IEICE TRANS. INF. & SYST., VOL.E97–D, NO.5 MAY 2014

PAPER

Retargeting Derivative-ASIP with Assembly Converter Tool

Agus BEJO[†a)], ***Nonmember***, **Dongju LI**[†], **Tsuyoshi ISSHIKI**[†], ***Members***, ***and*** **Hiroaki KUNIEDA**[†], ***Fellow***

SUMMARY This paper firstly presents a processor design with Derivative ASIP approach. The architecture of processor is designed by making use of a well-known embedded processor's instruction-set as a base architecture. To improve its performance, the architecture is enhanced with more hardware resources such as registers, interfaces and instruction extensions which might achieve target specifications. Secondly, a new approach for retargeting compiler by means of assembly converter tool is proposed. Our retargeting approach is practical because it is performed by the assembly converter tool with a simple configuration file and independent from a base compiler. With our proposed approach, both architecture flexibility and a good quality of assembly code can be obtained at once. Compared to other compilers, experiments show that our approach capable of generating code as high efficiency as its base compiler and the developed ASIP results in better performance than its base processor.

key words: *derivative ASIP, retargeting compiler, LISA*

1. Introduction

In system-on-chip (SoC) design, flexibility and performance are two important aspects which usually need to be optimized during the design phase. Recently, application specific instruction-set processor (ASIP) has been one of the most popular solutions proposed by many researchers to compromise those two aspects. ASIP solution balances architecture flexibility through software programmability and hardware performance through instruction extensions. Employing ASIP, however, leads to more complicated works since four standard design phases, architecture exploration, architecture implementation, software development tool and application design, and system integration and verification must be considered as a single design framework.

In the conventional ASIP design method all those design phases are done manually, consecutively and iteratively with less automation. Therefore, the conventional ASIP design method suffers from four things. Fist, it is a long, tedious and extremely error-prone process so that it becomes difficult to achieve time-to-market constraint. Second, it is less flexible against architecture exploration. Third, with less automation it is more difficult to match the profiling tools to an abstract specification of the target architecture. Fourth, manual process may cause inconsistency among architecture specification, software development tools and hardware implementation.

Manuscript received December 28, 2012.
Manuscript revised September 30, 2013.

[†]The authors are with the Department of Communications and Computer Engineering, Tokyo Institute of Technology, Tokyo, 152–8550 Japan.
a) E-mail: agus@vlsi.ss.titech.ac.jp

DOI: 10.1587/transinf.E97.D.1188

Currently, the trend of ASIP design method has been addressed to retargetable approach based on machine description language. It can generate more consistent software development tools and synthesizable HDL code automatically. Using machine description language ensures that a processor architecture can be expressed easily either in instruction accurate or cycle accurate. Architecture modification or even new architecture creation can be flexibly done with higher level abstraction language. Profiling, simulation and software verification become faster since all those processes do not use the detail information of the hardware implementation. A set of software development tool consist of compiler, assembler, linker, debugger and simulator can also be generated automatically. This method reduces the design effort and the design time significantly.

One remaining problem involves how to provide an efficient compiler for the ASIP architecture. Some ASIP design methods have been facilitated with compiler generation tool. Several works focusing on compiler design and retargeting compiler for ASIP have also been proposed. Those compilers, however, could not achieve as high efficiency as the conventional GPP's compilers. The conventional GPP's compilers usually result in better code quality than the existing ASIP's compilers.

In order to solve this problem, we propose a new approach to design ASIP with efficient compiler. Our proposed approach, called Derivative ASIP, is developed with LISA design framework. The compiler of the developed ASIP architecture is retargeted from a base compiler by using assembly converter tool. The contribution of our approach is to provide a practical method for retargeting ASIP architecture which can solve both architecture flexibility and compiler performance problems. With our approach, a good quality of assembly code can be inherited from its base compiler and the same performance at least can be achieved as its base processor.

2. Related Work

As it has been a new trend, several works on machine description language for ASIP design have been proposed. nML, ISDL, MIMOLA, EXPRESSION, FlexWare2, Xtensa and LISA are some of examples.

nML language is firstly proposed in [1], [2]. It allows designer to describe processor architecture in instruction-set level. However it lacks of flexibility on expressing pipeline mechanism since it cannot describe cycle-accurate model.

