A Processor Generation Method from Instruction Behavior Description Based on Specification of Pipeline Stages and Functional Units

Takeshi SHIRO, Masaaki ABE, Keishi SAKANUSHI, Yoshinori TAKEUCHI, and Masaharu IMAI

Graduate School of Information Science and Technology,
Osaka University,
1-5, Yamada-oka, Suita, Osaka 565-0871, Japan
e-mail: {t-siro, m-abe, sakanusi, takeuchi, imai}@ist.osaka-u.ac.jp

Abstract— This paper proposes a method for generating a pipeline processor from the behavior description of instructions. In the proposed method, a micro-operation description is generated by complementing the behavior description with specifications of the pipeline stages, such as the number of pipeline stages, the attributes of each stage. From the behavior description, software development tools, such as an instruction-set simulator (ISS), a compiler, and an assembler can be generated, and a synthesizable HDL description of a processor can be generated from the micro-operation description. Compared with the conventional method of writing individual descriptions, the proposed method can dramatically reduce the code size of the architectural description language and the design time without degrading the design quality. As a result, a design space exploration can be performed efficiently.

I. Introduction

In recent years, embedded systems that require severe design goals have been employed in application-specific instruction set processors (ASIPs), because ASIPs are more flexible than application-specific integrated circuits (ASICs) and have higher performance than general-purpose processors. The performance of an ASIP depends on the instruction sets, functional units, and the number of pipeline stages. Therefore, a design space exploration is required to design an optimum ASIP. To evaluate an ASIP architecture, software development tools, such as instruction-set simulators (ISS), compilers, and assemblers, are required. However, the specifications of the software development tools must be changed if the architecture changes. Since designing and modifying processors and software development tools are time-consuming tasks, the ASIP design and performance evaluation process is not efficient. An ASIP design environment that can generate a synthesizable HDL description of an ASIP and software development tools is of great benefit to explore design space of ASIPs.

ASIP Meister [1] is an ASIP design environment. It automatically defines the data paths and control logic of a processor, and generates a synthesizable HDL description and software development tools, such as an instruction-set simulator, a C compiler, and an assembler, from its specific architectural description language (ADL). The instructions are specified in two descriptions, a micro-operation description and a behavior description. The micro-operation description represents the operations of each pipeline stage for each instruction, and is used to generate the HDL description. On the other hand, the behavior description represents instruction semantics, and software development tools can be generated from it. With these two forms, the designer can obtain various ASIPs and evaluate them with the generated software development tools.

However, in conventional ASIP design environments, if there is inconsistency between the micro-operation description and the behavior description, the generated ISS cannot correctly simulate the behavior of the generated processor, or the generated processor cannot execute code generated by the assembler or compiler. Furthermore, the code size of the micro-operation description is larger than that of the behavior description, and usually the task of describing the micro-operation description takes up most of the ASIP design time. In addition, since the micro-operation description of all instructions must be rewritten by changing the number of pipeline stages, it is difficult for a designer to explore the optimum pipeline architecture within the limited processor design term. Consequently, the design space exploration is not efficient.

In this paper, we propose a method of generating the micro-operation description from the behavior description. In this method, by using abstract syntax trees (ASTs) constructed from the behavior description, micro-operation descriptions are generated without pipeline stages and functional units (micro-operation segments). The number of pipeline stages and the attributes of each stage defined by a designer are automatically reflected in generated micro-operation segments. Then, functional units are bound to each micro-operation segment. Accordingly, the processor and software development tools can be efficiently designed by writing only the behavior description. A designer can quickly evaluate the processor architecture with the generated software development tools, and consequently, the design space exploration can be performed efficiently.

The rest of this paper is organized as follows. Section II introduces the conventional pipeline processor generation method. Section III describes the method of generating the micro-operation description from the behavior description. Experimental results are shown in Section IV, and Section V summarizes this paper.

