"	"7"	"8"	"9" [']		,		
<blank></blank>	::=	я я							

A.2 Grammer

A.2.1 Architecture Type Section

```
<arch type>
                            ::=
                                 "arch type"
                                  " { "
                                     <cpu type>
                                     <pipeline>
                                     <max instruction bit>
                                     <max data bit>
                                  " ] "
<cpu type> := "cpu type { pipeline }"
                                "pipeline"
<pipeline>
                            :=
                                 "{"
                                    <stage number>
                                    <common stage number>
                                    <phase par stage>
                                    <decode stage>
                                    <stage name>
                                    <delayed slot number>
                                 "}"
                                "stages" "{" <natural number> "}"
"common stages" "{" "0" "}"
"phase par stage" "{" "1" "}"
<stage number>
                            :=
<common stage number>
                           :=
<phase par stage>
                           :=
                                "decode stage" "{" <natural number> "}"
<decode stage>
                           :=
<stage name>
                            :=
                                 "stage name"
                                 " { "
                                    <each stage name>
                                    { <each stage name> }
                                 " } "
                           := <natural number> "{"
<each stage name>
                                <alphabets and numbers> "}"
"delayed slot" "{" <nonnegative number> "}"
<delayed slot number> :=
<max instruction bit> := "max instruction bit" "{" <natural number> "}"
<max data bit> := "max data bit" "{" <natural number> "}"
```

\subsection{Input/Output Section}

\begin{verbatim}
<port_decralation> := "inputports" "{" <ports> "}"
 "outputports" "{" <ports> "}"
<ports> := [<port_name> { "," <port_name> }]

A.2.2 Resource Class Declaration

<resource arch>

A.2. GRAMMER

		<pre>{ <each class="" resource=""> } "}"</each></pre>
		u } n
<each cla<="" resource="" td=""><td>ss> ::=</td><td><class name=""> "{" <resource func=""> { <resource func=""> } "}"</resource></resource></class></td></each>	ss> ::=	<class name=""> "{" <resource func=""> { <resource func=""> } "}"</resource></resource></class>
<resource func=""></resource>	::=	<resource func="" name=""> "{"</resource>
<class name=""></class>	::=	<alphabets and="" numbers=""></alphabets>
<interface></interface>	:= "int "{" { { {	erface" each interface> <each interface=""> }</each>
<each interface=""></each>	:= <por< td=""><td>t name> "{" <operand> "}"</operand></td></por<>	t name> "{" <operand> "}"</operand>
<port name=""> <operand></operand></port>	:= <alp := <alp< td=""><td>habets and numbers> habets and numbers></td></alp<></alp 	habets and numbers> habets and numbers>
<exec time=""></exec>	:= "lat "{" "}" "thr "{" "}"	ency" latency> oughput" throughput>
<latency> <throughput></throughput></latency>	:= <nat := <nat< td=""><td>ural number> ural number></td></nat<></nat 	ural number> ural number>

A.2.3 Structure Definition

<structure arch=""></structure>	:=	<pre>"structure" "{" <each instance="" resource=""> { <each instance="" resource=""> } "}"</each></each></pre>
<pre><each instance="" resource=""></each></pre>	:=	<pre>["portion"] <instance name=""> "{"</instance></pre>

```
"}"
                           := <alphabets and numbers>
<instance name>
<resource class> := "class" "{" <alphabets and numbers> "}"
<stage> := "stage" "{" <natural number> "}" |
                         "multi stage"
                                   "{" <natural number> ".."
                                                                     "}"
                                        <natural number>
<output ports>
                         := "output port"
                                  " { "
                                     <each output port> { <each output port> }
                                  " } "
                        := <resource instance> "." <port name>
<each output port>
                                  "connection"
                                     { <each connected port> }
"}"
                                     " { "
<each connected port> := [ "stage" <natural number> ]
                                  <resource instance> "." <port name>
<resource instance> := <alphabets and numbers>
<port name> := <alphabets and numbers>
<port name>
```

A.2.4 Storage Definition

```
<storage arch>
                  ::= "storage"
                          " { "
                              <instance section>
                              <stack model>
                             <flag section>
                          "}"
<instance section>::= "instance"
                          " { "
                              <storage instance> { <storage instance> }
                          " ] "
<storage instance name> ::= <alphabets and numbers>
<resource instance name> ::= <alphabets and numbers>
                              ["[" <natural number
[ ".." <natural number> ] "]" ]
<storage instance> := <storage instance name>
                         " { "
                              <avail field>
                              <class field>
                              <resource field>
                              <number field>
                              <width field>
                              <cost field>
                             [ <type field> ]
                         " } "
```

A.2. GRAMMER

:= "avail" "{" "T" "}" "avail" "{" "F" "}" := "class" "{" <storage class> "}" <avail field> <class field> := "reg" | "I_mem" | "D_mem" | "pc" "zero" | "sp" | "fp" | "link" <storage class> <resource field> := "resource" " { " <resource instance name> { "&" <resource instance name> } "}" := "number" "{" <natural number> ")" := "width" "{" <natural number> "}" := "data_type" "{" <data type class> "}" <number field> <width field> <type field> <data type class> ::= "any" | "int" | "float" | "fixed" <stack model> := "stack" "{" <stack width> <stack alignment> <stack depth> "}" <stack width> := "width" "{" <natural number> "}"
<stack alignment> := "alignment" "{" <natural number> "}"
<stack depth> := "depth" "{" <natural number> "}" <flag section> ::= "condition flag" "{" { <each flag> } "}" <each flag> ::= "Neg_flag" <flag instance> ";" | "Zero_flag" <flag instance> ";" | "Carry_flag" <flag instance> ";" | "Overflow_flag" <flag instance> ";" | <flag instance> ::= <alphabets capital>

A.2.5 Instruction Definition

```
<instruction set file> ::= "instruction"
                                        " { "
                                            <each instruction>
                                          { <each instruction> }
                                        "}"
<each instruction>
                        ::= <inst name>
                                      " { "
                                          <inst operand>
                                          <inst format>
                                          <inst functions>
                                          <inst behavior>
                                     "}"
<inst name>
                       ::= <alphabets and numbers>
<inst functions>
                       ::= "functions"
                                     " { "
                                          <each stage>
                                        { <each stage> }
                                     " } "
```

::= "stage" "(" <stage number> ")" <each stage> " { " { <each function> } "}" ::= <natural number> <stage number> ::= <resource name> "." <function name> <each function> ["(" <parameter> { "," <parameter> } ")" 1 ";" ::= <alphabets and numbers>
::= <alphabets and numbers> <resource name> <function name> ::= "operand" <inst operand> "{" <each operand> { <each operand> } "}" <each operand> <addressing mode> <memory mode> <other mode> ::= <register direct> | <register mode> <register indirect> ::= <register class> <register direct> <register class> ::= <alphabets and numbers> ::= "[" <register class> "," "displacement" "]"
 ":" <memory class> <register indirect> ::= <alphabets and numbers> <memory class> ::= "@" <memory class>
 "@" "[" <memory class> "]" <memory mode> ::= "#" "Imm" <immediate size> <other mode> "#" "label" "#" "global" ::= <natural number> <immediate size> ::= "int" | "uint" | "float" |
 "fix" <fix spec>| "any" | <macro typedef>
 ::= "(" <nonnegative number> ".."
 <nonnegative number> ")" <data type> <fix spec> ::= <alphabets small>
::= <alphabets and numbers> <parameter> <macro typedef> ::= "format" <inst format> "{" <format element> { <format element> } "}" ::= """ <alphabets string> """ | <format element> <parameter> ::= "functions" <inst functions> " { " <each stage> { <each stage> } "}"

A.2. GRAMMER

```
::= "stage" "(" <stage number> ")"
"{"
<each stage>
                                         { <each function> }
                                       "}"
<stage number>
                         ::= <natural number>
<each function>
                         ::= <resource name> "." <function name> ";"
<resource name>
                         ::= <alphabets and numbers>
<function name>
                         ::= <alphabets and numbers>
<inst behavior>
                         ::= "behavior"
                                      " { "
                                           <normal operation>
                                           <control operation>
                                           <compare operation>
                                      "}"
                        ::= <expression> { <expression> }
::= [ <left term> "=" ] <right term> ";" [ <set flag> ]
<normal operation>
<expression>
<left term>
                        ::= <parameter>
                        "(" <parameter> { "," <parameter> "}" ")"
::= <term layer9> { "|" <term layer9> }
::= <term layer8> { "^" <term layer8> }
<right term>
<term layer9>
<term layer8>
                         ::= <term layer7> { "&" <term layer7> }
<term layer7>
                         ::= <term layer6>
                                { [ "==" | "!=" ] <term layer6> }
                         ::= <term layer5>
<term layer6>
                                { [ "<" | "<=" | ">" | ">=" } <term layer5> }
<term layer5>
                         ::= <term layer4>
                                { [ "<<" | ">>" | ">>>" ] <term layer4> }
                         ::= <term layer3>
{ [ "+" | "-" ] <term layer3> }
<term layer4>
<term layer3>
                         ::= <term layer2>
                        { [ "*" | "/" | "%" ] <term layer2> }
::= "~" <term layer1> |
<term layer2>
                              "Sign_extended" "(" <term layer1> ")"
                              "Next" "(" <term layer1> ")"
<term layer1>
                         ::= "(" <right expression> ")" [ "[" <field> ":"
                                                                  <field> "]"
                                                            1
                              "(" <right expression> ")" [ "[" <bit> "]" ]
                              <parameter> [ "[" <field> ":"
                                                  <field> "]"
                                           1
                              <parameter> [ "[" <bit> "]" ]
                              <constant>
<constant>
                        ::= <natural number>
<set flag>
                        ::= "{" <flag instance> { "," <flag instance> } "}"
                        ::= "If" <conditions>
<control operation>
                                             " { "
                                                 <normal operation>
                                             "}"
                                           [ "Else"
                                             " { "
                                                  <normal operation>
                                             "}"
                                           1
```

