Regular Paper

Software Development Tool Generation Method Suitable for Instruction Set Extension of Embedded Processors

Takahiro Kumura,[†1,†2] Soichiro Taga,[†3] Nagisa Ishiura,[†3] Yoshinori Takeuchi[†2] and Masaharu Imai[†2]

This paper proposes a method of software development tool generation suitable for instruction set extension of existing embedded processors. The key idea in the proposed method is to enhance a base processor's toolchain by adding plugins, which are software components that handle additional instructions and registers. The proposed method can generate a compiler, assembler, disassembler, and instruction set simulator. Generated compilers with the plugins provide intrinsic functions that are translated directly into the new instructions. To demonstrate that the proposed method works effectively, this paper presents an experimental result of the proposed method in the study of adding SIMD instructions to the embedded microprocessor V850. In the experiment, by using intrinsic functions, the compiler generated good code with only 7% increase in the number of instructions against the hand-optimized assembly codes.

1. Introduction

Application specific instruction set processors (ASIPs) are increasingly employed in embedded systems for multimedia and mobile wireless applications because they are programmable and provide better performance for applications. The flexibility of software allows designers to make late design changes or additions, and the instruction sets tuned for target applications improve performance. To design ASIPs, people recently use design tools to generate hardware description language (HDL) source codes or software development tools. There have been many previous approaches in terms of exploring both the processor architecture and instruction set architecture (ISA) for ASIPs using architecture description languages (ADLs) such as nML, LISA, and EXPRESSION[1)–4)].

One of the main challenges in making the best use of ASIPs is to provide software development tools such as assemblers and C compilers at the early stages of architecture design. To enable a good C compiler to be provided at the early stages of architecture design, most ASIPs consist of application specific functional units and a base processor, which has a fundamental instruction set. Such ASIPs consisting of application specific functional units and a base processor are accepted in the market and are commercially available from several sources such as Tensilica. The design tools for the ASIPs have also been released by CoWare Inc., Target Compiler Technologies, and ASIP Solutions.

Looking closer at tool generation processes, most of these ASIP design tools and previous related works have focused on generating whole part of the tools from scratch and have not considered reusing existing tools for base processors or manually improving the generated tools. If existing processors are used as the base processors for the ASIPs, a complete set of tools is already available. The tools would include a hand-optimized compiler or a simulator equipped with features such as particular performance profiling. These features may not be always available on the tools generated by using the conventional methods.

Another important aspect on the tool generation for ASIPs is fundamental toolchain used for the tool generation. While most of ASIP design tools and previous related works[3)–5),8)] have been developed based on their own compilers, simulators, or binary tools, some of the conventional approaches[6),7),9),10),12)] have used the GNU toolchain, which is an open-source and a de-fact toolchain in the field of embedded software development. Since the GNU toolchain supports many kinds of processors, it is very suitable to generate software development tools for the ASIPs based on existing embedded processors. However, excepting the tool generation method for the Xtensa[12)], the conventional tool generation methods targeted at the GNU toolchain have mainly focused on part of the toolchain like GNU Binutils not the whole part. Although the tool generation method for the Xtensa can generate all the tools based on the GNU toolchain, its target processor is limited to the Xtensa and does not support any other processor architectures. Therefore, we determine that a new method of generating tools based on the

†1 NEC Corporation
†2 Osaka University
†3 Kwansei Gakuin University

GNU toolchain is required.

This paper proposes a new method of software development tool generation for instruction set extension of existing embedded processors. The key idea behind our method is to enhance a base processor's toolchain by adding plugins, which are software components that handle additional instructions and registers added to the base processor. Plugins are generated from the specification information of the additional instructions and registers. The only modification that needs to be made to the base processor's toolchain is to provide the sockets to accept the plugins. This paper overviews our approach in terms of how the specifications for additional resources are described, what plugins are generated to handle the additional resources, and what modifications need to be made to the base toolchain to accept plugins. In addition, the experimental results on the study of adding SIMD extension to the V850 microcontroller are presented.

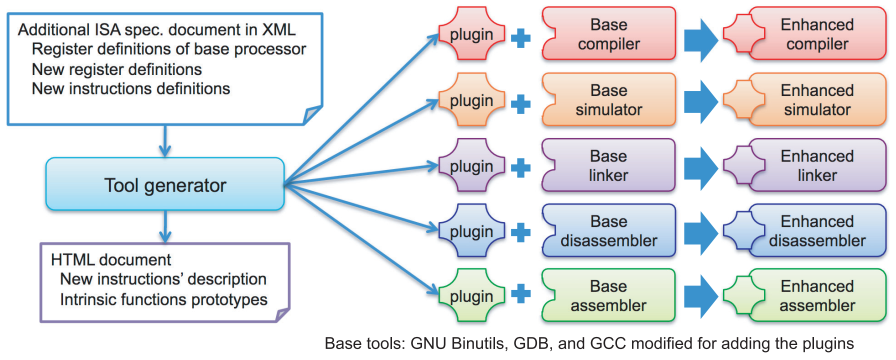
The rest of this paper is organized as follows. Section 2 introduces our proposed tool generation method. Section 3 gives an outline of the ISA description language for the proposed tool generation. Section 4 describes an experimental result. Section 5 discusses the difference between the proposed method and related work. Finally, Section 6 conludes this paper.

2. Software Development Tool Generation Using Plugins

Our goal is to provide an efficient way of generating the software development tools for instruction enhanced processors, which are based on existing microprocessors. To achieve this goal, we chose a method using plugins, where software components are added as plugins to the existing toolchain to handle additional features, e.g., for parsing, encoding, or decoding new instructions. The plugins allow us to continue to use the existing toolchain and to enhance its functionalities for new instructions by adding more plugins. As a result, our method can generate a compiler, assembler, disassembler, linker, and simulator for instruction enhanced processors.

2.1 Tool Generation Flow

Figure 1 shows the concept flow underlying the proposed method for tool generation. The flow begins with a specification document written in extensible markup language (XML). The XML document contains an additional ISA specification for the target processor. Here, we have assumed that the target processor's ISA consists of the base processor's instruction set and the additional instruction set which is written in the XML document. The additional ISA specification includes the base processor's name, the base processor's register information, and the specifications for new registers and instructions added to the base processor. However, it does not include the definition of the base processor's instruction set. Designers write this XML document for their target processor and feed the document to the tool generator.



Additional ISA spec. document in XML
Register definitions of base processor
New register definitions
New instructions definitions
Tool generator
HTML document
New instructions' description
Intrinsic functions prototypes
plugin
Base compiler
Enhanced compiler
Base simulator
Enhanced simulator
Base linker
Enhanced linker
Base disassembler
Enhanced disassembler
Base assembler
Enhanced assembler
Base tools: GNU Binutils, GDB, and GCC modified for adding the plugins

Fig. 1 Concept flow underlying tool generation.

The tool generator, then, generates the plugins to be added to the software development tools of the base processor (base tools). The base tools are modified once in advance so that they have sockets that can be connected to the plugins, the plugins can be integrated with the base tools, and the integrated software can become enhanced tools for the target processor. The enhanced tools with the plugins have four main features of assembling, disassembling, relocating, and executing the new instructions, and they also provide intrinsic functions translated by the enhanced compiler directly into the new instructions.