II. Design of Pipeline Processors

We implemented the proposed method in the framework of ASIP Meister. This section describes related work and the design flow of a pipeline processor and software development tools with ASIP Meister.

A. Related Work

There are several processor generation methods. Xtensa [3] is a customizable configurable processor core, and MetaCore [2] is a digital signal processor (DSP) design environment. A designer can add new instructions and hardware units to the base processor in these systems. Although specific instructions defined by a designer can be implemented in the target processor, the number of pipeline stages cannot be customized in these approaches. Therefore, they have low flexibility. MIMOLA hardware design system [4] [5] can generate an application-specific instruction set processor, a compiler, and an instruction-set simulator. MIMOLA provides a flexible definition of the processor architecture. The description of a processor in MIMOLA is similar to HDL, MIMOLA does not provide definition of data paths of a processor automatically. A designer must describe data paths of a processor. The modification of a processor is difficult in MIMOLA. LISA [6][7] is an Architecture Description Language (ADL). The LISA language describes an instruction set, a pipeline architecture, and behavior of each instruction. A designer can get the HDL description of a processor and software development tools by describing them in the LISA language. In LISA, a designer has to specify the control logic of the pipeline registers. Therefore, the design space exploration of the number of pipeline stages is not efficient, and a designer must understand the details of the controller and describe the control logic, which are usually error-prone tasks. C-DASH [8] is a system design language based on an instruction set architecture (ISA). A designer does not specify the control logic of pipeline registers. In C-DASH, the instructions must be described at both the behavior level and micro-operation level. However, if there is inconsistency between the two descriptions, the generated hardware core or related software development tools will not work correctly.

B. Processor Design Flow in ASIP Meister

ASIP Meister's design flow for processors and software development tools consists of six steps. (1) First, the architecture of the processor is defined. The design goal (area, delay, and power), the number of pipeline stages, the attributes of each stage, and the number of delay slots are defined. (2) Next, the resources (storages and functional units) in the processor are defined. The flexible hardware model (FHM) [9] is used as a reusable resource model. In FHM, parameters, such as bit range, are parameterized, and resource instances are generated by specifying parameter values. (3) After that, the instruction set is defined. Instruction name, operation code, and operands are defined. (4) The semantics of each instruction are then defined with the behavior description, and (5) the operations of each instruction are defined with the micro-operation description. (6) Finally, the HDL description of the processor and software development tools, such as instruction set simulator, C compiler, and assembler, are generated.

```
Instruction ADD {
  Operand Definition{
        1{opname{rd}
           usage{register}
           addressing mode{register direct (GPR) }
           datatype{SInt31to0}}

        2{ opname{rs0}
           usage{register}
           addressing mode{register direct (GPR)}
           datatype{SInt31to0}}

        3{ opname{rs1}
            usage{register}
            addressing mode{register direct (GPR)}
            datatype{SInt31to0}}}

  Semantics Definition{ rd = rs0 + rs1; }
}
```

Fig. 1. Behavior Description of Instruction ADD

The behavior description is used to generate the software development tools. The behavior description consists of an operand definition and semantics definition. In the operand definition, a designer defines the name, usage, addressing mode, and type (signed/unsigned and bit range) for each operand. In the semantics definition, the designer defines the instruction semantics.

Figure 1 shows the behavior description of the instruction ADD. In Figure 1, operands of the instruction ADD ($rs0$, $rs1$, and rd) are registers of the registerfile GPR. In the semantics definition, $rs0$ and $rs1$ are added, and the result is stored in rd.

