Configurable Processor Hardware Developing Environment for RISC-V with Vector Extension

Ryo Taketani
Department of Information Systems Engineering
Graduate School of Infomation Science and Technology
Suita, Osaka 565-0871
r-taketn@ist.osaka-u.ac.jp

Yoshinori Takeuchi
Department of Electric and Electronic Engineering
School of Science and Engineering
Higashi-Osaka, Osaka 577-8502
takeuchi@ele.kindai.ac.jp

Abstract— This study proposes a processor hardware development environment that can easily change the architecture configuration suitable of specific processing for RISC-V. RISC-V is getting more attention as an open Instruction Set Architecture (ISA) nowadays. RISC-V has vector extension specified for parallel computing that takes power savings and high executed cycle performance into consideration. On the other hand, in the age of the Internet of Things (IoT) in recent years, the number of types of devices equipped with microprocessor has increased. In other words, each IoT device demands processors specialized for each application. Therefore, expectations for configurable processors are increasing. This paper challenged to construct a HW development environment for a RISC-V based configurable processor with vector extension. In addition, validation of the generated processor was performed to evaluate the performance of the HW, and design man-hours were compared with the conventional method for instruction expansion.

I. Introduction

In recent years, microprocessors tend to be mounted on various devices in Internet of Things (IoT) era. Each IoT device has demanded application-specific processing due to low-volume, high-mix production of them. Thus, configurable processors which easily modify its Instruction Set Architecture (ISA) are receiving increased expectations all over the world.

On the other hand, RISC-V ISA [1] has achieved remarkable progress nowadays. RISC-V is simple and highly expandable ISA have been proposed by University of California, Berkeley. Furthermore, many of RISC-V software (SW) development tools are free and open such as binary utilities, compiler, operating system, and ISA simulator. These SW tools are unified into one github repository riscv-tools [2]. Therefore, we can easily extend ISA of these SW tools by adding instruction opcodes, operands, and behavior descriptions. Moreover, many of RISC-V hardware (HW) environment are also free and open such as Rocket-Chip [3], Boom [4], VexRiscv [5], and so on. However, in contrast to SW environment, designers need to rewrite Hardware Description Language (HDL) directly for extend ISA of these HW tools. HDLs are low abstraction level description and difficult for developers except for HW professionals. Moreover, RISC-V has vector extension called "V" extension. Vector architecture includes parallel computing unit similar to SIMD (Single Instruction Multiple Data) computing unit. SIMD computer executes operation on multiple data in parallel by dividing one register. On the other hand, Vector computer holds vector registers for parallel operation separately from the general purpose register. In addition that, RISC-V vector architecture has additional convenient functions such as register type conversion, predication, and so on. RISC-V vector extension is important when processors for embedded devices demand power savings and high performance.

From the above background, this study challenges to construct RISC-V based HW development environment for configurable processor. For constructing HW of configurable processor, this study uses ASIP Meister [6]. ASIP Meister is an interactive tool for developing application-specific processor. ASIP Meister generates the processor HDL automatically from design parameters and micro operation descriptions defined by developers. A method that can extend the instruction set with behavior level descriptions on ASIP Meister has been proposed [7]. By using ASIP Meister, designers can easily extend ISA with a much smaller amount of description than using conventional environments. Implementing RISC-V vector extension on ASIP Meister contributes to improve the performance of the processor and enable designers to adjust the number of instructions freely. This paper shows that challenging to construct of the HW processor based RISC-V ISA including vector extension, confirms the efficiency of the processor such as area, delay, the number of execution cycles, design man-hour when extending ISA, and checks the validation as the processor has RISC-V ISA.

The organization of this paper is as follows. Section I I explains the features of RISC-V ISA and ASIP Meister. Section Ⅲ describes specification of the processor

proposed by this study. Section IV shows and discusses the experimental results. Finally, section V concludes this paper.

II. Background

This section explains basic knowledge of this study such as RISC-V ISA specification, automatic generation of processor.

A. RISC-V

This section explains about RISC-V overview, ISA specification adopted on this paper, and ecosystems.

A.1. ISA Overview

RISC-V is composed of basic ISA called "I" and extended ISAs. ISA "I" has basic integer instructions such as mathematical operations, logical operations, load, store, and jump operations. Extended ISAs consist of multiplication and division ISA "M", atomic operation ISA "A", floating operation "F", compressed ISA "C", vector ISA "V", and so on. "I" ISA is essential and users can add desired extended ISAs. Moreover, RISC-V has custom instruction fields for users to add instruction freely. In summary, RISC-V has a much simpler ISA and has higher extensibility than conventional ISAs.