<conditions></conditions>	::=	"(" <normal conditions=""> ")" "(" <flag conditions=""> ")"</flag></normal>
<normal conditions=""></normal>	::=	<pre><parameter> <comparator sign=""> <parameter> </parameter></comparator></parameter></pre>
<comparator sign=""></comparator>	::=	"==" "!=" "<" "<=" ">" ">=" "always
<flag conditions=""></flag>	::=	<flag condition=""></flag>
-		<pre>{ <logical operator=""> <flag condition=""> }</flag></logical></pre>
<flag condition=""></flag>	::=	<flag instance=""> "==" <flag value=""></flag></flag>
<flag instance=""></flag>	::=	<alphabets capital=""></alphabets>
<flag value=""></flag>	::=	"0" "1"
<logical operator=""></logical>	::=	"&&" " " "
<compare operation=""></compare>	::=	"Compare" "(" <parameter> ","</parameter>
		<parameter> ")" ";" <set flag=""></set></parameter>

Appendix B

MIPS-R3000 Architecture Description for the Proposed Compiler Generator
```
1 arch type
2 {
3
    cpu type { pipeline }
 4
    pipeline
 5
     {
                       { 5 }
 6
      stage
                       { 0 }
7
       common stage
 8
      phase par stage { 1 }
9
       decode stage { 2 }
10
       stage name
11
       {
           1 { IF }
12
         2 { ID }
13
         3 { EXE }
14
         4 { MEM }
15
16
         5 { WB }
17
       }
                       { 0 }
18
       slot
19
     }
     max instruction bit { 32 }
20
21
    max data bit
                    { 32 }
22 }
23
24 resource class
25 {
26
     PC
27
     ſ
28
       port
29
       ł
         input { in1 }
30
         output { out1 }
31
32
       }
33
       function
34
       {
35
         read
36
         {
37
           interface
38
           {
39
            out1 { a }
40
           }
41
           latency
                     {1}
42
           throughput { 1 }
         }
43
44
         write
45
         {
46
           interface
47
           {
48
             in1 { a }
49
           }
50
           latency { 1 }
           throughput { 1 }
51
52
         }
53
         inc
54
         {
55
           interface
56
           {
             in1 { a }
57
```

```
58
            }
59
            latency { 1 }
            throughput { 1 }
60
 61
          }
62
        }
 63
      }
64
      IMEM
 65
      {
 66
        port
 67
        {
 68
          input { in1 }
          output { out1 }
 69
 70
        3
 71
        function
 72
        {
 73
          read
 74
          {
 75
            interface
 76
            {
 77
              in1 { a }
 78
              out1 { b }
 79
            }
            latency { 2 }
 80
 81
            throughput { 2 }
 82
          }
 83
        }
      }
 84
 85
      IR
 86
      {
 87
        port
 88
        {
          input { in1 }
 89
          output { out1 }
 90
 91
        }
 92
        function
 93
        {
 94
          read
 95
          {
 96
            interface
 97
            {
 98
              out1 { a }
 99
             }
100
            latency { 1 }
             throughput { 1 }
101
102
          }
103
          write
104
           {
105
            interface
106
             {
107
              in1 { a }
108
            }
109
            latency { 1 }
110
            throughput { 1 }
111
          }
112
        }
113
      }
114
      GPR
```

115 { 116 port 117 { input { in1 , in2 , 118 in3 , in4 } 119 output { out1 , out2 } 120 121 } 122 function 123 { 124 read0 125 { 126 interface 127 { 128 in1 { a } 129 out1 { b } 130 } 131 latency { 1 } 132 throughput { 1 } 133 } 134 read1 135 { 136 interface 137 { 138 in2 { a } out2 { b } 139 140 } 141 latency {1} 142 throughput { 1 } 143 } 144 write 145 { 146 interface 147 { 148 in3 { a } 149 in4 { b } 150 } 151 latency { 1 } 152 throughput { 1 } 153 } 154 } 155 } EXT 156 157 { 158 port 159 { 160 input { in1 } 161 output { out1 } 162 } 163 function 164 { 165 zero_ext 166 { 167 interface 168 { 169 in1 { a } 170 out1 { b } 171 }

latency { 1 } throughput { 1 } } sign { interface { in1 { a } out1 { b } } latency { 1 } throughput { 1 } } } } ADD { port { input { in1 , in2 } output { out1 } } function { add { interface { in1 { a } in2 { b } out1 { c } } latency { 1 } throughput { 1 } } adc { interface { in1 { a } in2 { b } out1 { c } } latency { 1 } throughput { 1 } } } } ALU { port { input { in1 , in2 } output { out1 } } function {

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301 302

303

304

305

306

307

308

309

310

311

312

313

314 315

316

317

318

319 320

321 322

323

324

325

326

327

328

329

330

331

332

333

334

335 336

337

338

339

340

341

342

229 addu 230 { 231 interface 232 { in1 { a } in2 { b } out1 { c } 233 234 235 236 } 237 latency {1} throughput { 1 } 238 } 239 add 240 241 { 242 interface 243 { in1 { a } in2 { b } 244 245 out1 { c } 246 247 } latency { 1 } 248 throughput { 1 } 249 250 } 251 subu 252 { interface 253 254 { in1 { a } in2 { b } 255 256 257 out1 { c } 258 } latency {1} 259 throughput { 1 } 260 261 } 262 sub 263 { 264 interface 265 { in1 { a } in2 { b } 266 267 out1 { c } 268 269 } latency { 1 } 270 throughput { 1 } 271 272 } 273 and 274 { 275 interface 276 { in1 { a } in2 { b } 277 278 out1 { c } 279 280 } latency { 1 } 281 282 throughput { 1 } 283 } 284 compu 285 {

interface { in1 { a } in2 { b } out1 { c } } latency {1} throughput { 1 } } comp { interface { in1 { a } in2 { b } out1 { c } } latency { 1 } throughput { 1 } } compzu { interface { in1 { a } out1 { b } } latency { 1 } throughput { 1 } } compz { interface { in1 { a } out1 { b } } latency { 1 } throughput { 1 } } nor { interface { in1 { a } in2 { b } out1 { c } } latency { 1 } throughput { 1 } } or { interface { in1 { a } in2 { b }

343 out1 { c } 344 } 345 latency { 1 } 346 throughput { 1 } 347 } 348 xor 349 { 350 interface 351 { 352 in1 { a } 353 in2 { b } 354 out1 { c } 355 } 356 latency {1} 357 throughput { 1 } 358 } 359 } 360 } 361 SFT 362 { 363 port 364 { 365 input { in1 , in2 } 366 output { out1 } 367 } 368 function 369 { 370 s11 371 { 372 interface 373 { in1 { a } in2 { b } 374 375 376 out1 { c } 377 } 378 latency { 1 } 379 throughput { 1 } 380 } 381 sra 382 { 383 interface 384 { in1 { a } in2 { b } 385 386 387 out1 { c } 388 } latency { 1 } 389 throughput { 1 } 390 391 } 392 srl393 { 394 interface 395 { 396 in1 { a } in2 { b } 397 out1 { c } 398 399 }

latency { 1 } throughput { 1 } } } } DMEM { port { input { in1 , in2 } output { out1 } 3 function { load { interface { in1 { a } out1 { b } } latency { 2 } throughput { 2 } } 1hu { interface { in1 { a } out1 { b } • } latency { 2 } throughput { 2 } } lh { interface { in1 { a } out1 { b } } latency { 2 } throughput { 2 } } 1bu { interface { in1 { a } out1 { b } 3 latency { 2 } throughput { 2 } } 1b { interface

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

457 { 458 in1 { a } out1 { b } 459 460 } latency { 2 } 461 throughput { 2 } 462 463 } 464 store 465 { 466 interface 467 { 468 in1 { a } in2 { b } 469 470 } 471 latency { 2 } throughput { 2 } 472 473 } 474 sh475 { 476 interface 477 { 478 in1 { a } 479 in2 { b } 480 } 481 latency { 2 } throughput { 2 } 482 483 } 484 sb 485 { 486 interface 487 { 488 in1 { a } 489 in2 { b } 490 } 491 latency { 2 } throughput { 2 } 492 493 } 494 } 495 } 496 NOT 497 { port 498 499 { input { in1 } 500 501 output { out1 } 502 } 503 function 504 { 505 not 506 { 507 interface 508 { 509 in1 { a } 510 out1 { b } 511 } 512 latency { 1 } 513 throughput { 1 } 514 } 515 } 516 } 517 CMAC 518 { · 519 port 520 { 521 input { in1 , in2 } output { out1 522 } 523 } 524 function 525 { 526 cmac 527 { 528 interface 529 { 530 in1 { a } 531 in2 { b } 532 } latency { 34 } 533 534 throughput { 34 } 535 } 536 clracc 537 { 538 interface 539 { 540 in1 { a } 541 } latency { 1 } 542 543 throughput { 1 } 544 } 545 readacc 546 { 547 interface 548 { 549 out1 { a } 550 } 551 latency { 1 } throughput { 1 } 552 553 } 554 ifracdigits 555 { interface 556 557 { 558 in1 { a } 559 } 560 latency { 1 } 561 throughput { 1 } 562 } 563 ofracdigits 564 { 565 interface 566 { 567 in1 { a } 568 569 latency {1} 570 throughput { 1 }