ISDL language is an advanced version of nML proposed in [3]. Although it supports synthesizable HDL code and auto generation of software development tools consisting of compiler, assembler, linker and simulator, it still has the same limitation as nML language that is cannot cover cycle-accurate model. Likewise, MIMOLA is a machine description language with Pascal-like syntax which is intended for microarchitecture design. It can generate compiler, simulator and hardware synthesizer but cannot support pipelined model. EXPRESSION language proposed in [4] provides better features which allows designer to describe processor architecture in cycle-accurate. However, it suffers from simulation speed and lacks of synthesizable HDL code generation. FlexWare2 proposed in [5] is also a framework which capable of generating a complete software development tools. However, the synthesizable HDL code generation is also not available and the compiler generation must be done in a separate framework by CoSy. Xtensa is another ASIP design method proposed by Tensilica [6]. In Xtensa, a customized RISC processor architecture can be created based on instruction-set templates. Although this method is flexible and retargetable but the architecture exploration space is limited by the availability of instruction template. Therefore, Xtensa is not suitable for ASIP which employs only a few instructions. Furthermore, Xtensa compiler cannot support instruction extensions directly unless the instruction extensions are given in HDL description.

A unified design method for ASIP design which provides a common basis for all design phases is proposed in [7], [8]. This method is Language for Instruction-set Architectures (LISA). LISA uses similar concept to the nML with more features. It can describe processor architecture in cycle-accurate model and capable of generating synthesizible HDL code. It can also generate a complete set of tools includes architecture exploration tool, software development tools, instruction-set simulator and debugger tool. Nevertheless, there still exists one problem in LISA design method. LISA does not have efficient compiler. A C compiler generator tool for LISA-modeled architecture has been proposed [9], [10]. Unfortunately, the assembly code generated by this compiler is less efficient than the one generated by the conventional GPP compiler. Moreover, a well understanding on compiler design is required to create compiler in LISA design framework. Because of these reasons, a new approach for retargeting the LISA-modeled processor architecture by means of assembly converter tool is proposed.

3. Derivative ASIP Approach

The concept of Derivative ASIP approach is to create a processor where its architecture is developed by referring to an existing embedded processor model. By using this concept, a compiler corresponding to the referred processor model can be retargeted to the developed processor architecture. In addition, the developed processor architecture can be either simplified or enhanced with more hardware resources such as instruction extensions, registers or interfaces to optimize its performance. Because of this reason our developed processor is called Derivative ASIP (DASIP) which means the architecture is derived from certain embedded processor architecture and enhanced with instruction extensions. The DASIP architecture is created by using LISA design framework. To enable our retargeting approach, original LISA design framework must be modified so that it provides a new entity for retargeting compiler through assembly converter tool. Figure 1 shows the modified LISA design framework to support Derivative ASIP approach. Original LISA design framework is shown on the left side and our proposed retargeting approach is added on the right side.

Derivative ASIP design flow begins with modeling processor architecture in LISA description language. LISA expresses processor architecture in C-like language. The ar-