The micro-operation description is used to generate a HDL description of a processor. It defines the micro-operations of each stage with the variables and resources that were defined in the resource definition step. A definition of a variable is described in the following format.

$$\textbf{wire}\ \textbf{[}\ bit\_range\ \textbf{]}\ name\_of\_variable\ \textbf{;}$$

An operation in a stage is described in the following format.

$$variable\ \textbf{=}\ resource\ .\ function\ (\ variables\ )\ \ \textbf{;}$$

Figure 2 shows the micro-operation description of the instruction ADD in a processor with five pipeline stages. Variables used in multiple pipeline stages are defined in $variable$ as global variables. In $stage1$, an instruction word is fetched.

```
Instruction ADD{
  variable{
        wire[31:0] source0;
        wire[31:0] source1;
        wire[31:0] result;}

  1{   wire[31:0] current_pc;
        wire[31:0] inst;
        current_pc = PC.read();
        inst = IMAU.read(current_pc);
        null = IR.write(inst);
        null = PC.inc();}

  2{    source0 = GPR.read0(rs0);
        source1 = GPR.read1(rs1);}

  3{    wire[3:0] flag;
       <result,flag> = ALU0.add(source0,source1);}

  4{  }

  5{   null = GPR.write0(rd,result);}
}
```

Fig. 2. Micro-Operation Description of Instruction ADD

In $stage2$, $rs0$ and $rs1$, which are registers of the registerfile GPR are loaded and stored in intermediate variables $source0$ and $source1$. In $stage3$, $source0$ and $source1$ are added with the function add of $ALU0$, and the result is stored in $result$. In $stage4$, there is no code in the instruction ADD. Finally, in $stage5$, result is stored in register rd.

III. Method of Generating Micro-Operation Description

This section describes the method of generating the micro-operation description from the behavior description.

A. Micro Operation Description and Behavior Description

ASIP Meister uses both the micro-operation description and the behavior description of the processor. The micro-operation description represents the operations of each pipeline stage. The designer must explicitly specify the functional units. On the other hand, the behavior description represents the semantics of the instructions. The designer does not have to specify the instruction behavior of each pipeline stage and functional unit. The code size of the micro-operation description is larger than that of the behavior description. Moreover, if there is inconsistency between the micro-operation description and the behavior description, the generated processor or software development tools will not work correctly. Therefore, a designer must carefully write two descriptions, and this is usually troublesome. In addition, since the micro-operation description of all instructions must be rewritten if the number of pipeline stages is changed, it is difficult for a designer to explore the optimum pipeline architecture within the limited processor design term.

Specification of Pipeline Stages	
Stage Name	Attribute
Stage1	Instruction Fetch
Stage2	Operand Fetch Sign-Extension
Stage3	Execution Jump or Branch
Stage4	Memory Read Memory Write
Stage5	Register Write

Fig. 3. Specification of Pipeline Stages

To manage this problem, the instructions should have only one description, the behavior description is the easiest choice. Hence it is that we propose generating the micro-operation description from the behavior description. However, the behavior description is insufficient for specifying the pipeline stages and functional units of the micro-operation description. Therefore, for our purposes, the behavior description must be complemented with such information. In particular, our method uses the architecture definition information about the pipeline stages. The resulting micro-operation description is consistent with the behavior description, and the design time is dramatically reduced.

Although interrupts (external, internal, reset, etc.) must be described with the micro-operation description, our method does not generate the micro operation description of interrupts because the code size of the interrupts is considerably smaller than that of the instructions. That is, since describing interrupts with micro-operation description hardly affects the design time, we assume that the designer can describe them without difficulty.

B. Specification of Pipeline Stages

The proposed method assumes a pipeline processor with a Harvard architecture. The pipeline stages, such as the number of stages and the attributes of each stage, are defined in the architecture definition step.

Figure 3 shows an example of specification of pipeline stages that was defined by a designer. The example is for a processor with five pipeline stages. stage1 is for instruction fetch, stage2 is for instruction decode, operand fetch, and sign extension, Execution and access to the program counter in branch or jump instructions are performed in stage3, stage4 is for memory access, and stage5 is for writing back to the register file.