571 } 572 } 573 } 574 CMP 575 { 576 port 577 { 578 input { in1 , in2 } 579 output { out1 } 580 } 581 function 582 { 583 Cmp 584 { 585 interface 586 { 587 in1 { a } 588 in2 {b} 589 out1 { c } 590 } 591 latency { 1 } 592 throughput { 1 } 593 } 594 cmpz 595 { 596 interface 597 { 598 in1 { a } 599 in2 {b} 600 out1 { c } 601 3 602 latency { 1 } 603 throughput { 1 } 604 } 605 } 606 } 607 MUL 608 { 609 port 610 { 611 input { in1 , in2 } 612 output { out1 } 613 } 614 function 615 { 616 multiply_u 617 { 618 interface 619 { in1 { a } in2 { b } 620 621 622 out1 { c } 623 } 624 {1} latency 625 throughput { 1 } 626 } 627 multiply_s

{ interface { in1 { a } in2 { b } out1 { c } 3 latency {1} throughput { 1 } } } } ΗI { port { input { in1 } output { out1 } } function { direct_read { interface { out1 { a } } latency {1} throughput { 1 } } direct_write { interface { in1 { a } } latency { 1 } throughput { 1 } } } } \mathbf{LO} { port { input { in1 } output { out1 } } function { direct_read { interface { out1 { a } } latency { 1 }

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

685 throughput { 1 } 686 3 direct_write 687 688 { 689 interface 690 { 691 in1 { a } 692 } latency { 1 } 693 throughput { 1 } 694 695 } 696 } 697 } 698 699 DIV 700 { 701 port 702 { 703 input { in1 , in2 } output { out1 , out2 } 704 705 } function 706 707 { 708 divide_u 709 { 710 interface 711 { in1 { a } in2 { b } 712 713 out1 { c } 714 715 out2 { d } } 716 latency { 1 } 717 throughput { 1 } 718 719 } divide_s 720 721 { interface 722 723 { in1 { a } in2 { b } 724 725 out1 { c } 726 out2 { d } 727 728 } latency { 1 } 729 throughput { 1 } 730 731 } 732 } 733 } 734 735 } 736 737 storage 738 { 739 instance 740 { 741 PC

742 { 743	avail	{	F }	
744	class	{	pc }	
745	resource	{	PC }	
746	number	ł	1 }	
747	width	ł	34 }	
748	data type	٤	any }	
749 }	D			
750 5	P			
/5⊥ 1 750	arra i l	r	τ Γ	3
752	class	τ τ	sn	ş
754	resource	{	GPR[29]	ì
755	number	ł	1	ĵ
756	width	{	32	}
757	data type	{	any	}
758 }				
759 F	'P			
760 {				
761	avail	{	т	}
762	class	{	fp	}
763	resource	{	GPR[30]	}
764	number	{	1	}
765	width	{	32	}
766	data type	{	any	}
767)				
768 I	JINK			
769			_	,
770	avail	ł	T	}
771	class	{		} l
772	resource	۱ ۲	GFR[JI]	ז ו
113	number	ı r	⊥ 30	1
775	data type	ι {	anv	}
776		ι	uny	1.
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	ZERO			
778	(
779	avail	{	F	}
780	class	{	zero	}
781	resource	{	GPR[0]	}
782	number	{	1	}
783	width	{	32	}
784	data type	{	any	}
785	}			
786	RETURN			
787	{			
788	avail	{	Т	}
789	class	{	return	}
790	resource	{	GPR[28]	}
791	number	_{	1,	3
792	width { 3	4,	} 	
793	data type	1	any j	
/94	้ เวพ ะพ			
795	l seren			
797	avail	ſ	т	ł
798	class	{	D_mem	}

799	resource { DMEM }
800	number { 1 }
801	width { 32 }
802	<pre>data type { any }</pre>
803	}
804	IMEM
805	{
806	avail {F}}
807	class { I_mem }
808	resource { IMEM }
809	number { 1 }
810	width { 32 }
811	data type { any }
812	}
813	IR
814	{
815	avail {F}
816	class { reg }
817	resource { IR }
818	number {1}
819	width { 32 }
820	data type { any }
821	}
822	GPR
823	{
824	avail {T}
825	class { reg }
826	resource { GPR }
827	number { 32 }
828	width {32}
829	data type { any }
030	}
832	f f
833	เ อหอ่าไ (กา ไ
834	
835	resource { CMACA }
836	number { 1 }
837	width $\{40\}$
838	data type { any }
839	}
840	HI
841	{
842	avail {T}
843	class { reg }
844	resource { HI }
845	number {1}
846	width { 32 }
847	data type { any }
848	}
849	LO
850	{
851	avail { T }
852	class { reg }
853	resource { LO }
854	number { 1 }
855	width { 32 }

856	<pre>data type { any }</pre>	
857	}	
858	HL	
859	{	
860	avail { T }	
861	class { reg }	
862	resource { HI&LO	}
863	number $\{1\}$	1
864	width $\{64\}$	
865	data type { any }	
866	}	
867	}	
868	stack	
869	{	
870	width { 16 }	
871	depth $\{200\}$	
872	}	
873	condition flag	
874	{	
875	}	
876	}	
877	ſ	
878	instruction	
879	{	
880	source data type spec	
881	{	
882	char	
883	{	
884	alignment (8)	
885	size (8)	
886	}	
887	short	
888	{	
889	alignment { 16 }	
890	size $\{16\}$	
891	}	
892	short2	
893	{	
894	alignment { 16 }	
895	size { 16 }	
896	}	
897	int	
898	{	
899	alignment { 32 }	
900	size { 32 }	
901	}	
902	long	
903	{	
904	alignment { 32 }	
905	size { 32 }	
906	}	
907	long2	
908	{	
909	alignment { 64 }	
910	size { 64 }	
911	}	•
912	float	

913 { alignment { 32 } 914 915 size { 32 } 916 } 917 double 918 { alignment { 64 } 919 920 size { 64 } 921 } 922 quad 923 { alignment { 64 } 924 925 size { 64 } 926 } 927 point 928 { 929 alignment { 32 } size { 32 } 930 931 } 932 struct 933 { alignment { 8 } 934 935 3 936 data 937 { 938 alignment { 8 } 939 } 940 } 941 942 943 struct declaration 944 { 945 struct man { 946 char person_name[20]; 947 int age; 948 }; 949 struct complex { 950 int real; 951 int imaginary; 952 }; } 953 954 955 956 macro typedef 957 { Int7to0 958 959 { 960 signed unsigned { char } 961 } 962 SInt7to0 963 964 { 965 signed { char } 966 } 967 UInt7to0 968 969 {

unsigned { char } 970 971 } 972 973 Int15to0 974 { signed unsigned { char short } 975 976 } 977 SInt15to0 978 979 { signed { char short } 980 981 } 982 UInt15to0 983 984 { 985 unsigned { char short } 986 } 987 Int31to0 988 989 { signed unsigned { char short 990 991 int long } 992 } 993 SInt31to0 994 995 { signed { char short int long } 996 997 } 998 999 UInt31to0 1000 { unsigned { char short int long } 1001 1002 3 1003 } 1004 1005 1006 ckf prototype 1007 { complexMAC 1008 void (unsigned int , unsigned int); 1009 unsigned int loadAcc (); 1010 accumClear (); void 1011 setCpos (int); setRpos (int); 1012 void void 1013 madd1(int, int); madd2(int, int); 1014 short short void 1015 blockadd(int, int); 1016 1017 } 1018 1019 1020 1021 ADD 1022 ſ operand 1023 1024 { GPR SInt31to0 a; 1025 GPR SInt31to0 b; 1026

1027	GPR SInt31to0 c;	1084	stage(2)
1028	}	1085	f
1029	format	1086	GPR road0
1030	{	1087	EXTRO at an
1031	"ADD" a ", " b ", " c	1088	
1032	}	1000	/ stars(2)
1033	functions	1009	stage(3)
1034	{	1090	
1035	stage(1)	1091	ALUU.add
1036		1092	}
1037	PC road	1093	stage(4)
1038	IC. Ieau IMEM road	1094	{}
1039	PC inc	1095	stage(5)
1040	IP road	1096	{
1040	IK.Ieau	1097	GPR.write
1042	s (2)	1098	}
1042	stage(2)	1099	}
1043		1100	behavior
1044	GPR.readU	1101	{
1045	GPR.read1	1102	a = b + c;
1040	}	1103	}
1047	stage(3)	1104	·}
1048	{	1105	ADDIU
1049	ALU0.add	1106	{
1050	}	1107	operand
1051	stage(4)	1108	{
1052	{}	1109	GPR UInt31to0 a;
1053	<pre>stage(5)</pre>	1110	GPR UInt31to0 b;
1054	{	1111	'Imm 16 UInt15to0 c;
1055	GPR.write	1112	}
1056	}	1113	format
1057	}	1114	{
1058	behavior	1115	"ADDTU" a " " b " " c
1059	{	1116	}
1060	a = b + c;	1117	functions
1061	}	1118	{
1062	}	1119	stage(1)
1063	ADDI	1120	{
1064	{	1121	PC read
1065	operand	1122	IMEM read
1066	{	1123	PC inc
1067	GPR SInt31to0 a;	1124	TR read
1068	GPR SInt31to0 b;	1125)
1069	'Imm 16 SInt15to0 C:	1126	stage(2)
1070	}	1127	stuge (z)
1071	format	1120	
1072	{	1120	GPR. readu
1073	"ADDT" a " " b " " a	1120	EXTU.Sign
1074	}	1121	}
1075	functions	1120	stage(3)
1076	{	1132	{
1077	stage(1)	1133	ALU0.add
1078	scage(1)	1134	}
1079	PC road	1135	stage(4)
1080	rt.reau	1136	{}
1000	IMAM. read	1137	stage(5)
1001	PU.inc	1138	{
1002	1K.read	1139	GPR.write
1083	}	1140	}