A.2. Target Instruction Set

RISC-V has 32-bit and 64-bit and 128-bit addressing mode instruction sets. This study focuses on 32-bit addressing mode instruction called RV32. This paper targets RV32IMV that has basic integer ISA "I" , multiplication and division extension "M", and vector extension "V". RV32V is vector architecture focusing on data level parallelism.

Figure 1 shows the structure of register files RV32V has. Maximum feature of RV32V is a register file dedicated to vector operations apart from the basic general-purpose register for integer operations. From this, size and data type of each vector register is configurable suitable for implementation forms by using `vl` and `vtype` registers unlike the SIMD (Single Instruction Multiple Data) operator which decides the encoding of instructions. RV32V has

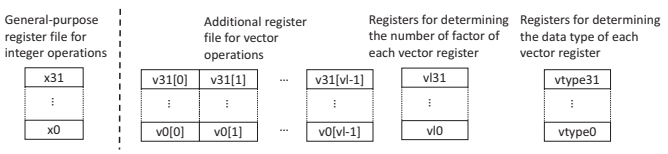


Fig. 1. Structure of register files of RV32V

other important features such as configurable vector registers, predications, and stride and indexed loads/stores. Specification of RV32V is explained in more detail in Section III.

A.3. Conventional ecosystems and HW tools

One github repository riscv-tools is free and avalable to anyone, and has almost RISC-V SW tools such as compilers, ISA simulator, and operating system. riscv-tools are easy to extend ISA by defining instruction opcodes, operands, and behavior descriptions.

As well, many RISC-V HW implementations are proposed such as Rocket-Chip, Boom, VexRiscv, and so on. However, all of these HW implementations are separated from riscv-tools. For extending ISA of conventional RISC-V HWs, you need to modify HDL directly. ISA extension by editing HDL requires many places to rewrite in programs, for examples of instruction definitions, function blocks, state machines. For that reason, instruction extension for RISC-V HW has problems that require high expertness on HDL and huge man-hours for code analysis and rewrites.

Hence, the purpose of this paper is to implement a RISC-V based HW processor whose ISA can be extended with much less amount and simpler description than conventional RISC-V HWs. For achieving this purpose, this study implements and evaluates RISC-V based HW processor using ASIP Meister.

B. ASIP Meister

ASIP Meister is an interactive tool for generating HDL more easily. This section explains processor design method using ASIP Meister.

B.1. Processor design flow

Processor design flow on ASIP Meister is composed of 8 steps as follows.

Definition of Architecture spec. defines the number of pipeline stages and so on.

Resource Declaration defines HW resources for using your processors such as ALU, storage, and so on. ASIP Meister uses parameterized resource models called Flexible Hardware Model (FHM). FHM are HW resources whose parameters can be changed. You need to construct new FHM which be adapted to your processor.

Definition of storage spec. sets the specification of register, register file, and memories.

Definition of IO interfaces sets entities and IO port of processor.

Format: ABSO rd, rs1, rs2

Inst. type	MSB	LSB	Field Type	Field Attr	Value
R	31	25	opcode	binary	0000000
I	24	20	operand	name	rs2
S	19	15	operand	name	rs1
U	14	12	opcode	binary	000
B	11	7	operand	name	rd
J	6	0	opcode	binary	1010111

(a) Definition Opcodes , Operands, Formats of Instruction

Stages	Micro operation description

VARIABLE	`wire [31:0] source0;`
	`wire [31:0] source1;`
	`wire [31:0] result;`

IF	`FETCH()`

ID	`wire[31:0] temp0;`
	`wire[31:0] temp1;`
	`temp0 = GPR.read0(rs1);`
	`temp1 = GPR.read1(rs2);`
	`source0 = FWU1.forward(rs1,temp0);`
	`source1 = FWU2.forward(rs2,temp1);`

EXE	`wire [3:0] flag;`
	`wire [31:0] temp2;`
	`wire[31:0] reverse;`
	`<temp2, flag> = ALU.sub(source0, source1);`
	`reverse = ~temp2;`
	`result = (temp2[31]) ? temp2 : reverse;`
	`null = FWU1.forward1(rd.result);`
	`null = FWU2.forward1(rd,result);`

MEM	

WB	`null = GPR.write0(rd, result);`
	`null = FWU1.forward3(rd.result);`
	`null = FWU2.forward3(rd,result);`

(b) Micro Operation Description for each Pipeline Stages

Fig. 2. Instruction extension method using ASIP Meister

Instruction Definition defines instruction set and instruction type, opcode, and operands of each instructions. Figure 2 (a) shows an example of necessary description on this method. In this step, appropriate instruction type and values of opcode and operand are set.