C. The Proposed Method

The proposed method consists of four parts: (1) Abstract syntax trees (ASTs) are constructed from the behavior description; (2) By scanning constructed ASTs, the micro-operation descriptions are generated without having to specify the pipeline stages and the functional units (micro-operation segments); (3) Micro-operation segments are allocated to the

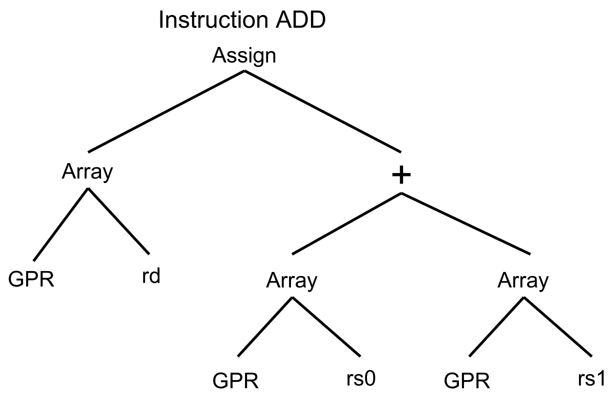


Fig. 4. AST of Instruction ADD

pipeline stages; (4) Functional units are defined for each micro-operation segment.

C.1 Abstract Syntax Trees (ASTs)

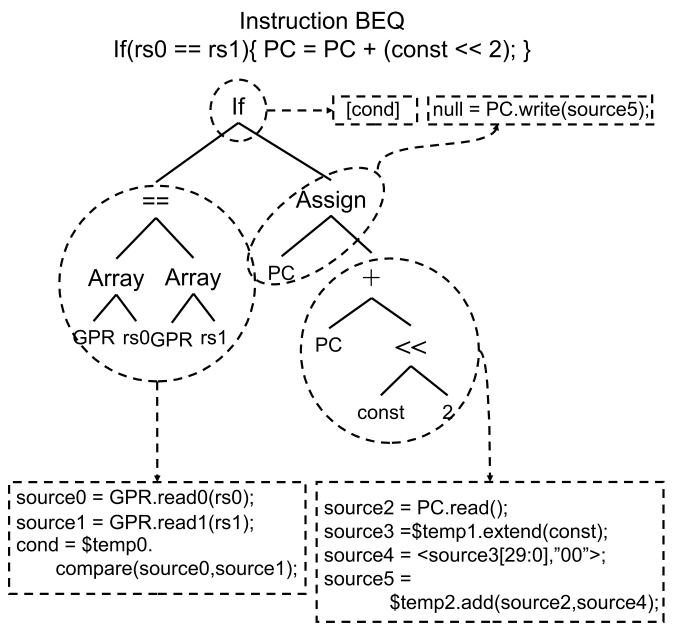
An AST is constructed from the behavior description of each instruction. An AST expresses sentences or expressions. A sentence expresses assignments or if-statements. Assignments and if-statements are expressed as ASTs that have $Assign$ and If respectively as a root node. An expression expresses functional expressions, data type, operands, or integer values. A functional expression is expressed as ASTs that have an operator ($+$, $-$, $*$, $/$, $\%$, $<<$, $>>$, $>$, $<$, $<=$, $>=$, $==$, $!=$, and, or, xor, not, $extend$, $Cast$, $Array$) as a root node. Type, operands, and integers are expressed as nodes of type name (signed/unsigned bit-range), string, and integer value, respectively. An operator $Cast$ expresses a cast of a type. An operator $Cast$ has a data type name as a left child and expression as a right child. An operator $Array$ expresses an array. An operator $Array$ has an array name as a left child and an index as a right child.

Figure 4 shows an example of an AST for Figure 1. The registers $rs0$ and $rs1$ in the registerfile GPR are added in the right child, and the result is stored in register rd in the left child.

C.2 Micro-Operation Segments

The micro-operation segments are generated by scanning the constructed AST in post order. To generate nodes of operators, we have to define a function for a node of an operator. In this step, only functions are defined. That is, the functional units are not specified yet.