1141 } 1142 behavior 1143 { a = b + c;1144 1145 } 1146 } 1147 ADDU 1148 { 1149 operand 1150 { GPR UInt31to0 a; 1151 GPR UInt31to0 b; 1152 GPR UInt31to0 c; 1153 1154 } 1155 format 1156 { "ADDU" a ", " b ", " c 1157 1158 } functions 1159 1160 { stage(1) 1161 1162 { PC.read 1163 IMEM.read 1164 1165 PC.inc IR.read 1166 1167 } 1168 stage(2) 1169 { 1170 GPR.read0 GPR.read1 1171 1172 3 1173 stage(3) 1174 ſ ALU0.add 1175 1176 } 1177 stage(4) 1178 {} stage(5) 1179 1180 { GPR.write 1181 } 1182 1183 } 1184 behavior 1185 { a = b + c;1186 1187 } 1188 } 1189 AND 1190 { 1191 operand 1192 { 1193 GPR any a; GPR any b; 1194 1195 GPR any c; 1196 } 1197 format

1198 { "AND" a ", " b ", " c 1199 1200 } 1201 functions 1202 { 1203 stage(1) 1204 { 1205 PC.read IMEM.read 1206 1207 PC.inc TR.read 1208 1209 } 1210 stage(2) 1211 { GPR.read0 1212 GPR.read1 1213 1214 } 1215 stage(3) 1216 { 1217 ALU0.and 1218 } 1219 stage(4) 1220 {} stage(5) 1221 1222 { GPR.write 1223 1224 } 1225 3 1226 behavior 1227 { a = b & c; 1228 1229 3 1230 } ANDT 1231 1232 { operand 1233 1234 { 1235 GPR any a; GPR any b; 1236 'Imm 16 any c; 1237 1238 } format 1239 1240 { "ANDI" a ", " b ", " c 1241 1242 } functions 1243 1244 { 1245 stage(1) 1246 { PC.read 1247 1248 IMEM.read PC.inc 1249 1250 IR.read 1251 } stage(2) 1252 1253 { GPR.read0 1254

1255 EXT0.zero_ext 1256 } 1257 stage(3) 1258 { 1259 ALU0.and 1260 3 1261 stage(4) 1262 {} 1263 stage(5) 1264 { 1265 GPR.write 1266 ł 1267 } 1268 behavior 1269 { 1270 a = b & c; 1271 } 1272 } 1273 BEQ 1274 { 1275 operand 1276 { 1277 GPR any a; 1278 GPR any b; 1279 'label any c; 1280 PC any d; 1281 } 1282 format 1283 { 1284 "BEQ" a ", " b ", " c 1285 } 1286 functions 1287 { 1288 stage(1) 1289 ſ 1290 PC.read 1291 IMEM.read 1292 PC.inc 1293 IR.read 1294 } 1295 stage(2) 1296 { 1297 GPR.read0 1298 GPR.read1 1299 EXT0.sign 1300 } 1301 stage(3) 1302 ł 1303 PC.read 1304 ADD0.add 1305 CMP0.cmp 1306 PC.write 1307 } 1308 stage(4) 1309 {} 1310 stage(5) 1311 {}

} behavior { if(a == b)ſ d = c;1318 } } } BGEZ { operand { GPR SInt31to0 a; 'label any b; PC any C; } format { "BGEZ" a ", " b } functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) { GPR.read0 EXT0.sign } stage(3) { PC.read ADD0.add CMP0.cmpz PC.write } stage(4) {} stage(5) {} } behavior { if $(a \ge 0)$ { c = b: } } } BGEZAL

{

1312

1313

1314

1315

1316

1317

1319

1320

1321

1322

1323

1324

1325

1326

1327

1328

1329

1330

1331

1332

1333

1334

1335

1336

1337

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

1361

1362

1363

1364

1365

1366

1367

1426

1427

1428

1429

1430

1431

1432

1433

1434

1435

1436

1437

1438

1439

1441

1369 operand 1370 { 1371 GPR SInt31to0 a; 1372 'label any b; 1373 PC any c; 1374 d; LINK any 1375 } 1376 format 1377 { "BGEZAL" a ", " b 1378 1379 } 1380 functions 1381 { 1382 stage(1) 1383 { 1384 PC.read 1385 IMEM.read PC.inc 1386 1387 IR.read 1388 } 1389 stage(2) 1390 { GPR.read0 1391 EXT0.sign 1392 1393 3 1394 stage(3) 1395 { PC.read 1396 1397 ADD0.add CMP0.cmpz 1398 1399 PC.write 1400 } 1401 stage(4) 1402 { } stage(5) 1403 1404 { GPR.write 1405 1406 } 1407 3 1408 behavior 1409 { 1410 if $(a \ge 0)$ 1411 { d = Next(c);1412 1413 c = b;1414 } 1415 } 1416 } 1417 BGTZ 1418 { 1419 operand 1420 ſ SInt31to0 a; GPR 1421 1422 'label any b; any c; 1423 \mathbf{PC} 1424 3 1425 format

{ "BGTZ" a ", " b } functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) { 1440 GPR.read0 EXT0.sign 1442 } 1443 stage(3) 1444 { 1445 PC.read ADD0.add 1446 1447 CMP0.cmpz PC.write 1448 1449 } 1450 stage(4) 1451 $\{ \}$ stage(5) 1452 1453 {} 1454 3 1455 behavior 1456 { if (a > 0)1457 1458 {· c = b;1459 1460 } 1461 } 1462 3 1463 BLEZ 1464 { 1465 operand 1466 { GPR SInt31to0 a; 1467 1468 'label any b; any c; PC 1469 1470 } 1471 format 1472 { "BLEZ" a ", " b 1473 1474 } 1475 functions 1476 { 1477 stage(1) 1478 { 1479 PC.read IMEM.read 1480 PC.inc 1481 IR.read 1482

1483 } 1484 stage(2) 1485 { 1486 GPR.read0 1487 EXT0.sign 1488 } 1489 stage(3) 1490 { 1491 PC.read 1492 ADD0.add 1493 CMP0.cmpz 1494 PC.write 1495 } 1496 stage(4) 1497 {} 1498 stage(5) 1499 {} 1500 } 1501 behavior 1502 { 1503 if(a <= 0)1504 { 1505 c = b;1506 } 1507 } 1508 } 1509 BLTZ 1510 { 1511 operand 1512 { 1513 GPR SInt31to0 a; 1514 'label any b; 1515 \mathbf{PC} any c; 1516 } 1517 format 1518 { 1519 "BLTZ" a ", " b 1520 } 1521 functions 1522 { 1523 stage(1) 1524 { 1525 PC.read 1526 IMEM.read 1527 PC.inc 1528 IR.read 1529 } 1530 stage(2) 1531 { 1532 GPR.read0 1533 EXT0.sign 1534 } 1535 siage(3) 1536 { 1537 PC.read 1538 ADD0.add 1539 CMP0.cmpz

PC.write } stage(4) {} stage(5) {} } behavior { if (a < 0){ c = b;} } } BLTZAL { operand { GPR SInt31to0 a; 'label any b; \mathbf{PC} any с; LINK d; any } format { "BLTZAL" a ", " b } functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) ſ GPR.read0 EXT0.sign } stage(3) { PC.read ADD0.add CMP0.cmpz PC.write } stage(4) {} stage(5) { GPR.write } } behavior

1540

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

1551

1552

1553

1554

1555

1556

1557

1558

1559

1560

1561

1562

1563

1564

1565

1566

1567

1568

1569

1570

1571

1572

1573

1574

1575

1576

1577

1578

1579

1580

1581

1582

1583

1584

1585

1586

1587

1588

1589

1590

1591

1592

1593

1594

1595

1597 { 1598 if(a < 0)1599 { 1600 d = Next(c);1601 c = b;1602 } 1603 } 1604 } 1605 BNE 1606 { 1607 operand 1608 { 1609 GPR 1610 GPR 'label any c; 1611 PC 1612 1613 } 1614 format 1615 { "BNE" a ", " b ", " c 1616 1617 } 1618 functions 1619 { 1620 stage(1) 1621 { PC.read 1622 1623 IMEM.read 1624 PC.inc 1625 IR.read 1626 } stage(2) 1627 1628 { GPR.read0 1629 GPR.read1 1630 1631 EXT0.sign 1632 } stage(3) 1633 1634 { PC.read 1635 1636 ADD0.add ALU0.cmp 1637 PC.write 1638 1639 } stage(4) 1640 1641 {} stage(5) 1642 1643 {} 1644 } 1645 behavior 1646 { if (a != b) 1647 1648 { 1649 d = c;1650 3 1651 } 1652 }

any a;

any b;

any d;

1653

J

1654 { 1655 operand 1656 { 1657 'label any a; any b; 1658 PC 1659 } 1660 format 1661 { "J" a 1662 1663 } 1664 functions 1665 { 1666 stage(1) 1667 { PC.read 1668 IMEM.read 1669 PC.inc 1670 IR.read 1671 1672 } 1673 stage(2) 1674 {} 1675 stage(3) 1676 { 1677 PC.write 1678 } 1679 stage(4) 1680 {} stage(5) 1681 1682 {} 1683 } 1684 behavior 1685 { if (always) 1686 1687 { 1688 b = a;1689 } 1690 } 1691 3 1692 JALR 1693 { operand 1694 1695 { GPR any a; 1696 PC any b; 1697 LINK any c; 1698 1699 } 1700 format 1701 { "JALR" a 1702 1703 } functions 1704 1705 { stage(1) 1706 1707 { 1708 PC.read IMEM.read 1709 1710 PC.inc