2.2 Structure of the Generated Plugins

What distinguishes our proposed method from conventional methods is to propose plugins produced from templates and parameters and to easily add the support for new instructions to existing toolchain for base processors. This work is a very complicated task and requires deep knowledge about the GNU toolchain if you do manually the same things. Since the templates are designed so as to support a variety of instructions, the specification of desired new instructions can be described in a simple format. In addition, generating plugins from templates and parameters increases readability of the plugins and decreases possibility of involving errors in the plugins.

Each command of the GNU toolchain such as gas has an internal flow in which it identifies one instruction and another extracted from input programs and performs particular operations on the instruction. The operations performed to instructions can be composed of several fundamental sub-operations. Field structures and syntaxes of instructions determine which sub-operation is used and how sub-operations are mixed. The proposed method represents a combination of sub-operations as a template and parameters, which will become a plugin. Templates are code fragments commonly used for every instruction, and parameters are a variety of information items on instructions. The parameters drive the templates to perform particular operations. Such generation scheme allows us to easily understand the structure of the generated plugins and to verify if the generated plugins work correctly. If different control code fragments were generated for different instructions, it could make the tool generator complicated and make it difficult to verify the generated plugins since the generated plugins would differ greatly depending on instructions.

Figure 2 shows the process to generate the plugins for the GNU toolchain in detail. The tool generator generates the plugins by adding data arrays and code fragments into pre-described template files. While the template files are commonly used, the data arrays and code fragments vary depending on XML documents and instruction behavior description files.

Template files have common functions used to handle any new instructions and registers. For example, the common functions include functions for encoding, decoding, assembling, and disassembling new instructions. These common functions perform their operations according to the information on new instructions and registers. The information is generated into plugins as structure data arrays, which are used as a database on new instructions and registers.

For GNU Binutils and GDB, instruction information and register information required for assembling, disassembling, linking and executing are generated as

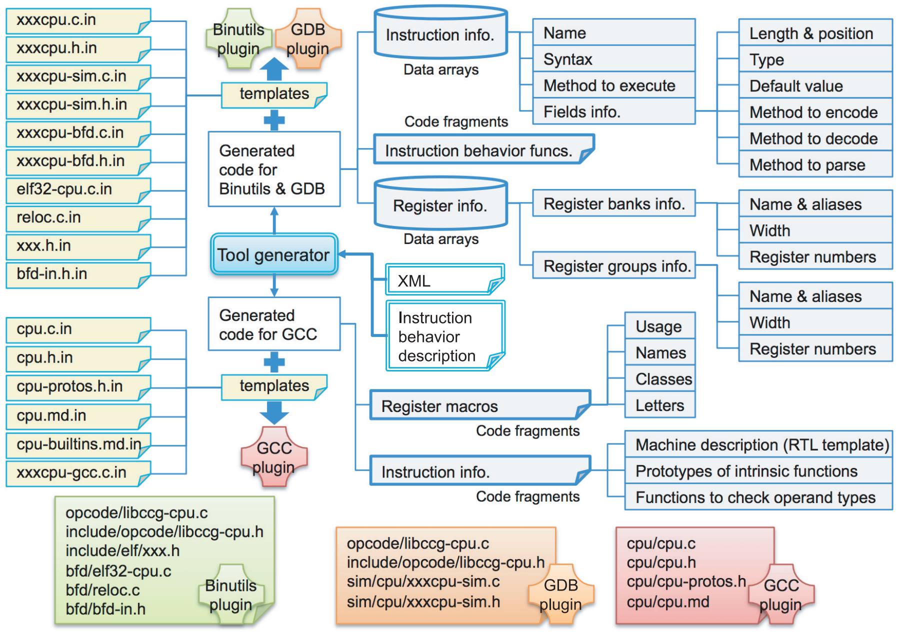


Fig. 2 Generating plugins for GNU Binutils, GDB, and GCC.

data arrays, and the body of instruction behavior functions are generated as code fragments. The information for each of instructions includes a name, syntax in assembly language, pointer of the method to execute the instruction, and the information on the instruction fields. Each of the instruction fields information further includes its length and position in the instruction codeword, a filed type to specify what the field represents (register, immediate, or operation code), a default value expression of the field, and three pointers to methods for encoding/decoding/parsing.

For the GCC, register macros and instruction information are generated as code fragments. On the GCC, the register-related macros give the registers information such as their usage, names, classes, and letters to represent registers. The instruction information includes machine description, prototype definitions of the intrinsic functions, and functions to check operand types for the instructions.

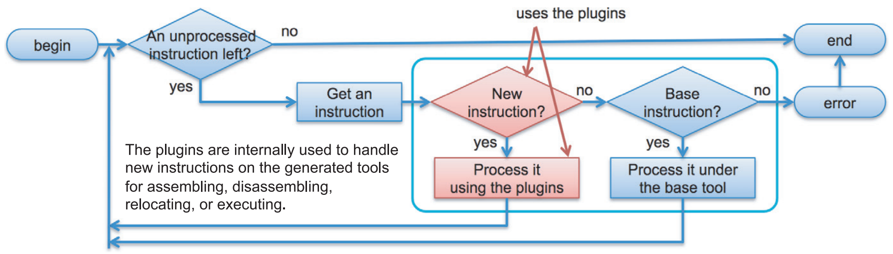

begin
An unprocessed instruction left?
no
yes
uses the plugins
end
error
Get an instruction
New instruction?
no
yes
Base instruction?
no
yes
Process it using the plugins
Process it under the base tool
The plugins are internally used to handle new instructions on the generated tools for assembling, disassembling, relocating, or executing.

Fig. 3 Tool internal flow for enhanced assemblers, disassemblers, linkers, and simulators working with plugins.

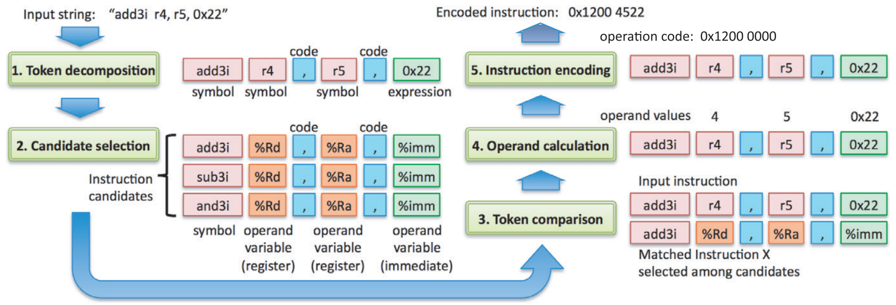

Input string: "add3i r4, r5, 0x22"
1. Token decomposition
2. Candidate selection
3. Token comparison
4. Operand calculation
5. Instruction encoding
Encoded instruction: 0x1200 4522
operation code: 0x1200 0000
operand values 4 5 0x22
Input instruction
Matched Instruction X selected among candidates
Instruction candidates
symbol
code
expression
operand variable (register)
operand variable (immediate)

Fig. 4 Assembling instructions on the plugins.