Assembler generation ASIP Meister attaches meta assembler. ASIP Meister generate assembler description for meta assembler. This assembler is used for the experiment of this paper.

Definition of micro operation definition describes the micro operation description of all instructions for each pipeline stages. Micro operation description needs to be described by processor description language [8]. Figure 2 (b) shows the example of necessary micro operation description for defining ADD instruction that adds two register values on this method. Focusing on description `source0 = GPR.read0(rs1)` in Fig. 2(b), "GPR" is HW resource of register file, "read0" is function GPR has, and "rs0" is one of operands that means the first source register. Like this example, description referring to defined HW resources and functions of them are written.

Generation HDL For these information, the synthesize model written by HDL are generated.

For extending instruction set, designers write instruction definitions and micro operation descriptions. This study uses ASIP Meister for constructing a RISC-V based processor.

III. Implementation

This section explains the implementation of processor.

A. Implemented Processor Overview

Figure 3 shows the construction of processor implemented in this paper. The number of pipeline stages is 5 stages composed of IF (Instruction Fetch), ID (Instruction Decode), EXE (Execution), MEM (Memory access), and WB(Write Back). In IF stage, instructions are fetched. In ID stage, instructions are decoded through reading operands of instructions and general purpose register files. In EXE stage, calculations are executed using ALU, multiplier, shifter, and comparator. In MEM stage, memory reads/writes are executed through accessing a data memory. In WB stage, register writes are executed.

Instruction register, instruction memory, and program counter are not different from basic integer processor because instructions are issued one by one in this processor too. Register file can be treated as both one register (32bit) and vector (32bit*4) and arithmetic units can execute calculations in 4-way parallels. Predication register file was constructed to implement predication function of RISC-V "V" extension. Unfortunately, data memory cannot be accessed in 4-way parallel currently because only

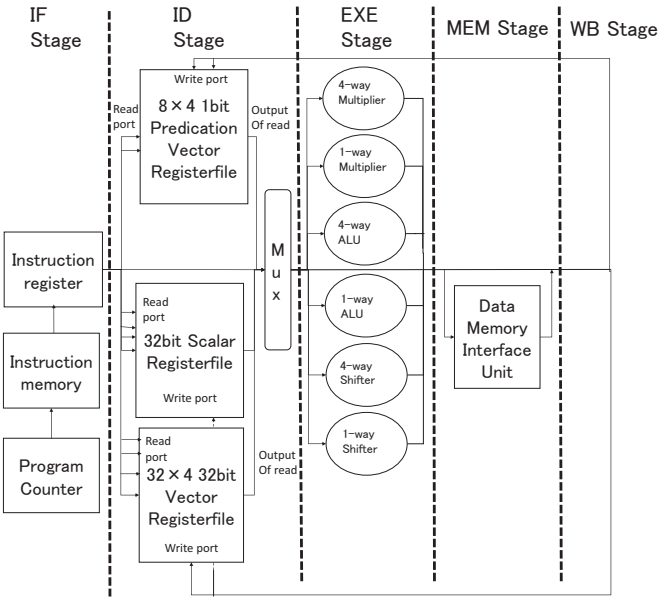


Fig. 3. Processor overview

one data bus between data memory in CPU and data memory interface unit can be set in ASIP Meister. In conclusion, This processor can execute register reads/writes and calculations in 4-way parallel.

B. Implementation instruction sets

This study challenges to implement RV32IMV that 32-bit addressing mode instruction sets which are composed of basic integer, multiple, and 4-way vector instructions.

Table I shows all implemented integer and multiple instruction sets. `rs1` and `rs2` mean source scalar registers. `rd` means a destination scalar register. `imm` means an immediate. `Mem` means a data memory. `pc` means a program counter. `(i)` means that the second source `rs2` replaces the immediate `imm`. `(u)` means that operands in instructions are read as unsigned values. Table. II shows all vector instruction sets. `vs1` and `vs2` mean source vector registers. `vd` means a destination vector register.