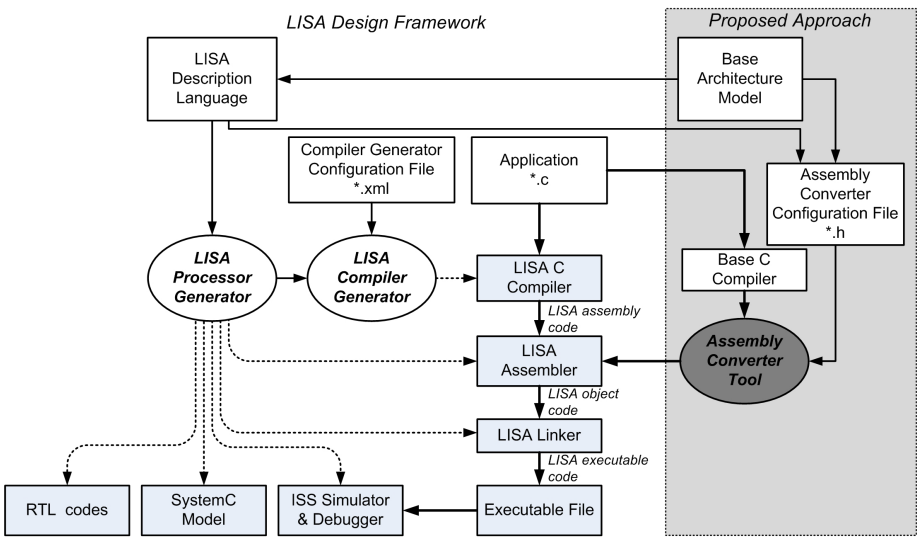

Fig. 1 Derivative ASIP approach design flow.

chitecture is basically defined by two components: (1) resources definition such as pipeline stages, interfaces, memory and registers, and (2) instruction-set definition includes syntax, coding and behavior. Since the DASIP architecture refers to an existing embedded processor model its instruction-set must be designed equivalently to the referred processor model. In this paper, the referred processor model will be called as a base processor and its compiler is called as a base compiler.

Once the processor architecture has been modeled, LISA processor generator generates RTL codes, SystemC model, instruction-set simulator (ISS), assembler and linker automatically. All these auto-generated files are indicated by light blue boxes. A LISA C compiler (LCC) can also be generated by using LISA compiler generator. However, a set of rules indicated by compiler generator configuration file is required to configure the C compiler generation. Creating this configuration file usually takes time and needs well understanding on compiler design. It is not suitable for non-expert ASIP designer. In original LISA toolchain flow, an application which is written in C is compiled by LISA C compiler. It generates LISA assembly code. The LISA assembly code then will be feeded to LISA assembler and LISA linker to generate LISA object code and LISA executable code respectively. The LISA executable code finally can be simulated or executed either in ISS simulator, SystemC model simulator, RTL simulator or hardware implementation such as FPGA.

In Derivative ASIP approach, we propose a different toolchain flow. Figure 2 shows the Derivative ASIP toolchain flow. Instead of using LISA C compiler, we compile the application using a base C compiler. By employing a well-optimized base C compiler, a good quality of assembly code can be obtained. Assembly converter tool is employed to convert the base assembly code into LISA assembly code. After the LISA assembly code is available, standard LISA toolchain flow can be used to generate LISA object code (*.lof) and LISA executable code (*.out) consecutively.

To verify the functionality of the DASIP architecture and the developed application software, LISA debugger is used. LISA debugger is a debugging tool that can monitor all hardware resources includes register, memory and I/O in every step cycle. It also provides instruction set simulator that can simulate the behavior of all instructions includes instruction extensions. With LISA debugger, the developed application can be emulated on the DASIP prior to FPGA hardware implementation.

4. Derivative ARM ASIP

This section presents an example of ASIP design with Derivative ASIP approach. Actually, the base processor architecture can be selected from any existing embedded processor model. In this paper, however, we used ARM as the basis of DASIP architecture because of its superiority among other embedded processors and the benefit of high quality code generated by ARM C compiler (ARMCC). Therefore, the developed DASIP will be called Derivative ARM ASIP or DAA for short. DAA is a 32-bit processor. It is designed with Harvard RISC architecture. Similar to ARM9 as its base architecture, DAA has 16×32-bit general purpose registers. However, it only has 4 pipeline stages consist of Fetch (FE), Decode (DC), Execute (EX), and Writeback (WB). This is different from ARM9 which has 5 pipeline stages. Figure 3 shows LISA codes to define DAA resources such as register, pipeline, memory map and IO interface.