2.3 Internal Tool Flow Working with the Plugins

The base tools and the plugins work together on the enhanced tools for the target processor, as shown in **Fig. 3**, which is a simplified flow chart on how the plugins are employed to process each of instructions for the target processor. On the enhanced tools for the target processor, instructions are processed through either a conventional procedure or an additional procedure. First, the enhanced tools determine whether a given instruction is of new instructions added to the base processor or not. Then, the enhanced tools process the instruction through an approriate procedure. The conventional procedure is a path to handle the instructions of the base processor, and the additional procedure, which is a feature provided by the plugins, is a path to handle the new instructions added to the base processor. In order to make the enhanced tools behave in this way, the base tools need to be modified at once so as to work with the plugins.

2.4 Assembling and Encoding New Instructions

Here, how the enhanced assemblers process new instructions is outlined. The parser plugin built in the enhanced assemblers can handle any one of three instruction syntax styles in assembly language: a mnemonic style, a function style, and an algebraic style. In the mnemonic style, a mnemonic comes first and is followed by several operands. In the function style, input operands and an output operand are denoted respectively as function arguments and a return value. In the algebraic style, operators such as '+', '-', and '*' denote instructions' operations and '=' is used to specify a destination operand.

Mnemonic style:	`add3i  r4,r5,0x22`
Function style:	`r4=add3i(r5,0x22)`
Algebraic style:	`r4=r5+0x22`

The mnemonic style is the most major syntax style, and the latter two styles are used favorably for digital signal processors to make it easy to understand their assembly codes. Supporting these three styles contributes to increasing the freedom to choose syntax styles for readability.

The parser plugin processes new instructions as shown in the following five steps and **Fig. 4**. After assembling a new instruction, if any of operands refers to an unresolved symbol, the enhanced assemblers make relocation information for the unresolved symbol.

1. **Token decomposition:** The parser plugin breaks the input string into tokens, which are classified into either a symbol token, expression token, or a code token. In Fig. 4, there are 6 tokens.
2. **Candidate selection:** The parser plugin chooses instruction candidates which have the same number of tokens as in the input instruction. In Fig. 4, three instructions are selected as candidates.
3. **Token comparison:** For each of candidates, the parser plugin compares tokens at the corresponding position of the input instruction and a candidate, and find the instruction X which fits to the token pattern of the input instruction. In Fig. 4, the instruction `add3i` has the same token pattern as of the input instruction.

4. **Operand calculation:** The tokens corresponding to operand variables like %xxx in the instruction X are operands. The parser plugin calculates operand values from those tokens of the instruction X. For example, if a token corresponding to an operand variable represents a register name, the assembler translates the name to an operand value.
5. **Instruction encoding:** The parser plugin encodes the input instruction by using the operation code of the instruction X and the operand values of the input instruction.

2.5 Machine Description of New Instructions

The tool generator generates machine description of the new instructions. The machine description is the instruction patterns of the new instructions, and is used in the processor-dependent components of the GCC. The instruction patterns are written in the GCC's intermediate representation called the register transfer language (RTL). **Figure 5** shows an example of the generated machine description.

Normal instruction patterns usually have behavior description specified in their contents. With the behavior description given in the pattern, the GCC recognizes what kinds of operations the instruction performs. The generated instruction patterns, however, do not have well specified meaningful behavior, but just simple information that gives input and output operands. With the information on input and output operands given for machine description, the GCC can recognize which operand is used in which instruction and can also schedule instructions without meaningful behavior in machine description. Therefore, we decided to omit meaningful behavior from generated instruction patterns.

```
;; syntax:  myadd reg1, reg2, reg3
(define_insn "builtin_cpu_MYADD"
[
(set
  (match_operand:SI 0 "cpu_gpr_regs_operand" "=r")
  (unspec:VOID [
    (match_operand:SI 1 "cpu_gpr_regs_operand" "r")
    (match_operand:SI 2 "cpu_gpr_regs_operand" "r")
    ]  UNSPEC_BUILTIN_MYADD))
]
""
"myadd %1, %2, %0"
[(set_attr "length" "4")]
)
```

Fig. 5 A generated instruction pattern.

If a target processor has move instructions for new registers which do not exist in the base processors, the tool generator generates move instruction patterns from and to the new registers. The generated move instruction patterns have meaningful behavior description, so that the GCC can recognize that they perform move operations and the GCC can use them in code generation process.

2.6 Adding Intrinsic Functions on the GCC

In order to add intrinsic functions on the GCC, two macro functions `TARGET_INIT_BUILTINS` and `TARGET_EXPAND_BUILTIN` are used.
`TARGET_INIT_BUILTINS` represents the name of a function that performs initialization for intrinsic functions. For each of intrinsic functions, the initialization function makes registration of the information such as a return value type, arguments types, a function name, and an ID number. The tool generator generates statements for this registration of intrinsic functions. The generated statements are inserted into the function represented by the macro `TARGET_INIT_BUILTINS`.

`TARGET_EXPAND_BUILTIN` represents the name of a function that generates instruction patterns from given intrinsic functions. The function corresponding the macro `TARGET_EXPAND_BUILTIN` determines which instruction pattern should be used for a given intrinsic function and how arguments of the given intrinsic function should be translated into operands of the instruction pattern. The tool generator generates tables used for retrieving instruction patterns and their operand types from given intrinsic functions.

On the GCC, instead of intrinsic functions, there is another way to use new instructions added to the base processors, that is, inline assembly can also exploit the new instructions in C codes. Intrinsic functions, however, is better than inline assembly in terms of the following points.

- There is possibility that the GCC can schedule the instructions corresponding to intrinsics and optimize them if information on the instructions such as a latency and code size is given to the compiler. Inline assembly, however, has no means for better scheduling and optimization.
- The GCC can check types of output and input operands for intrinsic func-

tions. Inline assembly, however, can not do that.
- Although inline assembly is available only for its target processors, intrinsic functions could be emulated on non-target processors if emulation functions for the intrinsic functions were provided.

These reasons made us to choose intrinsic functions rather than inline assembly.

Several methods 14)–16) for automated instruction set extension have been reported so far, which generate compilers that can exploit the extended instructions without the need for modifying source codes of applications. Contrary to this, on the proposed method, the enhanced compiler does not automatically exploit the new instructions and programmers have to invoke intrinsic functions if they want to make the compilers to use the new instructions. However, changing algorithms or rewriting source codes for more speed is still important in many practical situations. Therefore, using intrinsic functions to accelerate target applications is a practical method.

3. Instruction Set Description Using XML

This section explains the additional ISA specification in our proposed framework. Our proposed tool generation flow begins with an XML document, which contains an additional ISA specification for the target processor. The XML provides a flexible and extensible framework for representing and structuring all kinds of data. In addition, XML is widely used in the World Wide Web community as a means of structured information exchange, and there are many software components and libraries that handle XML documents. In our case, an XML document is used to describe the specification of new registers and instructions to be added to the base processor.

The XML of the proposed method can describe not only ISAs of general RISC processors but also the following complex instructions:

(1) instructions that have an operand with a restriction, e.g., the operand must be an even number register.
(2) instructions in which an operand value is separated and placed into two different instruction fields. This tricky field arrangement might be used when there is not enough instruction field space for additional instructions.
(3) instructions in which a value calculated from an operand is placed into an instruction field. This calculation might happen when the least significant n-bits of an immediate operand is trimmed off before the operand is placed into an instruction field.
(4) instructions that use pair registers that consist of two contiguous registers.
(5) instructions that have two or more output operands or have many input operands (The number of total operands must be less than or equal to ten).
(6) instructions in which any combinations of the above ones are used.