1711 IR.read 1712 } 1713 stage(2) 1714 { 1715 GPR.read0 1716 } 1717 stage(3) 1718 { 1719 PC.write 1720 } 1721 stage(4) 1722 {} 1723 stage(5) 1724 { 1725 GPR.write 1726 } 1727 3 1728 behavior 1729 { 1730 if (always) 1731 { 1732 c = Next(b);1733 b = a;1734 } 1735 } 1736 } 1737 JR 1738 { 1739 operand 1740 { 1741 GPR any a; 1742 PC any b; 1743 } 1744 format 1745 { 1746 "JR" a 1747 } 1748 functions 1749 { 1750 stage(1) 1751 { 1752 PC.read 1753 IMEM.read 1754 PC.inc 1755 IR.read 1756 } 1757 stage(2) 1758 { 1759 GPR.read0 1760 } 1761 stage(3) 1762 { 1763 PC.write 1764 } 1765 stage(4) 1766 {} 1767 stage(5)

1768 {} 1769 3 1770 behavior 1771 ł 1772 if (always) 1773 { 1774 b = a; 1775 } 1776 } 1777 } 1778 LB1779 { 1780 operand 1781 { 1782 GPR SInt7to0 a; 1783 [GPR, disp]:DMEM SInt7to0 b; 1784 3 1785 format 1786 { 1787 "LB" a ", " b 1788 } 1789 functions 1790 { 1791 stage(1) 1792 { 1793 PC.read 1794 IMEM.read 1795 PC.inc 1796 IR.read 1797 3 1798 stage(2) 1799 { 1800 GPR.read0 1801 EXT0.sign 1802 3 1803 stage(3) 1804 { 1805 ALU0.add 1806 } 1807 stage(4) 1808 { 1809 DMEM.1b 1810 } 1811 stage(5) 1812 { 1813 GPR.write 1814 } 1815 1 1816 behavior 1817 { 1818 a = *b[7:0];1819 } 1820 } 1821 LBU 1822 Ł 1823 operand 1824 {

1025	GPR	UInt7to0	a:	1882	IR.read
1925	[CPR disp]:DMEM	UInt7to0	b;	1883	}
1927)	01110.000	,	1884	stage(2)
1027	format			1885	{
1020	f f			1886	GPR.read0
1029	ו אד פוזיז מי שי " b			1887	EXT0.sign
1830	TBO a , D			1888	}
1030	} formaticana			1889	stage(3)
1832	functions			1890	{
1833	{			1891	ALUO, add
1834	stage(1)			1892	}
1835	{			1993	stage(4)
1836	PC.read			1993	
1837	IMEM.read			1005	TMEM 1h
1838	PC.inc			1095	
1839	IR.read			1896	J atago (5)
1840	}			1897	stage (J)
1841	stage(2)			1898	
1842	{			1899	GPR.WIICe
1843	GPR.read0			1900	}
1844	EXT0.sign			1901	}
1845	}			1902	behavior
1846	stage(3)			1903	{
1847	{			1904	a = *b[15:0];
1848	ALU0.add			1905	}
1849	}			1906	}
1850	stage(4)			1907	LHU
1851	{			1908	{
1852	DMEM.1bu			1909	operand
1853	} .			1910	{
1854	stage(5)			1911	GPR UInt15to0 a;
1855	{			1912	[GPR, disp]:DMEM UInt15to0 b;
1856	GPR.write			1913	}
1857	}			1914	format
1858	}			1915	{
1859	behavior			1916	"LHU" a ", " b
1860	{			1917	}
1861	a = *b[7:0];			1918	functions
1862	3			1919	{
1863	}			1920	stage(1)
1864	, LH			1921	{
1865	[1922	PC.read
1866	operand			1923	IMEM.read
1967	1			1924	PC.inc
1969	CPR	SInt15to)0 a	; 1925	IR.read
1060	ICPR displ:DMEM	I SInt15to	50 E	: 1926	}
1070	[GFR, GISP].DHL	Dincipee		1927	stage(2)
1071	format			1928	1
1871	format (1929	GPR.read0
1872	1. 			1930	EXT0.sign
18/3	"LH" a ", " D			1931	}
1874	}			1932	stage(3)
1875	functions			1077	{
1876	{			103/	ALUO add
1877	stage(1)			1025	}
1878	{			1026	stage(4)
1879	PC.read			1027	Staye (=)
1880	IMEM.read			1020	L DMEM 1 hu
				1948	1701Fa71 - 111C

1939 } 1940 stage(5) 1941 { 1942 GPR.write 1943 } 1944 } 1945 behavior 1946 { 1947 a = *b[15:0];1948 } 1949 } 1950 LUI 1951 { 1952 operand 1953 { 1954 GPR anv a: 1955 'Imm 16 Int15to0 b; 1956 } 1957 format 1958 { 1959 "LUI" a ", " b 1960 } 1961 functions 1962 { 1963 stage(1) 1964 { 1965 PC.read 1966 IMEM.read 1967 PC.inc 1968 IR.read 1969 } 1970 stage(2) 1971 {} 1972 stage(3) 1973 {} 1974 stage(4) 1975 {} 1976 stage(5) 1977 { 1978 GPR.write 1979 } 1980 } 1981 behavior 1982 { 1983 a = b << 16;1984 1 1985 } 1986 LW 1987 { 1988 operand 1989 { 1990 GPR any a; 1991 [GPR, disp]:DMEM any b; 1992 } 1993 format 1994 { 1995 "LW" a ", " b

} functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) { GPR.read0 EXT0.sign } stage(3) { ALU0.add 3 stage(4) { DMEM.read } stage(5) { GPR.write } } behavior { a = *b; } } NOR { operand { GPR any a; GPR any b; GPR any c; } format { "NOR" a ", " b ", " c } functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) ſ GPR.read0

1996

1997

1998

1999

2000

2001

2002

2003

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

2021

2022

2023

2024

2025

2026

2027

2028

2029

2030

2031

2032

2033

2034

2035

2036

2037

2038

2039

2040

2041

2042

2043

2044

2045

2046

2047

2048

2049

2050

2051

2053 GPR.read1 2054 } 2055 stage(3) 2056 { 2057 ALU0.nor 2058 1 2059 stage(4) 2060 {} 2061 stage(5) 2062 { GPR.write 2063 2064 } 2065 } 2066 behavior 2067 { 2068 a = (b | c);2069 } 2070 } 2071 OR 2072 { 2073 operand 2074 { 2075 GPR any a; GPR any b; 2076 2077 GPR any c; 2078 } 2079 format 2080 { "OR" a ", " b ", " c 2081 2082 } 2083 functions 2084 { 2085 stage(1) 2086 { 2087 PC.read IMEM.read 2088 2089 PC.inc 2090 IR.read 2091 } 2092 stage(2) 2093 ſ GPR.read0 2094 2095 GPR.read1 2096 } 2097 stage(3) 2098 { 2099 ALU0.or 2100 } 2101 stage(4) 2102 {} 2103 stage(5) 2104 { 2105 GPR.write 2106 } 2107 } 2108 behavior 2109 {

2110 a = b | c ; } 2111 2112 } 2113 ORI 2114 { 2115 operand 2116 { 2117 GPR any a; GPR any b; 2118 2119 'Imm 16 any c; 2120 .} 2121 format 2122 { "ORI" a ", " b ", " c 2123 2124 } 2125 functions 2126 { 2127 stage(1) 2128 { 2129 PC.read IMEM.read 2130 PC.inc 2131 2132 IR.read 2133 } 2134 stage(2) 2135 { GPR.read0 2136 2137 EXT0.zero_ext 2138 } 2139 stage(3) 2140 ſ ALU0.or 2141 2142 } 2143 stage(4) 2144 {} 2145 stage(5) 2146 { 2147 GPR.write 2148 } 2149 } 2150 behavior 2151 { a = b | c ;2152 2153 } 2154 } 2155 SB2156 { 2157 operand 2158 { any a; GPR 2159 2160 [GPR, disp]:DMEM any b; 2161 } 2162 format 2163 { "SB" a ", " b 2164 2165 } 2166 functions

2167 { 2168 stage(1) 2169 ſ 2170 PC.read 2171 IMEM.read 2172 PC.inc 2173 IR.read 2174 } 2175 stage(2) 2176 { 2177 GPR.read0 2178 GPR.read1 2179 EXT0.sign 2180 } 2181 stage(3) 2182 { 2183 ALU0.add 2184 } 2185 stage(4) 2186 { 2187 DMEM.sb 2188 } 2189 stage(5) 2190 {} 2191 } 2192 behavior 2193 { 2194 *b = a[7:0];2195 } 2196 } 2197 SH2198 { 2199 operand 2200 { 2201 GPR any a; 2202 [GPR, disp]:DMEM any b; 2203 } 2204 format 2205 { 2206 "SH" a ", " b 2207 } 2208 functions 2209 { 2210 stage(1) 2211 { 2212 PC.read 2213 IMEM.read 2214 PC.inc 2215 IR.read 2216 } 2217 stage(2) 2218 { 2219 GPR.read0 2220 GPR.read1 2221 EXT0.sign 2222 } 2223 stage(3)

{ ALU0.add } stage(4) { DMEM.sh 3 stage(5) {} 3 behavior { *b = a[15:0];} } SLL { operand { GPR any a; GPR any b; 'Imm 5 any c; } format { "SLL" a ", " b ", " c } functions { stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) { GPR.read0 } stage(3) { SFT0.sll } stage(4) {} stage(5) ſ GPR.write } 3 behavior ſ a = b << c ; 3 3