To maintain the compatibility with ARMCC, DAA instruction-set is created equivalently to ARM instruction-set. This means that every instruction in DAA must have the equivalent one in ARM. Nevertheless, both instruction syntax and coding are not necessary to be the same. Moreover,

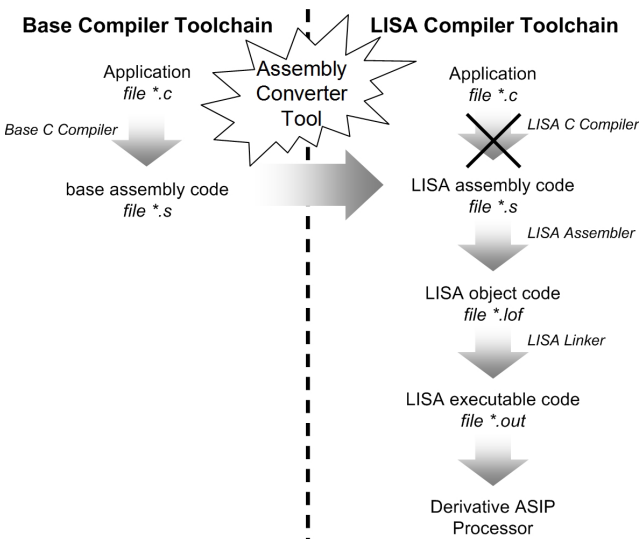

Fig. 2 Derivative ASIP toolchain flow.

```
1 : RESOURCE
2 : {
3 :     /* Register file : 16 x 32-bit registers */
4 :     REGISTER      TClocked<uint32>  R[0..15];
5 :     /* Program Counter Register */
6 :     REGISTER      TClocked<uint32>  PC ALIAS R[15];
7 :     ...
8 :     /* 4 pipeline stages */
9 :     PIPELINE                        pipe = {FE;DC;EX;WB}
10 :    /* pipeline register to pass data through the pipeline */
11 :    PIPELINE_REGISTER IN pipe
12 :    {
13 :        uint32   insn;          /* instruction word   */
14 :        uint8    rs1;           /* operand register 1 */
15 :        uint8    rs2;           /* operand register 2 */
16 :        ...
17 :    }
18 :    /* Memory Mapping for Program and Data */
19 :    MEMORY_MAP
20 :    {
21 :        RANGE(PMEM_START,PMEM_END)->prog_mem[(31..2)];
22 :        RANGE(DMEM_START,DMEM_END)->data_mem[(31..2)];
23 :    }
24 :    /* IO interface */
25 :    PIN OUT       TClocked<bit[32]>          PO;
26 :    PIN IN        TClocked<bit[32]>          PI;
27 : }
```

Fig. 3 LISA code to define DAA resources.

they can be totally different. To show the instruction equivalency between DAA and ARM, *add* instruction is given as an example. Figure 4 shows LISA codes to declare *add* instruction in DAA architecture. Every instruction declaration must contain three information, coding, syntax and behavior each of which is used to define the binary code, assembly code and functionality.

Figure 5 shows the instruction syntax and coding comparison betwen ARM and DAA. Although it is an equivalent instruction, the syntax and coding are different. The instruction equivalency here is applicable in instruction level, not in cycle-accurate level. Two instructions are equivalent if it has the same behavior that process the same input and produces the same output regardless how it works internally. DAA microarchitecture can be different from that of ARM. However, the input and output will be the same for every equivalent instruction. Moreover, instruction scheduling needs to be considered in the microarchitecture level. The DAA architecture must be designed so that its microarchitecture will be compatible with the instruction scheduling generated by ARMCC. In our example case, the number of pipeline stages on DAA is less than the one on ARM9. Therefore, bypassing and stalling techniques are employed to handle hazard conditions which may be caused by ARMCC instruction scheduling. With this design consideration, the number of pipeline stages will not affect the processor operation in general.

DAA owns basic ARM9 architecture with optimized resources for the target application. DAA, on the one side, simplifies ARM9 architecture by eliminating some redundant resources such as multi operation modes, bank register, debugging hardware, THUMB instruction-set (16-bit encoding), DSP enhancement instructions (64-bit long multiply and long multiply-accumulate instructions) and swap instruction. On the other side, DAA enhances its architecture with instruction extensions. In summary, Table 1 and 2 respectively show the feature similarities and the feature differences between ARM9 and DAA.