Although these complex instructions do not appear in general RISC processors, in fact, they are used in our experiment based on the V850 processor described afterward. In addition, since the behavior of instructions is written in C language apart from XML documents, any kinds of operations of instructions can be described.

Here is a simple example in which the new instruction MYADD depicted in **Fig. 6** is added to a base processor. **Figure 7** shows an XML document for the simple example. The XML document contains:

(1) the base processor's nickname defined by the `<nickname>` tag,
(2) the base registers on the base processor and new registers added to it defined by the `<register_bank>` tag,
(3) new instructions defined by the `<insn>` tag,
(4) GDB register numbers defined by the `<gdb>` tag, and
(5) the file name containing the behavior of instructions defined by the `<behavior>` tag.

The behavior of the new instructions in the XML document is described in C language in a different file.

Since new definitions of instructions are important for generating the enhanced tools and they take up many lines in an XML document, we will explain how instructions and registers are defined in an XML document in the following sections.

3.1 Register Definition

Register information is described using three tags: `<register_type>`, Syntax: `MYADD reg1, reg2, reg3`

	31–27	26–16	15–11	10–5	4–0
Fields:	reg3	opc1	reg2	opc0	reg1

Name	Type	Bits	Description
reg1	GPR	5	Register index of GPR registers
reg2	GPR	5	Register index of GPR registers
reg3	GPR	5	Register index of GPR registers
opc0	opcode	6	Bit pattern to distinguish it from other instructions
opc1	opcode	11	Bit pattern to distinguish it from other instructions

Fig. 6 Instruction field structure of MYADD.

```
<Processor>
<nickname>cpu</nickname>
<register_type length="32">GPR_type</register_type>
<register_bank type="GPR_type" size="32"
               prefix="R" base="true">GPR</register_bank>
<insn_length>32</insn_length>
<insn>
  <mnemonic>MYADD</mnemonic>
  <syntax>MYADD %reg1, %reg2, %reg3</syntax>
  <field type="GPR"    length="5">reg1</field>
  <field type="opcode" value="0b11_1111" length="6">opc0</field>
  <field type="GPR"    length="5">reg2</field>
  <field type="opcode" value="0b111_1100_1000" length="11">opc1</field>
  <field type="GPR"    length="5">reg3</field>
  <input>
    <operand type="GPR" width="32">reg1</operand>
    <operand type="GPR" width="32">reg2</operand>
  </input>
  <output>
    <operand type="GPR" width="32">reg3</operand>
  </output>
  <description>
    This instruction calculates the sum of the contents of
    registers reg1 and reg2, and stores the result into register reg3.
  </description>
</insn>
<gdb>
  <regnum name="R0">0</regnum>
  <regnum name="R1">1</regnum>
  ....
  <regnum name="R31">31</regnum>
</gdb>
<behavior>cpu-isa.c</behavior>
</Processor>
```

Fig. 7 Example of an XML document with additional ISA specification.

`<register_bank>`, and `<register_group>`. The `<register_type>` tag defines a register-type name and a register length in bits. The `<register_bank>` tag defines a register bank, which consists of the same type of registers defined by the `<register_type>` tag.

The `<register_bank>` tag has several attributes, which are '`type`', '`size`', '`prefix`', '`base`', and '`letter`'. The '`type`' denotes a register type, which is one of the register types defined by the `<register_type>` tag. The '`size`' denotes the number of registers included in the register bank. The '`prefix`' denotes a prefix of register names. The prefix and a register index in the register bank become a register name. The '`base`' denotes whether the base processor has the register bank or not. The '`letter`', which is not used in Fig. 7, denotes a letter for the GCC to represent a register class.

The `<register_group>` tag, which is not used in Fig. 7, defines a register group that consists of several registers. The registers in a register group must be a register in any of the register banks and may belong to different register banks. The register banks and register groups defined by the tags `<register_bank>` and `<register_group>` are used as register operand types in `<insn>` tag.

3.2 Instruction Definition

Each new instruction is defined using the `<insn>` tag. The `<insn>` tag has several child tags in its content. Here, we overview important tags to define a new instruction. The `<mnemonic>` tag defines a mnemonic of the new instruction. Since the mnemonic is used as an ID of the new instruction, it must be different from other instructions' mnemonics. The `<syntax>` tag defines an instruction syntax, in which the operands of the instruction are denoted by names that begin from `%`, e.g., `%reg1` or `%reg2`. The instruction syntax may be formatted in any one of three styles: a style of mnemonic plus operands, a function style, or an algebraic style. The plugins are constructed to be able to accept these styles. The `<length>` tag defines the length of the new instruction in bits. If this tag is omitted, the default instruction length defined by the `<insn_length>` tag is used.

The `<field>` tag defines each of the instruction fields that are part of an instruction code word. If the new instruction's code word has many different fields, the designer writes `<field>` tags for all the fields. The `<field>` tags are supposed to be listed from the least significant bit (LSB) to the most significant bit (MSB). The `<field>` tag has several attributes, which are '`type`', '`subtype`', '`length`', and '`value`'. The '`type`' denotes an instruction field type, i.e., what the instruction field represents. The field type may be either a register, an immediate value, or an operation code. If the field type is a register, its register type name is specified in '`subtype`'. The '`length`' denotes the length of the instruction field in bits. The '`value`' denotes the numeric value of the instruction field for immediate values and operation codes. The numeric value can be represented as an expression, e.g., `0xF<<2`, and the values of fields can be referred to in the expression using field names.

When a field's value is represented as an expression using the values of other fields, the tool generator needs to know how to obtain the values of instruction operands from the values of instruction fields. In this case, we use `<disas>` tag, which is not used in Fig. 7. The `<disas>` tag denotes an instruction operand type and how to calculate its value from the instruction fields. The `<disas>` tag also has the same attributes that `<field>` tag has. The calculation expression is represented in the attribute '`value`'. If we can obtain an operand value directly from the corresponding field value, we do not need to use `<disas>` tag.

If the new instruction has an immediate operand in its syntax and if the name of the immediate operand is defined by `<field>` tag or `<disas>` tag, the tool generator creates a new relocation type for the immediate operand. A new relocation type is also created if the attribute '`value`' of the `<field>` tag includes a variable named '`cia`' (current instruction address) or '`nia`' (next instruction address).

The `<input>` and `<output>` tags define the input and output operands of the new instruction. Each of the input and output operands is defined by the `<operand>` and `<memory>` tags. The new instructions can have multiple input operands and multiple output operands. The tool generator uses the information given by these tags when generating the plugins of the compilers for the target processor. The prototypes of intrinsic functions and machine descriptions of the new instructions are defined according to the definition of the `<input>` and `<output>` tags. If a new instruction has an output operand, the intrinsic function corresponding to it returns the output operand. If a new instruction has multiple-output operands or no output operands, the intrinsic function corresponding to it becomes a void-type function, and the output operands are passed as arguments of the intrinsic function.

```
/* MYADD is a mnemonic defined by <mnemonic> tag */
/* syntax:  MYADD %reg1, %reg2, %reg3 */
behavior (MYADD)
{
  /* GPR: register bank name. */
  /* reg1, reg2, and reg3: register index */
  int32_t val1 = REG_read32 (GPR, reg1);
  int32_t val2 = REG_read32 (GPR, reg2);
  int32_t val3 = val1 + val2;
  REG_write32 (GPR, reg3, val3);
}
```
Fig. 8 Behavior description of instruction MYADD written in C language.