SLLV

2224

2225

2226

2227

2228

2229

2230

2231

2232

2233

2234

2235

2236

2237

2238

2239

2240

2241

2242

2243

2244

2245

2246

2247

2248

2249

2250

2251

2252

2253

2254

2255

2256

2257

2258

2259

2260

2261

2262

2263

2264

2265

2266

2267

2268

2269

2270

2271

2272

2273

2274

2275

2276

2277

2278

2279

2338

2339

2340

2341

2342

2343

2344

2345

2346

2347

2348

2349

2350

2351

2352

2353

2354

2355

2356

2357

2358

2359

2360

2361

2362

2363

2364

2365

2366

2367

2368

2369

2370

2371

2372

2373

2374

2375

2376

2377

2378

2379

2380

2281 { 2282 operand 2283 ſ 2284 GPR any a; GPR any b; 2285 2286 GPR any C; 2287 } 2288 format 2289 { "SLLV" a ", " b ", " c 2290 2291 } 2292 functions 2293 { 2294 stage(1) 2295 { PC.read 2296 2297 IMEM.read PC.inc 2298 2299 IR.read 2300 } 2301 stage(2) 2302 { GPR.read0 2303 2304 GPR.read1 2305 } stage(3) 2306 2307 { SFT0.sll 2308 2309 } 2310 stage(4) 2311 { } 2312 stage(5) 2313 { GPR.write 2314 2315 } 2316 3 2317 behavior 2318 { a = b << c ;2319 2320 } 2321 } 2322 SLT 2323 { operand 2324 2325 { GPR any a; 2326 GPR SInt31to0 b; 2327 2328 GPR SInt31to0 c; 2329 } 2330 format 2331 { "SLT" a ", " b ", " c 2332 2333 } 2334 functions

2335

2336

2337

{

{

stage(1)

PC.read IMEM.read PC.inc IR.read } stage(2) { GPR.read0 GPR.read1 } stage(3) { ALU0.cmp } stage(4) {} stage(5) { GPR.write } } behavior { a = b < c;} } SLTI { operand { GPR any a; GPR SInt31to0 b; 'Imm 16 SInt15to0 c; } format { "SLTI" a ", " b ", " c } functions { stage(1) { PC.read IMEM.read 2381 PC.inc 2382 IR.read 2383 2384 } 2385 stage(2) 2386 { 2387 GPR.read0 2388 EXT0.sign 2389 } stage(3) 2390 2391 ł ALU0.cmp 2392 2393 } 2394 stage(4)

2395 {} 2396 stage(5) 2397 { 2398 GPR.write 2399 } 2400 } 2401 behavior 2402 { 2403 a = b < c;2404 } 2405 } 2406 SLTIU 2407 { 2408 operand 2409 { 2410 GPR any a; GPR UInt31to0 b; 2411 2412 'Imm 16 UInt15to0 c; 2413 } 2414 format 2415 { 2416 "SLTIU" a ", " b ", " c 2417 3 2418 functions 2419 { 2420 stage(1) 2421 { 2422 PC.read 2423 IMEM.read PC.inc 2424 2425 IR.read 2426 } 2427 stage(2) 2428 { 2429 GPR.read0 2430 EXT0.extend 2431 } 2432 stage(3) 2433 ł 2434 ALU0.cmp 2435 } 2436 stage(4) 2437 {} 2438 stage(5) 2439 { 2440 GPR.write 2441 } 2442 } 2443 behavior 2444 { 2445 a = b < c; 2446 } 2447 } 2448 SLTU 2449 { 2450 operand 2451 {

GPR any a; GPR UInt31to0 b; GPR UInt31to0 c; } format { "SLTU" a ", " b ", " c } functions { stage(1) { PC.read IMEM.read PC.inc IR.read 3 stage(2) ſ GPR.read0 GPR.read1 1 stage(3) { ALU0.cmp } stage(4) {} stage(5) { GPR.write } } behavior { a = b < c; } } SRA { operand { GPR SInt31to0 a; GPR SInt31to0 b; 'Imm 5 any c; } format { "SRA" a ", " b ", " c } functions ſ stage(1) { PC.read IMEM.read PC.inc

2452

2453

2454

2455

2456

2457

2458

2459

2460

2461

2462

2463

2464

2465

2466

2467

2468

2469

2470

2471

2472

2473

2474

2475

2476

2477

2478

2479

2480

2481

2482

2483

2484

2485

2486

2487

2488

2489

2490

2491

2492

2493

2494

2495

2496

2497

2498

2499

2500

2501

2502

2503

2504

2505

2506

2507

2509 IR.read 2510 } 2511 stage(2) 2512 { 2513 GPR.read0 2514 } 2515 stage(3) 2516 { SFT0.sra 2517 2518 } 2519 stage(4) 2520 {} stage(5) 2521 2522 { 2523 GPR.write 2524 } 2525 } 2526 behavior 2527 { 2528 a = b >> c ; 2529 } 2530 } 2531 SRAV 2532 { 2533 operand 2534 { GPR SInt31to0 a; 2535 GPR SInt31to0 b; 2536 2537 GPR any c; 2538 } 2539 format 2540 { 2541 "SRAV" a ", " b ", " c 2542 1 2543 functions 2544 { 2545 stage(1) 2546 { PC.read 2547 2548 IMEM.read PC.inc 2549 2550 IR.read 2551 } 2552 stage(2) 2553 { 2554 GPR.read0 GPR.read1 2555 2556 } 2557 stage(3) 2558 { 2559 SFT0.sra 2560 } 2561 stage(4) 2562 { } stage(5) 2563 2564 { GPR.write 2565

2566 } 2567 3 2568 behavior 2569 { a = b >> c ; 2570 2571 } 2572 } 2573 SRL 2574 { 2575 operand 2576 { GPR UInt31to0 a; GPR UInt31to0 b; 2577 2578 2579 'Imm 5 any c; 2580 } 2581 format 2582 { "SRL" a ", " b ", " c 2583 2584 } 2585 functions 2586 ſ 2587 stage(1) 2588 { PC.read 2589 IMEM.read 2590 2591 PC.inc IR.read 2592 2593 } 2594 stage(2) 2595 { 2596 GPR.read0 2597 } 2598 stage(3) 2599 { SFT0.srl 2600 2601 } 2602 stage(4) 2603 {} 2604 stage(5) 2605 { 2606 GPR.write 2607 } 2608 3 2609 behavior 2610 { 2611 a = b >>> c ; 2612 } 2613 } 2614 SRLV 2615 { 2616 operand 2617 { GPR UInt31to0 a; 2618 2619 GPR UInt31to0 b; GPR any c; 2620 2621 } 2622 format

2623	{	2680	GPR.read1
2624	"SRLV" a ", " b ", " c	2681	}
2625	}	2682	stage(3)
2626	functions	2683	{
2627	{	2684	ALU0.sub
2628	stage(1)	2685	}
2629	{	2686	stage(4)
2630	PC.read	2687	{}
2631	IMEM.read	2688	stage(5)
2632	PC.inc	2689	{
2633	IR.read	2690	GPR.write
2634	}	2691	}
2635	stage(2)	2692	}
2636	{	2693	behavior
2637	GPR.read0	2694	{
2638	GPR.read1	2695	a = b - c:
2639	}	2696	}
2640	stage(3)	2697	}
2641	{	2698	SUBU
2642	SFT0.srl	2699	{
2643	}	2700	operand
2644	stage(4)	2701	{
2645	{}	2702	GPR IIInt31to0 of
2646	stage(5)	2703	GPR UInt31to0 by
2647	{	2704	GPR UInt31to0 g
2648	GPR.write	2705	Sin Sincsicos c,
2649	}	2705	format
2650	}	2700	f f
2651	behavior	2707	
2652	{	2700	່ 3080 a ", "ມ", "c
2653	a = b >>> c	2705	funationa
2654	}	2710	functions
2655	}	2712	1 (1)
2656	SUB	2712	stage(1)
2657	{	2713	BC mond
2658	operand	2714	PC. read
2659	{	2715	IMEM. read
2660	GPR STRt31to0 a	2710	PC.Inc
2661	GPR SINESILOO L,	2710	IR.read
2662	GPR SINESILOO D,	2710	}
2663	3	2719	stage(2)
2664	format	2720	
2665	{	2721	GPR. readu
2666	"SITB" = " " b " " o	2722	GPR.read1
2667		2723	}
2668	functions	2724	stage(3)
2669		2725	{
2670	stage(1)	2/20	ALU0.sub
2671	stage(1)	2/2/	}
2672	DC road	2728	stage(4)
2672	TMEM mond	2729	{}
2673	IMEM. read	2730	stage(5)
2675	FC.INC	2731	{
2675	lk.read	2732	GPR.write
2677	stage (2)	2733	}
2011	stage(2)	2734	}
2670		2735	behavior
4019	GPR.read0	2736	{

2797

2799

any a;

2737 a = b - c;2738 } 2739 } 2740 SW 2741 { 2742 operand 2743 { 2744 GPR 2745 [GPR, disp]:DMEM any b; 2746 3 2747 format 2748 { "SW" a ", " b 2749 2750 } 2751 functions 2752 { 2753 stage(1) 2754 { 2755 PC.read IMEM.read 2756 2757 PC.inc 2758 IR.read 2759 3 2760 stage(2) 2761 { GPR.read0 2762 2763 GPR.read1 EXT0.sign 2764 2765 } 2766 stage(3) 2767 { 2768 ALU0.add 2769 3 2770 stage(4) 2771 { DMEM.write 2772 2773 } 2774 stage(5) 2775 {} 2776 } 2777 behavior 2778 { 2779 *b = a; 2780 } 2781 } 2782 XOR 2783 { 2784 operand 2785 { 2786 GPR any a; GPR any b; 2787 2788 GPR any c; 2789 } 2790 format 2791 { "XOR" a ", " b ", " c 2792