Vector extension of this study inherits RV32IM instructions such as integer addition, subtraction, logical operations, multiplication, and shift operations. This study excludes other instructions of vector extension such as floating point and atomic operations. RV32V instructions are classified into the type of two source registers by suffixes `.vs|.vv|.sv`. In the case of `.vs`, two registers are vector and scalar. Scalar is the first factor of vector registers. In the case of `.vv`, vector and vector. There is `.sv` type in asymmetric operations such as subtraction and shifts.

Load/store instructions of RV32V have three types such as sequential, stride, and indexed. First scalar source register of all load/store instructions is the base address. In sequential load, if first register value is 1024, 4 sequential data from 1024 address are loaded. Second source register of stride loads is scalar to which is address interval to refer. Data at address per stride value from base address

Inst. name	Behavior
add(i)	rd = rs1 + rs2
mul	rd = (rs1 * rs2)[31:0]
mulh(u)	rd = (rs1 * rs2)[63:32]
and(i)	rd = rs1 and rs2
or(i)	rd = rs1 or rs2
xor(i)	rd = rs1 xor rs2
sub(i)	rd = rs1 - rs2
sll(i)	rd = rs1 << rs2
sra(i)	rd = rs1 >>> rs2
srl(i)	rd = rs1 >> rs2
slt(i)(u)	rd=(rs1<imm)
l(b|h|w)(u)	rd = Mem[rs1]
s(b|h|w)(u)	Mem[rs1] = vs1
beq	if(rs1==rs2) pc=pc+imm
bne	if(rs1!=rs2) pc=pc+imm
blt(u)	if(rs1<rs2) pc=pc+imm
bge(u)	if(rs1>=rs2) pc=pc+imm
jal	pc=pc+imm
jalr	pc=rs1
lui	rd=imm[31:12],12'b0
auipc	rd=pc+imm[31:12],12'b0

TABLE I
Implemented Instruction Set("I" and "M")

Inst. name	Behavior
vadd(.vv|.vs)	vd = vs1 + vs2
vmul(.vv|.vs)	vd = vs1 * vs2
vand(.vv|.vs)	vd = vs1 and vs2
vor(.vv|.vs)	vd = vs1 or vs2
vxor(.vv|.vs)	vd = vs1 xor vs2
vsub(.vv|.vs|.sv)	vd = vs1 - vs2
vll(.vv|.vs|.sv)	vd = vs1 << vs2
vsra(.vv|.vs|.sv)	vd = vs1 >> vs2
vsrl(.vv|.vs|.sv)	vd = vs1 >> vs2
vld(s|x)	vd = Mem[rs1]
vst(s|x)	Mem[rs1] = vs1
vpeq(.vv|.vs)	vd = vs1 == vs2
vpne(.vv|.vs)	vd = vs1 != vs2
vplt(.vv|.vs)	vd = vs1 < vs2
vpge(.vv|.vs)	vd = vs1 >= vs2
vpand	vd = vs1 and vs2
vpandn	vd = vs1 nand vs2
vpor	vd = vs1 or vs2
vpxor	vd = vs1 xor vs2
vpn	vd = not vs1
vpswap	vd = vs1, vs1 = vd
vmov.vv	rd = imm[31:12]&12b0
vextract.vs	rd = rs1 or (rs2|imm)
vmerge.vv	vd = (vp1)vs2[vs1]
vselect.vv	vd = vs1[vs2]
vsetdcfg	vtype = imm

TABLE II
Implemented Instruction Set(Vector Extension)

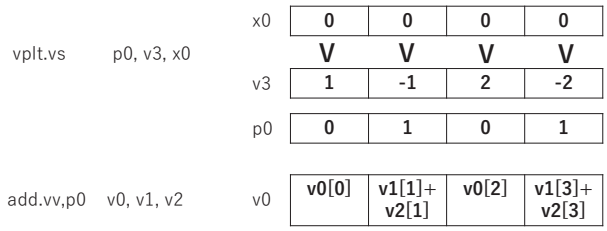
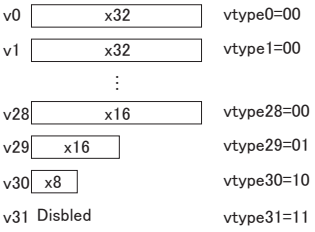


Fig. 4. Predication example



vtype value	register bit width
00	32 bit
01	16 bit
10	8 bit
11	Disabled

Fig. 5. Configurable register type

are loaded. For example, when base address value is 1024 and stride is 12, 4 data at 1024, 1036, 1048, 1060 addresses are loaded. Finally, in indexed load, second source register is vector index the address to refer. Data at address value base address adds second register are loaded. For example, when base address value is 1024 and indexes are 12, 8, 0, and 4, 4 data at 1036, 1032, 1024, 1028 addresses are loaded.