Figure 5 shows the micro-operation segments of the instruction BEQ. In the left child of If, micro-operation segments to read $rs0$ and $rs1$ from registerfile GPR, and to compare $rs0$ with $rs1$ are generated. A functional unit for comparison is expressed as a temporary functional unit $temp0$; only the function $compare$ is decided. The result of the comparison is stored in the variable $cond$. Variable $cond$ is '1' only if $rs0$ is equal to $rs1$. In the right child, the micro-operation segments for calculating the target address and jumping to the target address are generated. The functional units for sign extension and addition


Instruction BEQ
If(rs0 == rs1){ PC = PC + (const << 2); }
If
[cond]
null = PC.write(source5);
==
Assign
Array
Array
GPR
rs0
GPR
rs1
PC
+
PC
<<
const
2

```
source0 = GPR.read0(rs0);
source1 = GPR.read1(rs1);
cond = $temp0.
    compare(source0,source1);
```

```
source2 = PC.read();
source3 =$temp1.extend(const);
source4 = <source3[29:0],"00">;
source5 =
        $temp2.add(source2,source4);
```

Fig. 5. Micro-Operation Segments of Instruction BEQ

TABLE I
EXAMPLE OF DECIDING FUNCTIONAL UNITS

Instruction Name	Function	Functional Unit	Decided Instance of Functional Unit
ADD	add	alu	ALU0
SUB	sub	alu	ALU0
MULT	mul	multiplier	MULT0
DIV	div	divider	DIV0

are not decided; only the functions $extend$ and add are decided. $[cond]$ is placed in the head of "$PC.write(source5)$". This means the following sentence is executed only if $cond$ is '1'.

The proposed method assumes that the micro-operation description of the instruction fetch is written by the designer, because it is not expressed in the behavior description of the instructions.

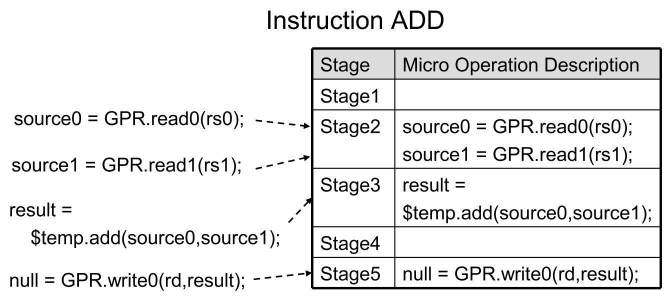
C.3 Allocation of Segments

After the micro-operation segments of all instructions are generated, the micro-operation segments are allocated to the pipeline stages according to the designer's specification.

Figure 6 is an example of micro-operation segment allocation for the five pipeline stages in the instruction ADD. The micro-operation segments are allocated according to the information in Figure 3. Read functions of register $rs0$ and $rs1$ from the registerfile GPR are allocated to stage2. Addition of $rs0$ and $rs1$ is allocated to stage3. Finally, a write back operation of the result is allocated to stage5.

C.4 Deciding Functional Units

Next, the temporary functional units are replaced by instances of functional units.


Instruction ADD
source0 = GPR.read0(rs0);
source1 = GPR.read1(rs1);
result = $temp.add(source0,source1);
null = GPR.write0(rd,result);

Stage	Micro Operation Description
Stage1	
Stage2	source0 = GPR.read0(rs0); source1 = GPR.read1(rs1);
Stage3	result = $temp.add(source0,source1);
Stage4	
Stage5	null = GPR.write0(rd,result);

Fig. 6. Example of Allocation of Micro-operation Segments to Pipeline Stages in Instruction ADD

TABLE II
COMPARISON OF CODE SIZE

MIPS R3000 subset	behavior description	micro-operation description	total
conventional	165	292	457
proposed	165	0	165

DLX subset	behavior description	micro-operation description	total
conventional	195	392	587
proposed	195	0	195

(Unit: lines)

TABLE III
COMPARISON OF DESIGN TIME

MIPS R3000 subset	behavior description	micro-operation description	others	total
conventional	45	100	85	230
proposed	45	0	75	120

DLX subset	behavior description	micro-operation description	others	total
conventional	60	125	115	300
proposed	60	0	95	155

(Unit: minutes)

Table I shows an example of decided instances of functional units. The first column shows the instruction name, and the function used in the instruction of the first column is shown in the second column. The third column shows the functional unit that has the function in the second column, and the fourth column shows an instance of a functional unit in the third column. For example, the function add used in instruction ADD is executed by $ALU0$.