2793

}

2794 functions 2795 { 2796 stage(1) { 2798 PC.read IMEM.read PC.inc 2800 IR.read 2801 2802 } 2803 stage(2) 2804 { 2805 GPR.read0 GPR.read1 2806 2807 } 2808 stage(3) 2809 { ALU0.xor 2810 2811 } stage(4) 2812 2813 {} stage(5) 2814 2815 { 2816 GPR.write 3 2817 2818 } 2819 behavior 2820 { 2821 $a = b^c;$ 2822 } 2823 } XORI 2824 2825 { 2826 operand 2827 { 2828 GPR any a; 2829 GPR any b; 'Imm 16 any c; 2830 2831 } 2832 format 2833 { "XORI" a ", " b ", " c 2834 2835 } 2836 functions 2837 { 2838 stage(1) 2839 { PC.read 2840 IMEM.read 2841 PC.inc 2842 IR.read 2843 2844 } stage(2) 2845 2846 { GPR.read0 2847 EXT0.zero_ext 2848 2849 } stage(3) 2850

2851 { 2852 ALU0.xor 2853 } 2854 stage(4) 2855 {} 2856 stage(5) 2857 { 2858 GPR.write 2859 } 2860 } 2861 behavior 2862 { 2863 $a = b^c;$ 2864 } 2865 1 2866 CKF_complexMAC 2867 { 2868 operand 2869 { 2870 GPR UInt31to0 a; 2871 GPR UInt31to0 b; 2872 } 2873 format 2874 { "CMULT" a ", " b 2875 2876 } 2877 functions 2878 { 2879 stage(1) 2880 { 2881 PC.read 2882 IMEM.read 2883 PC.inc 2884 IR.read 2885 } 2886 stage(2) 2887 { 2888 GPR.read0 2889 GPR.read1 2890 } 2891 stage(3) 2892 { 2893 CMAC0.mac 2894 } 2895 stage(4) 2896 {} 2897 stage(5) 2898 {} 2899 } 2900 behavior 2901 { 2902 complexMAC (a , b); 2903 } 2904 } 2905 CKF_LoadFromAcc 2906 { 2907 operand

{ GPR UInt31to0 a; } format { "CLOAD" a } functions { stage(1) { PC.read IMEM.read PC.inc IR.read 3 stage(2) ·{ } stage(3) { CMAC0.readacc } stage(4) {} stage(5) { GPR.write } 3 behavior { a = loadAcc (); 1 1 CKF_AccumClear { operand {} format { "ACMCLR" } functions ł stage(1) { PC.read IMEM.read PC.inc IR.read } stage(2) {} stage(3) { CMAC0.clracc }

2908

2909

2910

2911

2912

2913

2914

2915

2916

2917

2918

2919

2920

2921

2922

2923

2924

2925

2926

2927

2928

2929

2930

2931

2932

2933

2934

2935

2936

2937

2938

2939

2940

2941

2942

2943

2944

2945

2946

2947

2948

2949

2950

2951

2952

2953

2954

2955

2956

2957

2958

2959

2960

2961

2962

2963

2965 stage(4) 2966 {} stage(5) 2967 2968 {} 2969 } 2970 behavior 2971 { accumClear (); 2972 2973 } 2974 } 2975 CKF_setCpos 2976 { 2977 operand 2978 { 'Imm 5 UInt7to0 a; 2979 2980 } 2981 format 2982 ł 2983 "SETCPOS" a 2984 3 2985 functions 2986 { stage(1) 2987 2988 { PC.read 2989 2990 IMEM.read 2991 PC.inc IR.read 2992 2993 } stage(2) 2994 2995 {} 2996 stage(3) 2997 { 2998 CMAC0.ifracdigits 2999 } 3000 stage(4) 3001 {} stage(5) 3002 3003 {} 3004 } 3005 behavior 3006 { 3007 setCpos (a); 3008 } 3009 } 3010 CKF_setRpos 3011 { operand 3012 3013 { 'Imm 5 UInt7to0 a; 3014 3015 3 3016 format 3017 { 3018 "SETRPOS" a 3019 } 3020 functions 3021 {

3022 stage(1) 3023 { 3024 PC.read IMEM.read 3025 3026 PC.inc IR.read 3027 3028 } 3029 stage(2) 3030 {} 3031 stage(3) 3032 { 3033 CMAC0.ofracdigits 3034 } stage(4) 3035 3036 {} 3037 stage(5) 3038 {} 3039 } 3040 behavior 3041 { setRpos (a); 3042 3043 } 3044 } MFHT 3045 3046 { 3047 operand 3048 { 3049 GPR any a; HI any b; 3050 3051 } 3052 format 3053 ſ "MFHI" a 3054 3055 3 3056 functions 3057 { 3058 stage(1) 3059 { PC.direct_read 3060 IMEM.load_word 3061 PC.inc 3062 IR.direct_read 3063 3064 } 3065 stage(2) 3066 {} 3067 stage(3) 3068 { HI.direct_read 3069 3070 } 3071 stage(4) 3072 {} stage(5) 3073 3074 { GPR.write 3075 3076 } 3077 } behavior 3078

3079 { 3080 a = b; 3081 } 3082 } 3083 MFLO 3084 { 3085 operand 3086 { 3087 GPR any a; LO any b; 3088 3089 } 3090 format 3091 { 3092 "MFLO" a 3093 } 3094 functions 3095 { 3096 stage(1) 3097 { 3098 PC.direct_read 3099 IMEM.load_word 3100 PC.inc IR.direct_read 3101 3102 } 3103 stage(2) 3104 {} 3105 stage(3) 3106 { 3107 LO.direct_read 3108 } 3109 stage(4) 3110 {} 3111 stage(5) 3112 { 3113 GPR.write 3114 } 3115 } 3116 behavior 3117 { 3118 a = b; 3119 } 3120 } 3121 MTHI 3122 { 3123 operand 3124 { 3125 GPR any a; 3126 HI any b; 3127 } 3128 format 3129 { 3130 "MTHI" a 3131 } 3132 functions 3133 { 3134 stage(1) 3135 {

PC.direct_read IMEM.load_word PC.inc IR.direct_read } stage(2) { GPR.read0 } stage(3) {} stage(4) {} stage(5) { HI.direct_write } } behavior { b = a;} MTLO operand £ GPR any a; LO any b; 3 format { "MTLO" a } functions { stage(1) { PC.direct_read IMEM.load_word PC.inc IR.direct_read } stage(2) { GPR.read0 } stage(3) {} stage(4) {} stage(5) £ LO.direct_write } } behavior

3136

3137

3138

3139

3140

3141

3142

3143

3144

3145

3146

3147

3148

3149

3150

3151

3152

3153

3154

3155

3156

3157

3158

3159

3160

3161

3162

3163

3164

3165

3166

3167

3168

3169

3170

3171

3172

3173

3174

3175

3176

3177

3178

3179

3180

3181

3182

3183

3184

3185

3186

3187

3188

3189

3190

3191

3192

}

{

3250

3251

3252

3253

3254

3255

3256

3257

3258

3259

3260 3261

3262

3263

3264

3265

3266

3267

3268

3269

3270

3271

3272

3273

3274

3275

3276

3277

3278

3279

3280

3281

3282

3283

3284

3285

3286

3287

3288

3289

3290

3291

3292

3293

3294

3295

3296

3297

3298

3299

3300

3301

3302

3303

3304

3305

3306

3193 { b = a; 3194 3195 } 3196 } 3197 MULT 3198 { operand 3199 3200 { 3201 GPR SInt32to0 a; GPR SInt32to0 b; 3202 3203 HL SInt64to0 c; 3204 3 3205 format 3206 { "MULT" a ", " b 3207 3208 } 3209 functions 3210 { stage(1) 3211 3212 { PC.direct_read 3213 IMEM.load_word 3214 3215 PC.inc 3216 IR.direct_read 3217 } stage(2) 3218 3219 { GPR.read0 3220 3221 GPR.read1 3222 } 3223 stage(3) 3224 { MUL0.multiply_s 3225 3226 } stage(4) 3227 3228 {} 3229 stage(5) 3230 { 3231 HI.direct_write LO.direct_write 3232 3233 } 3234 } behavior 3235 3236 { c = a * b;3237 3238 } 3239 } MULTU 3240 3241 { operand 3242 3243 { 3244 GPR UInt32to0 a; GPR UInt32to0 b; 3245 3246 HL UInt32to0 c; 3247 } 3248 format 3249 {

"MULTU" a ", " b } functions { stage(1) { PC.direct_read IMEM.load_word PC.inc IR.direct_read 3 stage(2) { GPR.read0 GPR.read1 3 stage(3) { MUL0.multiply_u } stage(4) {} stage(5) { HI.direct_write LO.direct_write } } behavior { c = a * b;} } DIV ſ operand { GPR SInt31to0 a; GPR SInt31to0 b; HI SInt31to0 c; LO SInt31to0 d; } format { "DIV" a ", " b 3 functions ſ stage(1) { PC.direct read IMEM.load_word PC.inc IR.direct_read } stage(2)

3307 { 3308 GPR.read0 3309 GPR.read1 3310 3 3311 stage(3) 3312 { 3313 DIV0.divide_s 3314 } 3315 stage(4) 3316 {} 3317 stage(5) 3318 { 3319 HI.direct_write 3320 LO.direct_write 3321 } 3322 } 3323 behavior 3324 { 3325 c = a % b; 3326 d = a / b;3327 } 3328 ł 3329 DIVU 3330 { 3331 operand 3332 { . 3333 GPR UInt31to0 a; 3334 GPR UInt31to0 b; HI UInt31to0 c; LO UInt31to0 d; 3335 3336 3337 } 3338 format 3339 { "DIVU" a ", " b 3340 3341 } 3342 functions 3343 { 3344 stage(1) 3345 { 3346 PC.direct_read 3347 IMEM.load_word 3348 PC.inc 3349 IR.direct_read 3350 } 3351 stage(2) 3352 { 3353 GPR.read0 3354 GPR.read1 3355 } 3356 stage(3) 3357 { 3358 DIV0.divide_u 3359 } 3360 stage(4) 3361 {} 3362 stage(5)