3.3 Instruction Behavior Definition

Designers describe the behavior of a new instruction in C language in a different file other than in an XML document. There is an example of a behavior description in **Fig. 8**, which is the behavior description for the new instruction defined in the XML document in Fig. 7. The behavior of a new instruction is described in a single function. The function begins with '`behavior` (*mnemonic*)' and has several unspecified arguments available in the function. The arguments are the values of instruction fields other than those of operation codes. Each of the arguments has the name defined by the `<field>` tag or `<disas>` tag.

There are three non-operational-code fields in Fig. 7: `reg1`, `reg2`, and `reg3`. They are all register operands and the same named arguments are available in the behavior function in Fig. 8. Designers can access registers and memory in the behavior function through functions such as `REG_read32(REGTYPE,IDX)` and `MEM_read32(ADDR)`. We use the macro `NIA_SET(ADDR)` to modify the program counter and the macro `CIA` to get the content of the program counter.

4. Experiment

Here, we explain the experiment on the tool generation using our method. The base processor we use in this experiment is the V850 microcontroller. The V850

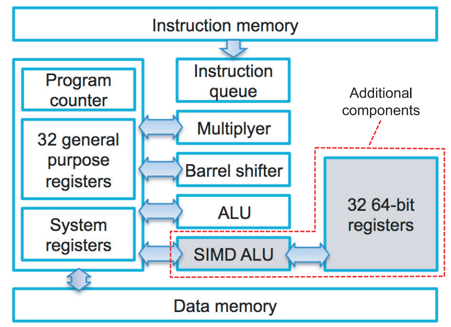


Fig. 9 Block diagram of the V850 processor with SIMD extension.

Table 1 Architecture summary of the V850 microcontroller with SIMD extension.

Base architecture	RISC processor for embedded systems. Harvard architecture. Single cycle instruction execution. Compact code size allowed by 2-byte insts. 32 32-bit general purpose registers.
SIMD extension	32 64-bit registers Data types: 16 bits × 4, 32 bits × 2, 64 bits × 1. Packed arithmetic insts. Load/store insts. Data type conversion. Logical operation.

Table 2 The code amount of the generated plugins for the SIMD extension.

Base tools	GNU Binutils 2.17 GDB 6.6 GCC 3.4.6
Number of instructions	179
Number of lines of the input XML file	5,934
Number of lines of the input behavior description	4,436
Number of lines of the assembler plugin	8,903
Number of lines of the simulator plugin	7,699
Number of lines of the compiler code fragments	13,496

is a RISC processor optimized for embedded systems [11]. In the experiment, our method generates a toolchain including a compiler for the target processor that consists of the V850 microcontroller and an SIMD extension.

Using the generated compiler and intrinsic functions for exploting the SIMD extension, we show code quality of the generated compiler. Note that in this experiment the inline assembly cannot be an alternative means to exloit the SIMD extension because the SIMD extension includes additional registers which cannot be addressed on the base compiler without any modification. Adding the plugins to the base compiler allows the compiler to handle the SIMD extension.

Figure 9 and **Table 1** show the target processor's block diagram and the summary of its SIMD extension. The SIMD extension includes 32 64-bit registers, an SIMD instruction set, and an SIMD functional unit which addresses 64-bit wide packed data. Available data types are 16 bits × 4, 32 bits × 2, and 64 bits × 1. The SIMD instruction set includes logical operations, data interleaving operations, data type conversions load/store operations, and packed arithmetic operations such as addition, subtraction, multiplication, and comparison.

4.1 Generated Toolchain for SIMD Extension

We translated the specification of the SIMD extension into an XML document and instruction behavior description, which are input files to our tool generator. Then, the tool generator generated a toolchain with the plugins in it for the target processor. **Table 2** shows the code amount of input files and the generated plugins. The number of SIMD instructions added to the V850 microcontroller is 179, and the input files to the tool generator have 10370 lines in total (XML: 5934, *.c: 4436). Although the code amount of the input files is not so small compared with the output files, which have 30098 lines in total, generating several tools such as a compiler, assembler, and simulator from simple specification documents is very beneficial.

4.2 Code Generation Using Intrinsic Functions

In order to investigate the compiler efficiency in terms of code generation using intrinsic functions, comparison between hand-optimized assembly codes and compiler-generated codes is discussed. For a number of basic signal processing functions such as filtering, sorting, and FFT, we developed two kinds of programs: (1) hand-optimized assembly codes using the SIMD extension, and (2) compiler-generated codes using intrinsic functions for the SIMD extension. Then the programs were profiled in terms of the number of executed instructions by using the generated simulator.

In order to build program (2), we developed a new C program using intrinsic functions from scratch for each of the signal processing functions. Since the programs of the signal processing functions written in normal C language have fewer lines than 100 lines, writing a new C program using intrinsic functions took a day or less per one signal processing function. However, for sorting and FFT which have more lines and require algorithm tuning suited to SIMD instructions, writing programs using intrinsic functions for them took a month per each. **Ta-**

Table 3 Lines of source codes: (a) original C code, (b) modified C code using intrinsic functions, (c) assembly code generated from (b), and (d) hand-optimized assembly code.

Function	(a)	(b)	(c)	(d)
32-bit normalization (N=64)	30	55	38	32
16-bit normalization (N=64)	30	57	41	35
32-bit minimum search (N=64)	20	66	41	34
16-bit minimum search (N=64)	20	79	49	41
32-bit maximum search (N=64)	20	66	41	34
16-bit maximum search (N=64)	20	79	49	41
32-bit LMS adaptive FIR (N=1,T=16)	25	104	71	63
16-bit LMS adaptive FIR (N=1,T=16)	18	102	70	62
32-bit IIR with scaling (N=1, B=2)	30	50	29	26
32-bit IIR with scaling (N=8, B=2)	33	56	38	30
32-bit IIR w/o scaling (N=1, B=2)	29	43	28	23
32-bit IIR w/o scaling (N=8, B=2)	33	48	34	28
16-bit IIR with scaling (N=1, B=2)	31	47	32	26
16-bit IIR with scaling (N=8, B=2)	41	59	39	31
16-bit IIR w/o scaling (N=1, B=2)	30	46	29	22
16-bit IIR w/o scaling (N=8, B=2)	42	62	35	28
32-bit FIR (N=1, T=16)	22	62	45	40
32-bit FIR (N=16, T=16)	33	99	71	61
16-bit FIR (N=1, T=16)	17	56	44	39
16-bit FIR (N=16, T=16)	33	80	60	62
32-bit complex FIR (N=1, T=16)	34	69	46	41
32-bit complex FIR (N=16, T=16)	41	82	56	45
16-bit complex FIR (N=1, T=16)	26	81	55	47
16-bit complex FIR (N=16, T=16)	33	77	55	41
32-bit bitonic sort (N=64)	127	224	215	197
16-bit bitonic sort (N=64)	127	188	156	141
32-bit complex FFT (N=256)	330	320	242	274
16-bit complex FFT (N=256)	320	338	274	292

N: Number of input samples, T: Number of filter taps, B: Number of biquad stages

ble 3 shows lines of source codes: (a) original C code, (b) C code using intrinsic functions, (c) assembly code generated from (b), and (d) hand-optimized assembly code. Regarding all signal processing functions in Table 3, code (b) using intrinsic functions has more lines than code (a) written in normal C language.