After deciding the instance of the functional unit, the temporary functional units in micro-operation segments are replaced by decided instances.

The proposed method does not consider the design space exploration of combinations of functional units. That is, it assumes that there is only one functional unit for a certain function.

IV. EXPERIMENTS

We designed several processors with the proposed method and the conventional method, in which the micro-operation description is manually described by a designer, and compared their code sizes, design times, and design qualities. The processors were of the MIPS R3000 [11] subset (42 instructions implemented) and DLX [12] subset (51 instructions implemented). The number of pipeline stages of the DLX subset was changed to three to determine whether the proposed method could generate processors with different numbers of stages. Then, to confirm that the proposed method could generate specific instructions, and quickly change the number of pipeline stages, we expanded the DLX subset by implementing several specific instructions and changing the number of pipeline stages.

A. Experimental Results

Table II shows the results of the code size comparison. The code size is the sum of the number of lines of the operand definition and the semantics definition. In ASIP Meister, the designer can write the micro-operation description by using macros, which can generate operations that are common among instructions, such as instruction fetches or arithmetic operations. In this experiment, the code size of the micro-operation description was taken to be the total code size of the macros and the micro-operation description with macros. Inspection of Table II confirms that the proposed method reduced the code size by about 64% in MIPS R3000 and about 67% in DLX compared with the conventional method.

Table III compares design times. The conventional method took longer in describing the micro-operation description. Compared with the conventional method, the proposed method reduced design time by about 50% for both MIPS R3000 and DLX

We synthesized HDL descriptions of the generated processors and evaluated their area and delay. Table IV compares the results of the logic synthesis. We used Synopsys Design Compiler [13] as a logic synthesis tool. In MIPS R3000 subset, area of the processor generated by the proposed method increased, compared with that of the processor generated by the conventional method, because the combination of functional units is different between the proposed method and the conventional method. However, as a whole, the Table IV shows that the proposed method could generate synthesizable HDL descriptions of processors and hardly degrade the design quality from that of the conventional method.

B. Modification of DLX Processor

We implemented several specific instructions in the DLX processor and changed the number of pipeline stages. Implemented instructions are shown in Table V. The implemented instructions were multiply and accumulate (MAC), increment/decrement memory access, calculation of absolute, and compare and exchange. The expanded DLX processor had five pipeline stages and two execution stages.

Figure VI shows the design quality and the design time of

TABLE IV
DESIGN QUALITY

MIPS R3000 subset		
	Area (gates)	Delay (ns)
conventional	36829	8.95
proposed	39102	8.97

DLX subset		
	Area (gates)	Delay (ns)
conventional	36019	9.72
proposed	36114	9.83

(library: 0.18μm CMOS)

TABLE V
IMPLEMENTED INSTRUCTIONS

Instruction Type	Behavior
MADD, MSUB	Multiply and Addition/Subtraction
LDINC, LDDEC	Load and Increment/Decrement
STINC, STDEC	Store and Increment/Decrement
ABS	Absolute
CEX	Compare and Exchange

the customized DLX. The design time is the time to expand the DLX processor; add new instructions and change the number of pipeline stages. In the proposed method, the time to change the number of pipeline stages is perfomed within a few minutes, because the proposed method automatically allocates each micro-operation segment to the pipeline stages, base on the specification of each pipeline stages defined by a designer. Apparently, the proposed method could generate the processor with specific instructions and change the number of pipeline stages in a short period.