Unfortunately, these loads/stores cannot be executed in parallel now. We will consider to construct an unit for accessing memories in parallel.

RV32V supports a predication function. Figure 4 shows an example of the predication usage. There are 8 predication 1-bit vector registers have 4 scalars in this processor. Initial values of all predication registers are "1". Source registers of predication instructions are vector registers. Like Fig. 4, only factors in predication register that the conditional equations hold in vector registers are set "1". In vector and vector operations, only factors corresponded predication register is "1" are executed each operation.

RV32V has 32 operation vector registers and there are vector length registers `vl` for each operation register and maximum vector length register `mvl`. Data type and length associates with vector registers in programs dynamically.

Data type of each vector register is configurable by `vsetdcfg` instruction and system registers `vtype`. Figure 5 shows function of configurable register type. `vtype` exists the same number as vector registers. Right table in Fig. 5 shows `vtype` encodings for register bit width. By executing `vsetdcfg` instruction, we can change bit width at every vector register.

`vselect` is an instruction can collect elements in first

source vector register at the location designated by the second source vector register and generate new vector register. `vmerge` is an instruction that collects the element from the first source register if the predication register is "1", and collects from the second register if "0". `vextract` is an instruction that loads the first source vector register from the address designated by the second scalar register. `vmov.vv` is an instruction that moves values in one vector register. From the above, there are unique instructions that take advantage of vector registers. This paper challenged to construct above instructions by implement appropriate HW resources FHM and micro operation description for ASIP Meister.

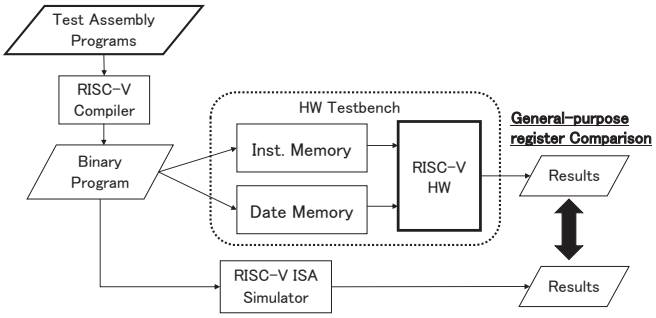


Fig. 6. Environment for evaluation of processor

IV. Evaluation

This section explains evaluation methods and results. We evaluated the basic RV32IM implementation.

A. Evaluation contents

This paper conducted 3 evaluations as follows. First, evaluate man-hours when adding unique instructions in a way to compare the amount of required modified description between HDL and ASIP Meister. Next, design quality of processor is evaluated by using 45nm open cell library and Design Compiler. Finally, processor ISA implementation is checked in a way to execute sufficient number of assembly test programs.

B. Evaluation of man-hours

We conducted ABSSUB and SWAP instructions in addition to basic ISA. When ABSSUB instruction executes, the absolute value of the difference of two source registers is stored to destination register. When SWAP instruction executes, data is loaded after endian is swapped.

First, we counted the line number of the required change descriptions when we added above instructions by using ASIP Meister. Similarly, we counted the line number of the required change description in Hardware Description Language (HDL).

Table III shows the compared result of required change descriptions. When ABSSUB instruction is added, the number of change description is reduced by 93.5%. When SWAP, the number is reduced by 94.5%. From the above result, design man-hours for adding instructions are greatly reduced.

C. Validation of generated processor

We confirmed the correct behavior of the implemented processor. Figure 6 shows the environment for this evaluation. Test programs consist of about 400 assembly programs that execute the same instruction in succession or all instructions randomly. Used general purpose registers are also set randomly. These test programs are executed in both RISC-V software riscv-tools and the implemented hardware. After programs are executed, we checked that values of all general purpose registers in software simulator and implemented hardware are same. As evaluation result, we confirmed that all test programs are passed without errors.