As an example, an FIR filtering function written in normal C language is shown in **Fig. 10** (a), its variant using intrinsic functions is shown in Fig. 10 (b), an assembly code generated from Fig. 10 (b) is shown in Fig. 10 (c), and a hand-optimized assembly code is shown in Fig. 10 (d). In Fig. 10 (b) and (c), intrinsic functions such as `_vxor()` and `_vld_dw_inc()` are translated into their corresponding assembly instructions such as `vxor` and `vld.dw`, respectively.

Figure 11 shows the increase in the number of executed instructions of the compiler-generated codes against the hand-optimized assembly codes. The number of executed instructions increases by only 7% on average when using intrinsic functions. Since compiler-generated codes without using intrinsic functions for the SIMD extension could increase to 900% on average against the hand-optimized assembly codes, only 7% increase in the executed instructions is acceptable. By using intrinsic functions, the compiler will replace variables with actual registers, ensuring better allocation, which is a bothersome work for programmers. In addition, the compiler-generated codes using intrinsic functions are nearly as good as the hand-optimized assembly codes. The proposed tool generation method made it possible to generate the compiler with such useful intrinsic functions.

The 7% increase could be caused by the following reasons:

Range checking before loops Compiler-generated codes have range checking before loops whether the condition to begin and continue the loops is met or not. Hand-optimized codes do not usually have such range checking because programmers know whether the range checking is unnecessary in their programs or not.

Redundant register transfers Compiler-generated codes have redundant register transfers on arguments and a return value of functions. Arguments given to functions are copied to local variables, and a local variable having a return value is also copied to a register at the end of functions. These register transfers sometimes may not be optimized and they remain as redundant move instructions.

A similar comparison we have done here was reported in the application note [17] for IDCT on Pentium4. The application note compared elapsed time for two types of IDCT implementation on Pentium4 with the SSE2 instruction set: one is written in C language using intrinsic functions and the other written manually in assembly language. The increase of the elapsed time when using intrinsic functions was 9% over that of the hand-optimized assembly code. This is a practical example of how to improve IDCT on Pentium4 using intrinsic functions instead of assembly language. Therefore, 7% increase on average in our examples

```
int32_t
fir16_single (
  int16_t *sig, int16_t *coef, int T, int offset)
{
  int i;
  int32_t y = 0;
  for ( i=0 ; i<T ; i++ )
  {
    int idx = i+offset;
    if (T <= idx) idx -= T;
    y += coef[i] * sig[idx];
  }
  return y;
}
```

(a) C code written in normal C language

```
int32_t
fir16_single_simd_intrinsic (
  int16_t *sig, int16_t *coef, int T, int offset)
{
  int i;
  int64_t w_3210, w_7654, w_ba98, w_fedc;
  int64_t x_3210, x_7654, x_ba98, x_fedc, acc;
  int32_t step_and_offset
    = (8<<16) | (2*(offset - (3 & offset)));
  int32_t buff_size = 2*T - 1;
  int64_t mod_param
    = _deposit_64 (buff_size, step_and_offset);
  int32_t align_byte = 2 * (3 & offset);
  short   loop_cnt = T >> 4;
  acc = _vxor (acc,acc);
  /* load coefficients */
  _vld_dw_inc (w_3210, coef);
  _vld_dw_inc (w_7654, coef);
  /* load input signals */
  _vld_dw_mod (x_3210, mod_param, sig);
  _vld_dw_mod (x_7654, mod_param, sig);
  _vld_dw_mod (x_ba98, mod_param, sig);
  _vld_dw_mod (x_fedc, mod_param, sig);
  for ( i = 0 ; i < loop_cnt ; i++ )
  { /* multiply and accumulate */
    int64_t xx;
    xx = _vconcat_b (align_byte, x_3210, x_7654);
    _vld_dw_inc (w_ba98, coef);
    acc = _vmsumad_h (acc, w_3210, xx);
    _vld_dw_mod (x_3210, mod_param, sig);
    xx = _vconcat_b (align_byte, x_7654, x_ba98);
    _vld_dw_inc (w_fedc, coef);
    acc = _vmsumad_h (acc, w_7654, xx);
    _vld_dw_mod (x_7654, mod_param, sig);
    xx = _vconcat_b (align_byte, x_ba98, x_fedc);
    _vld_dw_inc (w_3210, coef);
    acc = _vmsumad_h (acc, w_ba98, xx);
    _vld_dw_mod (x_ba98, mod_param, sig);
    xx = _vconcat_b (align_byte, x_fedc, x_3210);
    _vld_dw_inc (w_7654, coef);
    acc = _vmsumad_h (acc, w_fedc, xx);
    _vld_dw_mod (x_fedc, mod_param, sig);
  }
  return _extract_lo_64 (acc);;
}
```

(b) C code using intrinsic functions

```
_fir16_single_simd_intrinsic:
    mov -4,r10
    and r9,r10
    add r10,r10
    mov r8,r11
    add r8,r11
    movhi hi0(524288),r0,r12
    or r10,r12
    addi -1,r11,r13
    andi 3,r9,r9
    add r9,r9
    sar 4,r8
    sxh r8
    vxor vr3, vr3, vr3
    vld.dw [r7]+, vr9
    vld.dw [r7]+, vr8
    vld.dw [r6]%, r12, vr7
    vld.dw [r6]%, r12, vr6
    vld.dw [r6]%, r12, vr5
    vld.dw [r6]%, r12, vr4
    cmp 0,r8
    ble .L7
.L8:
    vconcat.b r9, vr7, vr6, vr2
    vld.dw [r7]+, vr0
    vmsumad.h vr9, vr2, vr3
    vld.dw [r6]%, r12, vr7
    vconcat.b r9, vr6, vr5, vr2
    vld.dw [r7]+, vr1
    vmsumad.h vr8, vr2, vr3
    vld.dw [r6]%, r12, vr6
    vconcat.b r9, vr5, vr4, vr2
    vld.dw [r7]+, vr9
    vmsumad.h vr0, vr2, vr3
    vld.dw [r6]%, r12, vr5
    vconcat.b r9, vr4, vr7, vr2
    vld.dw [r7]+, vr8
    vmsumad.h vr1, vr2, vr3
    vld.dw [r6]%, r12, vr4
    loop r8, .L8
.L7:
    mov.dw vr3, r10
    jmp [r31]
```