Instruction extensions are usually created based on application profiling information. Table 3 shows the profiling result of fingerprint authentication application [12] as the target application of DAA architecture. There are two functions that consume much more cycles than the others. *PreprocessFilter* and *Extraction* functions contribute 40% and 59% of the total execution cycles respectively. Based on this profiling result, two instruction extensions are created to improve the execution cycle of those functions. Table 4 shows

```
1 : /* Add instruction to add register data and immediate data */
2 : OPERATION addrri IN pipe.DC
3 : {
4 :   …
5 :   CODING { cond opcode I S 0b00 Rn Rd imm1 imm2 }
6 :   SYNTAX { opcode~cond~S~" " Rd" ," Rn" ,#" imm1",#" imm2 }
7 :
8 :   BEHAVIOR
9 :   {
10 :  …
11 :  }
12 : }
```

Fig. 4 LISA code to define *add* instruction on DAA.

ARM Instruction Format :

Condition (4-bit)	00 (2-bit)	I (1-bit)	OpCode (4-bit)	S (1-bit)	Rn (4-bit)	Rd (4-bit)	Operand (12-bit)

ARM Instruction Syntax :
<OpCode> {Condition} {S} Rd,Rn,#<Operand>
Example :
ADDEQ R2,R4,#0x12

DAA Instruction Format :

Condition (4-bit)	OpCode (4-bit)	I (1-bit)	S (1-bit)	00 (2-bit)	Rn (4-bit)	Rd (4-bit)	Op1 (4-bit)	Op2 (8-bit)

DAA Instruction Syntax :
<OpCode> {Condition} {S} Rd,Rn,#<Op1>,#<Op2>
Example :
DAAADDEQ R2,R4,#0,#0x12

Fig. 5 *add* instruction equivalency between ARM and DAA.

Table 1 ARM9 and DAA feature similarities.

Features	ARM9	DAA
Number of General Purpose Registers	16	16
ARM instruction-set (32-bit encoding)	YES	YES
Branch Instructions (B, BL)	YES	YES
Data Processing Instructions (AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN)	YES	YES
Multiply and Multiply-Accumulate (MUL, MLA)	YES	YES
Single Data Transfer Instructions (LDR, STR)	YES	YES
Block Data Transfer Instructions (LDM, STM)	YES	YES

Table 2 ARM9 and DAA feature differences.

Features	ARM9	DAA
Number of Operating Modes	6	1
Bank Registers	YES	NO
THUMB instruction-set (16-bit encoding)	YES	NO
Multiply Long and Multiply-Accumulate Long (MULL, MLAL)	YES	NO
Single Data Swap Instruction (SWP)	YES	NO
Instruction Extensions	NO	YES

Table 3 Fingerprint authentication profiling.

Function Name	Calls %	Cycles
SetFeatureData	0.06	320980
ZeroWordMem	0.04	209704
SetWorkspace	0.00	2108
PreprocessFilter	40.46	228676813
Extraction	59.44	335917509
CheckFeatureQuality	0.00	2012

Table 4 Two instruction extensions created on DAA architecture.

MC {B}{F} Ra,Rb,Rc **Filter {F} Ra,Rb,Rc,Rd**
{B}: Data type either byte (8-bit), half (16-bit) or word (32-bit)
{F}: Filter type either HPF or LPF
Ra: Source address
Rb: Destination address
Rc: Data size
Rd: Initial data

the syntax of instruction extensions.

MC is a new instruction created to perform memory copy operation with additional function to accumulate the copied data. *Filter* is another new instruction designed to perform filter operation. Operands *Ra*, *Rb*, *Rc* and *Rd* indicate source address, destination address, data size and initial data respectively. Parameter {*B*} determines the data type of the copied data. It can be either byte, short or word. Parameter {*F*} in *MC* instruction indicates the accumulation status. Asserted {*F*} means the copied data will be accumulated and stored in an internal register which later on can be used to support filter operation. Parameters {*F*} in *Filter* instruction determines the type of filter operation. It can be either *HPF* for high pass filter or *LPF* for low pass filter.

5. Assembly Converter Tool

Assembly converter tool plays an important role in the Derivative ASIP approach. It acts as a bridging tool that allows toolchain flowing from a base compiler toolchain to LISA compiler toolchain. Assembly converter tool performs assembly code conversion from one assembly code to another assembly code which is suitable for LISA environment. Besides, it also handles instruction extension generation. With this concept, the base compiler can be retargeted to the LISA-modeled processor architecture.