D. Evaluation of design quality

We evaluated the area and delay time of processors for benchmark by doing logical synthesis using Design Compiler by Synopsys and 45nm open cell library. We set delay time constraint to 200MHz (5ns/clock) and 400MHz (2.5ns/clock). We also evaluated VexRiscv [5] that implement RV32IM as well as processor of this paper because VexRiscv is one of implementations of RV32IM from HDL. Table IV shows the evaluation result of area and delay time of processor. Result shows that designed processor is 1.3 times in area and 1.4 time in delay time as VexRiscv respectively. Constructing processors on ASIP Meister may increase the number of multiplexers depending on descriptions. We assume that performance of this

TABLE III
Comparison of required change descriptions

	ASIP Meister (lines)	HDL(lines)
ABSSUB	22	341
SWAP	35	615

TABLE IV
The evaluation result of area and delay time

Processor	Frequency	Area [μm^2]	Gate number	delay time[ns]
ASIP Meister	200MHz	52,632	28,037	4.92
ASIP Meister	400MHz	53,272	28,378	2.42
VexRiscv	200MHz	37,245	19,841	3.70
VexRiscv	400MHz	37,332	19,887	1.50

processor can be improved by optimizing HDL codes, constructing RISC-V "C" extension.

V. conclusion

This paper proposed hardware developing environment for RISC-V 32-bit addressing integer architecture RV32IM and challenged to construct processor for RISC-V vector extension RV32V. About 400 test assembly programs are executed using implemented RV32IM based processor and we confirmed that all programs are passed without errors. Moreover, we added ABSSUB and SWAP instruction to processors and evaluated the amount of required change descriptions in ASIP Meister and HDL. As a result, in comparison with HDL, we can add these instructions by 6 percent number of change descriptions. On the other hand, in benchmark test using Design Compiler and Open Cell Library, result of processor of this paper is 1.3 times in area and 1.4 times in delay time as those of VexRiscv.

Future work includes completing processor design with vector extension and efficient processor design environment. Currently, we have not finished to implement data memory reads and writes in parallel and configurable vector register type function. RISC-V includes other convenient extensions such as compressed extension "C", floating point extension "F", and so on. In particular, "C" extension is important for performance improvement because "C" is instruction set that can execute two instructions at the same time by dividing the 32-bit instruction register into 16-bit. Therefore, we need to improve performance of processor in terms of area, execution cycle time, and so on. As a mean of performance improvement, we consider to optimize descriptions such as HDL and micro operation description and implement "C" extension.

The second work is to implement configurable environment that can add instructions to riscv-tools and proposed hardware processor at the same time by less number of change descriptions automatically. The method of this paper enable designers to extend ISA of HW processors by adding descriptions of instruction on behavior level. However, it is not possible to simulate processors with HW only. For simulation, SW of compiler, simulator, etc. are neccessary. Therefore, we aim to enable designers to extend ISA of both HW and SW simultaneously. We consider to construct an environment for configurable processor using riscv-tools and ASIP Meister. SW tools riscv-tools are already configurable because we can extend instruction set of all SW tools by change a single instruction definition. The goal is to build a configurable development environment that can simultaneously extend SW and HW instruction definitions from a single simple description.

Acknowledgments

Part of this research is supported by JSPS research grant JP17K00077. This research was conducted with the cooperation of Synopsys, Inc. through the University of Tokyo Large-scale Integrated Systems Design Education and Research Center.

References

[1] David Patterson and Andrew Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[2] "GitHub - riscv/riscv-tools: RISC-V Tools (GNU Toolchain, ISA Simulator, Tests)," Last access date:2019/06/21. [Online]. Available: https://github.com/riscv/riscv-tools

[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbeltm, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim and John Koenig, "The rocket chip generator," Electrical Engineering and Computer Sciences Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[4] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson and Krste Asanovi, "BOOMv2: an open-source out-of-order RISC-V core," Electrical Engineering and Computer Sciences Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, 2017.

[5] "GitHub - SpinalHDL/VexRiscv: A FPGA friendly 32 bit RISC-V CPU Implementation," Last access date:2019/06/21. [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[6] Masaharu Imai, Yoshinori Takeuchi, Keishi Sakanushi and Nagisa Ishiura, "Advantage and possibility of application-domain specific instruction-set processor (ASIP)," *IPSJ Transactions on System LSI Design Methodology*, vol. 3, pp. 161–178, 2010.

[7] Takeshi Shiro, Masaaki Abe, Keishi Sakanushi, Yoshinori Takeuchi and Masaharu Imai, "A processor generation method from instruction behavior description based on specification of pipeline stages and functional units," in *Asia and South Pacific Design Automation Conference*, 2007, pp. 286–291.

[8] Prabhat Mishra and Nikil Dutt, *Processor Description Languages Applications and Methodologies*. Morgan Kaufmann Publishers, 2008.