(c) assembly code generated from (b)

```
_asm_fir16_single_simd:
mov       r0, r10
andi      3, r9, r15
mov       r15, r14
shl       1, r14
sub       r15, r9
add       r9, r9
mov       r8, r11
sar       4, r11
movhi     8, r9, r12
mov       r8, r13
add       r8, r13
movea     -1, r13, r13
vld.dw      [r7]+, vr10
vld.dw      [r7]+, vr11
vxor        vr4, vr4, vr4
vld.dw      [r6]%, r12, vr6
vld.dw      [r6]%, r12, vr7
vld.dw      [r6]%, r12, vr8
vld.dw      [r6]%, r12, vr9
.L1:
vconcat.b r14, vr6, vr7, vr14
vld.dw      [r7]+, vr12
vmsumad.h vr14, vr10, vr4
vld.dw      [r6]%, r12, vr6
vconcat.b r14, vr7, vr8, vr14
vld.dw      [r7]+, vr13
vmsumad.h vr14, vr11, vr4
vld.dw      [r6]%, r12, vr7
vconcat.b r14, vr8, vr9, vr14
vld.dw      [r7]+, vr10
vmsumad.h vr14, vr12, vr4
vld.dw      [r6]%, r12, vr8
vconcat.b r14, vr9, vr6, vr14
vld.dw      [r7]+, vr11
vmsumad.h vr14, vr13, vr4
vld.dw      [r6]%, r12, vr9
loop      r11, .L1
.L2:
mov.w       0, vr4, r10
jmp         [lp]
```

(d) hand-optimized assembly code

Fig. 10 FIR filtering fuctions.

is acceptable overhead and enough efficient against hand-optimized assembler.

5. Related Work and Discussion

Here, we compare our approach with several related work for retargeting GNU

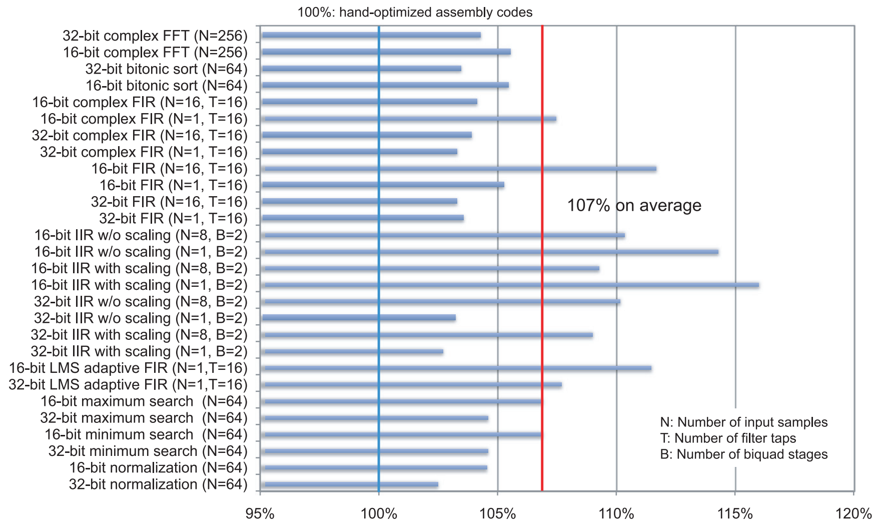


Fig. 11 Increase in the number of executed instructions of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes.

Binutils and discuss how they differs. There have been a lot of works for generating tools from processor architecture description language. **Table 4** shows a comparison among several conventional methods using the GNU toolchain and our approach.

Tensilica developed a configurable processor core, called Xtensa, in the late 1990s and provided a tool generator for the Xtensa[13]. Although the target processor architecture of the tool generator is limited to the Xtensa, it can generate ports of both GNU Binutils and GNU Debugger, simulators, and compilers for customized Xtensa cores. Unfortunately, according to the reference manual[13], there is not any description on the capability of adding extra relocation types used for extra instructions.

CGEN[6], which was released as open-source software from Red Hat in 2000, is a tool to generate code fragments for assemblers, disassemblers, and simulators. The generated codes can be embedded into GNU Binutils. Only for the MeP processor, CGEN has a feature to add intrinsic functions to the GCC.

Abbaspour presented in 2002 a systematic approach to retarget GNU Binutils[7]. An experimental result to generate GNU Binutils was reported in Ref. 7) for the SPARC architecture. In those years, the development of the ArchC started, which is an open-source binary utility generator[10]. Baldassin reported in Ref. 10) that the ArchC can retarget GNU Binutils and generate simulators for several processor architectures including the i8051 processor, for which there was no reference ports in the original GNU Binutils.

Table 4 Comparison among tool generation methods using the GNU toolchain.

(a) The Xtensa tool generator by Tensilica[13)]
(b) CGEN: Cpu generator by Red Hat[6)]
(c) rbinutils by Abbaspour[7)]
(d) ArchC by Baldassin[10)]
(e) The proposed method

Target base-processor architectures	
(a)	Xtensa configurable cores
(b), (c)	RISC CPUs
(d)	RISC/CISC CPUs
(e)	Existing CPUs which have ports of the GNU toolchain
Architecture description language (ADL)	
(a)	TIE (Tensilica Instruction Extension) language
(b)	Lisp-like language
(c)	Simple language
(d)	Simple C-like language
(e)	XML-based language
How to generate GNU Binutils	
(a)	Base binutils + dynamic link libraries (DLLs)
(b)	Only opcode library
(c), (d)	Templates + code fragments
(e)	Base binutils + Templates + plugins + code fragments
How to define relocation types	
(a), (b)	N/A
(c)	Explicit defitions written in instruction set description
(d), (e)	Implicit defitions extracted from instruction set description
How to generate simulators	
(a)	Base simulator + DLLs
(b)	Only libraries to execute/decode instructions
(c)	N/A
(d)	Templates + code fragments
(e)	Base simulator + templates + plugins + code fragments
How to generate compilers	
(a)	Base compiler + DLLs + header files for intrinsics
(b), (c), (d)	N/A
(e)	Base compiler + templates + code fragments + header files for intrinsics and emulation libraries

On the other hand, our proposed method can generate a set of the GNU toolchain including a compiler for a target processor. Regarding the kinds of tools that can be generated, i.e., compiler, assembler, simulator, etc., the proposed method has the same capability as the tool generator for the Xtensa. However, there is a difference between the tool generator for the Xtensa and the proposed method, in which the main scope of the proposed method is to add new instructions to existing base processors but not to modify the ISA of a base processor. While the Xtensa's tool generator can be used only for the Xtensa, our tool generator can be used for the existing processors that have ports of the GNU toolchain. There is also a difference in instruction syntaxes. The conventional methods (a)–(d) in Table 4 has a rule that instructions be written in a syntax of single mnemonic plus multiple operands. Our tool generator, however, generates the assembler that can handle not only a mnemonic style but also a function style and an algebraic style.

In terms of the amount of handwork necessary for generating toolchain, the proposed method and conventional methods require similar amount of handwork such as describing instruction specification. Although XML documents for the proposed method are tend to have more lines than conventional methods, it does not have that a big impact on the amount of handwork.

Regarding the time needed to generate toolchain, there is no big difference among the proposed method and conventional methods. The proposed method takes a couple of seconds to generate plugins, and takes several tens of minutes to build the toolchain involving the plugins. The time needed to build toolchain on the proposed method is the same as those of conventional methods such as ArchC and rbinutils.