By the experimental results, we confirm that the proposed method provides reduction of code size and design time, and generated processor by the proposed method hardly degrade the design quality compared with the conventional method.

V. CONCLUSION

This paper described a method of generating a micro-operation description from a behavior description. The proposed method quickly generates a synthesizable HDL description and software development tools, and the generated processor and software development tools are perfectly consistent because they are generated from a single description. In experiments, we designed three processors with the proposed method and the conventional method of manually generating both descriptions, and compared their code sizes, design times, and design qualities. The results showed that in comparison with the conventional method, the proposed method dramatically reduced code size and design time and did not degrade the design quality.

Future research will include optimizing the micro-operation segments and the combination of functional units, and implementation of an algorithm to minimize the number of pipeline stages [10].

TABLE VI
DESIGN QUALITY AND DESIGN TIME

DLX + Specific Instructions			
	Area (gates)	Delay (ns)	Design Time
conventional	52169	8.41	50
proposed	52888	8.52	20

(library: 0.18μm CMOS)

REFERENCES

[1] M. Imai, “ASIP Meister : A Configurable Processor Core Development System,” Proc. ITI of 3rd International Conference on Information & Communications Technology (ICICT 2005), Cairo, Egypt, Dec. 2005.

[2] J.-H. Yang, B.-W. Kim, S.-J. Nam, Y.-S. Kwon, D.-H. Lee, J.-Y. Lee, C.-S. Hwang, Y.-H. Lee, S.-H. Hwang, I.-C. Park, and C.-M. Kyung, “MetaCore: An Application Specific DSP Development System,” IEEE Transactions on Very Scale Integration (VLSI) Systems, Vol. 8, No.2, pp. 173-183, Apr. 2000.

[3] G. Ezer, “Xtensa with user defined DSP coprocessor microarchitectures,” Proc. of 2000 IEEE International Conference on Computer Design, VLSI in Computers & Processors (ICCD 2000), pp. 335-342, 2000.

[4] R. Leupers and P. Marwedel, “Retargetable Code Generation Based on Structural Processor Description,” Design Automation for Embedded Systems, Springer Netherlands, 1998.

[5] R. Leupers, J. Elste, and B. Landwehr, “Generation of interpretive and compiled instruction set simulators,” Proc. of Asia and South Pacific Design Automation Conference 1999 (ASP-DAC 1999), pp. 339-342, Hong Kong, Jan. 1999.

[6] A. Hoffmann, H. Meyr, and R. Leupers, “Architecture Exploration for Embedded Processors with LISA,” Kluwer Academic Publishers, 2002.

[7] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, “A methodology for the Design of Applications Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA,” Proc. of the International Conference on Computer Aided Design (ICCAD), pp. 625-630, San Jose, USA, Nov. 2001.

[8] H. Yanagisawa, M. Uehara, and H. Mori, “ISA based system design language in HW/SW co-design environment,” Proc. of the 13th IEEE International Workshop on Rapid System Prototyping (RSP 2002), pp. 122-127, 2002.

[9] T. Morifuji, Y. Takeuchi, J. Sato, and M. Imai, “Flexible Hardware Model Database Management System Implementation and Effectiveness,” Proc. of the Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 1997), pp. 83-89, Osaka, Japan, Dec. 1997.

[10] M. Abe, K. Sakanushi, Y. Takeuchi, and M. Imai, “Pipeline Stage Minimization Algorithm for Embedded Processors,” Technical Report of IEICE, DSP2003-60, Vol. 103, No. 147, pp. 55-60, 2003 (in Japanese).

[11] MIPS Technologies Inc., http://www.mips.com

[12] J. L. Hennessy, and D. A. Patterson, “Computer Architecture: A Quantitative Approach Second Edition,” Morgan Kaufmann Publishers, Inc., 1996.

[13] Synopsys Inc., http://www.synopsys.com