3363

{

3364 HI.direct_write 3365 LO.direct_write 3366 } 3367 } 3368 behavior 3369 { 3370 c = a % b; 3371 d = a / b;3372 } 3373 } 3374 3375 } 3376 3377 structure 3378 { 3379 PC { 3381 class { PC } stage { 1 } connection { out1 { IMEM.in1 ADD0.in1 3 } 3391 3 IMEM { class { IMEM } stage { 1 } connection { out1 { IR.in1 } } } IR { class { IR } stage { 1 } connection { out1 ſ stage 2 GPR.in1 stage 2 GPR.in2 stage 5 GPR.in3 stage 5 GPR.in4 EXT0.in1 SFT0.in1 } }

3380

3382

3383

3384

3385

3386

3387

3388

3389

3390

3392

3393

3394

3395

3396

3397

3398

3399

3400

3401

3402

3403

3404

3405

3406

3407

3408

3409

3410

3411

3412

3413

3414

3415

3416

3417

3418

3419

3420

}

3421 portion 3422 GPR { 3423 class { GPR } 3424 stage { 2 } 3425 3426 connection 3427 { 3428 out1 { ALU0.in1 3429 3430 3431 CMP0.in1 3432 CMAC0.in1 SFT0.in1 3433 3434 MUL0.in1 3435 DIV0.in1 3436 MADD1U0.in1 3437 MADD2U0.in1 3438 ADDBLOCK1U0.in1 3439 } 3440 out2 { PC.in1 ALU0.in2 CMP0.in2 CMAC0.in2 3441 3442 3443 3444 3445 SFT0.in2 3446 3447 DMEM.in2 3448 MUL0.in2 3449 DIV0.in2 3450 MADD1U0.in2 3451 MADD2U0.in2 3452 ADDBLOCK1U0.in2 3453 } 3454 } } 3455 portion 3456 3457 GPR 3458 { class { GPR } 3459 stage { 5 } 3460 connection 3461 3462 { out1 3463 3464 {} 3465 out2 3466 {} 3467 } 3468 } EXT0 3469 3470 { class { EXT } 3471 3472 stage { 2 } 3473 connection 3474 { out1 3475 { ALU0.in2 3476 3477

3478 ADD0.in2 } 3479 3480 } 3481 } 3482 ADD0 3483 { class { ADD } 3484 stage { 3 } 3485 connection 3486 3487 { 3488 out1 3489 { 3490 PC.in1 3491 } 3492 } 3493 } 3494 ALU0 3495 { class { ALU } 3496 3497 stage { 3 } connection 3498 3499 { 3500 out1 3501 { DMEM.in1 3502 3503 NOT0.in1 3504 stage 5 GPR.in4 3505 } 3506 } 3507 } 3508 DIV0 3509 { class { DIV } 3510 3511 stage { 3 } connection 3512 3513 { out1 3514 3515 { HI.in1 3516 } 3517 3518 out2 3519 { LO.in1 3520 3521 } 3522 } 3523 } SFT0 3524 3525 { class { SFT } 3526 stage { 3 } 3527 3528 connection 3529 { 3530 out1 3531 { stage 5 GPR.in4 3532 } 3533 3534 }

3535 } 3536 DMEM 3537 { 3538 class { DMEM } 3539 stage { 4 } 3540 connection 3541 { 3542 out1 3543 { 3544 stage 5 GPR.in4 3545 } 3546 } 3547 } 3548 CMP0 3549 { 3550 class { CMP } stage { 3 } 3551 3552 connection 3553 { 3554 out1 3555 { 3556 } 3557 } 3558 } 3559 CMAC0 3560 { 3561 class { CMAC } 3562 stage { 3 } 3563 connection 3564 { 3565 out1 3566 { 3567 stage 5 GPR.in4 3568 } 3569 } 3570 } 3571 NOT0 3572 { 3573 class { NOT } 3574 stage { 3 } 3575 connection 3576 { 3577 out1 3578 { 3579 stage 5 GPR.in4 3580 } 3581 } 3582 } 3583 MUL0 3584 { class { MUL } 3585 3586 stage { 3 } 3587 connection 3588 { 3589 out1 3590 { HI.in1 3591

3592 LO.in1 3593 } 3594 } 3595 } 3596 ΗI 3597 { 3598 class { HI } stage { 5 } 3599 3600 connection 3601 { 3602 out1 3603 { 3604 stage 5 GPR.in4 3605 3 3606 } 3607 } 3608 LO3609 { 3610 class { LO } 3611 stage { 5 } 3612 connection 3613 { 3614 out1 3615 { 3616 stage 5 GPR.in4 3617 } 3618 } 3619 } 3620 }

List of Major Publications of the Author

Journal Papers

- Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, and Masaharu Imai: "JPEG Encoder Implementation Using the ASIP Development System: PEAS-III," IPSJ Journal (submitted paper).
- [2] Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, and Masaharu Imai: "A Compiler Generation Method for HW/SW Codesign Based on Configurable Processors," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E85-A, No.12, pp. 2586-2595, Dec. 2002.
- [3] Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima and Masaharu Imai: "Proposal of a Multi-Threaded Processor Architecture for Embedded Systems and Its Evaluation," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E84-A, No. 3, pp. 748-754, Mar. 2001.

International Conference Papers

[1] Koji Okuda, Shinsuke Kobayashi, Yoshinori Takeuchi, Masaharu Imai: "A Simulator Generator Based on Configurable VLIW Model Considering Synthesizable HW Description and SW Tools Generation," Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies 2003, (to appear).

- [2] Akira Kitajima, Yoshinori Takeuchi, Akichika Shiomi, Jun Sato, Shinsuke Kobayashi, Masaharu Imai: "Architectural Design Space Exploration of Configurable Processors using ASIP Meister," Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies 2003, (to appear).
- [3] Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, Masaharu Imai: "Rapid Prototyping of JPEG Encoder Using the ASIP Development System: PEAS-III," IEEE International Conference on Acoustics, Speech, and Signal Processing 2003 (to appear).
- [4] Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, Masaharu Imai: "Design Space Exploration for DSP Applications using the ASIP Development System PEAS-III," Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing 2002, Vol.3, pp.3168 - 3171, May 13 - 17, 2002
- [5] Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai: "Compiler Generation in PEAS-III: an ASIP Development System," Proceedings of Software and Compilers for Embedded Systems 2001, Mar. 2001.
- [6] Toshiyuki Sasaki, Shinsuke Kobayashi, Tomohide Maeda, Makiko Itoh, Yoshinori Takeuchi, and Masaharu Imai: "Rapid Prototyping of Complex Instructions for Embedded Processors using PEAS-III," Proc. Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies 2001, pp 61-66, Nara, Japan, Oct. 2001.

National Conference Papers

- Kentaro Mita, Shinsuke Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi, Masaharu Imai: "A Proposal of Zero Overhead Loop Model in ASIP Meister," Technical Report in IEICE, CPSY2002-58, vol. 102, No. 478, pp. 43-48, Nov. 2002 (in Japanese).
- [2] Yoshinori Takeuchi, Shinsuke Kobayashi, Masaharu Imai: "An ASIP development environment ASIP Meister and its Application to DSP," Technical Report in IEICE, CAS2002-61, vol. 102, No. 295, pp. 73-78, Sep. 2002 (in Japanese).

LIST OF MAJOR PUBLICATIONS OF THE AUTHOR

- [3] Koji Okuda, Shinsuke Kobayashi, Yoshinori Takeuchi, Masaharu Imai: "Proposal of an Architecture Model and a Simulator Generator for Configurable VLIW Procesor," IPSJ Symposium Series, Vol. 2002, No.10, pp. 161-166, Jul. 2002 (in Japanese).
- [4] Nobuyuki Hikichi, Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, Masaharu Imai: "Proposal of Common Processor Architecture Description for ASIP Design Automation – Integration of Processor and Machine Description for Retargetable Compiler –," Technical Report in IEICE, VLD2002-60, vol. 102, No. 163, pp. 25-30, Jun. 2002 (in Japanese).
- [5] Shinsuke Kobayashi, Kentaro Mita, Yoshinori Takeuchi, Masaharu Imai: "A Compiler Generation Method in The PEAS-III System and Its Evaluation," Technical Report in IEICE, VLD2001-145, vol. 101, No. 577, pp. 101-108, Jan. 2002 (in Japanese).
- [6] Kentaro Mita, Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima, and Masaharu Imai: "A Case Study of Compiler Generator for PEAS-III System," IPSJ Symposium Series, Vol. 2001, No. 8, pp. 143-148, Jul. 2001 (in Japanese).
- [7] Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai: "An Evaluation of Processor Cores for Embedded Systems using Multithreading Mechnism," Proceedings of The 13th Workshop on Circuits and Systems in Karuizawa, pp. 533-538, Apr. 2000 (in Japanese).
- [8] Shinsuke Kobayashi, Yoshinori Takeuchi, Akira Kitajima, and Masaharu Imai: "A Proposal of a Processor for Multi-threading Using Interleaving Threads Mechanism," IPSJ Sig Notes, 99-ARC-135 Vol.99, No.100, pp.45-50, Nov. 1999 (in Japanese).