On the proposed method, programmers need source modification using intrinsic functions in order to exploit new instructions and to make applications faster. This is a negative point compared with auto-customization frameworks of ISAs reported in Refs. 14)–16). While rewriting source codes for more speed is still a practical method, exploiting new instructions without source modification on the proposed method is one of our future works.

Our approach is not only dedicated to the V850, but it can also be applied to other processors. In fact, our method can generate toolchains for processors with a simple instruction extension based on the ARM and MIPS processors although these trials based on the ARM and MIPS processors are very simple and not mentioned in details in this paper.

6. Conclusion

We have proposed a new method to generate software development tools for instruction set extension of existing embedded processors. Our approach generates a toolchain (assembler, disassembler, linker, simulator, and compiler) for a target processor, which is based on an existing processor and which has additional instructions and registers, by adding software components as plugins to the base processor's toolchain to handle additional instructions and registers. We demonstrated that our approach worked effectively through an experiment based on the V850 microcontroller. As shown in this paper, by using intrinsic functions, the generated compiler could give as good performance as that of hand-optimized assembly codes.

Acknowledgments This work described in this paper has been made possible by appropriate assistance and suggestions of Syohei Yoshida, Aiko Watanabe, and Dr. Yuki Kobayashi, to whom we express our sincere appreciation.

References

1) Fauth, A., Van Praet, J. and Freericks, M.: Describing Instruction Set Processors Using nML, *Proc. European Design and Test Conference*, pp.503–507 (March 1995).
2) Clements, P.C.: A Survey of Architecture Description Languages, *Proc. 8th International Workshop on Software Specification and Design*, p.16 (March 22-23, 1996).
3) Zivojnovic, V., Pees, S. and Meyr, H.: LISA-machine description language and generic machine model for HW/SW co-design, *Proc. IEEE Workshop on VLSI Signal Processing IX*, pp.127–136 (1996).
4) Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N. and Nicolau, A.: EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability, *Proc. Design Automation and Test in Europe* (*DATE '99*), pp.485–490 (1999).
5) Kobayashi, S., Takeuchi, Y., Kitajima, A. and Imai, M.: Compiler Generation in PEAS-III: an ASIP Development System, *International Workshop on Software and Compilers for Embedded Processors* (*SCOPES*) (2001).
6) CGEN, the Cpu tools GENerator: http://sources.redhat.com/cgen/.

7) Abbaspour, M. and Zhu, J.: Retargetable Binary Utilities, *Proc. 39th Design Automation Conference*, pp.331–336 (2002).
8) Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G. and van Someren, H.: A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models, *Proc. Design, Automation and Test in Europe Conference and Exhibition* (*DATE'04*) (2004).
9) Mishra, P., Shrivastava, A. and Dutt, N.: Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs, *ACM Transactions on Design Automation of Electronic Systems* (*TODAES*), Vol.11, No.3, pp.626–658 (July 2006).
10) Baldassin, A., Centoducatte, P. and Rigo, S.: An Open-Source Binary Utility Generator, *ACM Transactions on Design Automation of Electronic Systems*, Vol.13, No.2, Article 27 (Apr. 2008).
11) NEC Electronics Corp.: User's Manual: V850 Family for Architecture, Document No.U10243EJ7V0UM00 (March 2001), available at http://www.necel.com/.
12) Gonzalez, R.E.: Xtensa: A Configurable and Extensible Processor, *IEEE Micro*, Vol.20, No.2, pp.60–70 (March-Apr. 2000).
13) Tensilica, Inc.: Tensilica Instruction Extension (TIE) Language Reference Manual (Nov. 2006).
14) Goodwin, D. and Petkov, D.: Automatic generation of application specific processors, *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp.127–147 (2003).
15) Sun, F., Ravi, S., Raghunathan, A. and Jha, N.K.: Custom-instruction synthesis for extensible-processor platforms, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.23, No.2, pp.216–228 (2004).
16) Nagamatsu, Y., Ishiura, N. and Hikichi, N.: Retargeting GCC and GNU Toolchain for Extended Instruction Set, *Technical report of IEICE*, VLD2005-103, pp.37–41 (Jan. 2006) (in Japanese).
17) Intel Corp.: Using Streaming SIMD Extensions 2 (SSE2) to Implement an Inverse Discrete Cosine Transform, Version 2.0, AP-945 (Sep. 2000).

(Received December 1, 2009)
(Revised February 26, 2010)
(Accepted April 14, 2010)
(Released August 16, 2010)

(Recommended by Associate Editor: *Tsuyoshi Isshiki*)

Takahiro Kumura received his B.E. and M.E. degrees in communication engineering from Osaka University, Osaka, Japan, in 1996 and 1998, respectively. He joined NEC Corporation in 1998 and has been engaged in research and development on digital signal processors and their applications. His research interests span many aspects of digital signal processing, fast algorithms, VLSI implementation of programmable DSPs, and optimized compilers for DSPs. He is a member of IEICE of Japan.

Soichiro Taga received his B.E. degree in infomatics from Kwansei Gakuin University, Hyogo, Japan, in 2009. He is now with Graduate School of Science and Technology, Kwansei Gakuin University. His current research interests include retargeting of compilers, code optimization, and testing of compilers.

Nagisa Ishiura received his B.E., M.E. and Ph.D. degrees in information science from Kyoto University, Kyoto, Japan, in 1984, 1986, and 1991, respectively. In 1987, he joined Department of Information Science, Kyoto University, where he was an Instructor until April 1991. He joined Department of Information Systems Engineering, Osaka University, Osaka, Japan, as a lecturer where he was promoted to an associate professor in December 1994. Since 2002, he has been a professor at School of Science and Technology, Kwansei Gakuin University, Hyogo, Japan. His current research interests include compilers for embeded processors, hardware/software codesign, and high-level synthesis. He is a member of IEEE, ACM, and IEICE.

Yoshinori Takeuchi received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1987, 1989 and 1992, respectively. From 1992 through 1996, he was a research associate of Department of Engineering, Tokyo University of Agriculture and Technology. From 1996, he has been with Osaka University. He was a visiting scholar in University of California, Irvine from 2006 to 2007. He is currently an Associate Professor of Graduate School of Information Science and Technology at Osaka University. His research interests include System Level Design, VLSI design and VLSI CAD. He is a member of IEICE of Japan, IPSJ, ACM, and SP, CAS and SSC Society of IEEE.

Masaharu Imai received his B.S. degree in electrical engineering from Nagoya University, Nagoya, Japan in 1974, then M.S. and Ph.D. degrees in information science from Nagoya University in 1976 and 1979, respectively. From April 1979 through March 1996, he has been with Department of Information and Computer Sciences, Toyohashi University of Technology, Toyohashi, Japan, where his final title was a professor. He has been a visiting professor in University of South Carolina, Columbia, SC, U.S.A. from 1984 to 1985. From April 1996 to present, he is with Osaka University, Osaka, Japan, where he is a professor of Department of Information Systems Engineering, Graduate School of Information Science and Technology. His research interest includes ASIP (Application domain Specific Instruction set Processor) design automation, hardware/software codesign, VLSI architecture, and system level design methodology of embedded systems. Since 1991, he has been working for EDA standardization including VHDL under IEEE and JEITA (Japan Electronics and Information Technology Industries Association). He is a member of IEEE, ACM, IEICE of Japan, and IPSJ.