Our assembly conversion concept works well on some assumptions. Firstly, the instruction scheduling is handled not in assembly conversion stage, but in hardware design stage. Secondly, the memory mapping of DASIP architecture must be set in the same configuration as its base processor architecture. Thirdly, the register introduction and the register allocation must maintain one-to-one mapping between base architecture and DASIP architecture. Fourthly, the compiler parameter must be set up for not thumb, but arm architecture as a target CPU. Fifthly, for simplicity, the instruction extension must be designed with the maximum 4 arguments. Sixthly, compiler optimization must not affect the assembly code generation of calling convention. Therefore, when the application source code is compiled with optimization level -O1, -O2 or -O3, the body of instruction extension function must be kept the same as its original one. In contrast, if the application source code is compiled without optimization (-O0), the body of instruction extension function can be filled with an empty statement.

5.1 Assembly Conversion Mechanism

The idea behind assembly conversion mechanism is inspired by GCC machine description (MD) file. In GCC compiler internal mechanism, an input source code will be translated into intermediate representation in the form of register transfer level (RTL). This RTL contains instruction patterns that represent statements declared in the source code. Instruction pattern is independent from target processor architecture. Therefore, it can be freely ported to any processor architecture. In this case, MD file is required to help the assembly code generation. Every target processor architecture must have one MD file to describe their proper assembly code. This means that the key of retargeting compiler depends on the availability of intermediate representation which is usually obtained from C code in the front end of base compiler. Once intermediate representation has been obtained, it can be used to generate any assembly code suitable for the target processor architecture.

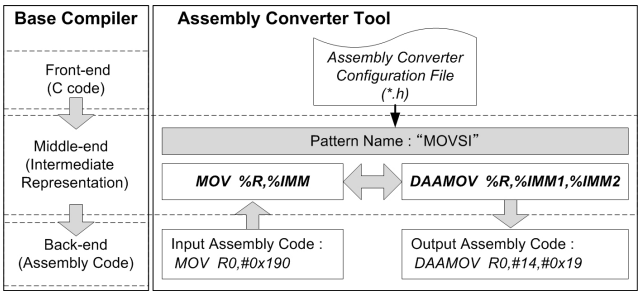

Fig. 6 Assembly conversion mechanism.

In our retargeting approach, instead of using one direction flow that is from intermediate representation to assembly code, we introduce the inverse one. This inverse direction flow allows an assembly code of specific processor architecture to be reversed back to its intermediate representation in the form of instruction pattern. With this concept, we can lift up any assembly code into instruction pattern and then retarget it to another assembly code. Figure 6 shows our proposed assembly conversion mechanism. Suppose that we already have assembly code generated by a base compiler. This assembly code will be the input of assembly converter tool. The conversion begins with scanning the input assembly code and parsing it into opcode and operand. Based on these opcode and operand information, it will be translated into instruction pattern form. If the input instruction pattern has been known, the equivalent one can also be obtained from the information given in configuration file. Using this instruction equivalency information, another assembly code for the target processor architecture can be generated.

Similar to GCC MD file, a file is required to configure the assembly conversion mechanism. This file is called assembly converter configuration file. It contains information such as register definition, register allocation, instruction pattern equivalency and instruction extension. Figure 7 shows an example of assembly converter configuration file suitable for retargeting ARMCC to DAA architecture. DAA registers are designed with the same structure as ARM. Lines 3-14 show the register definition and the register allocation. *R0-R3* and *R4-R8* are allocated as register argument and register variable. *R0*, *R12*, *R13*, *R14* and *R15* are defined as register return value, register index pointer, register stack pointer, link register and register program counter. Lines 16-23 describe instruction pattern equivalency. There will be a pair of instruction pattern for each input assembly code, one for the base instruction pattern and one for the target instruc-

```
 1 : #define DER(__INSTRUCTION__) "DER"__INSTRUCTION__
 2 :
 3 : REGDEF   REGLIST[] = {
 4 : …
 5 : {REG_RTN,   0,   0},
 6 : {REG_ARG, 0-3, 0-3},
 7 : {REG_VAR, 4-8, 4-8},
 8 : …
 9 : {REG_IP,   12,  12},
10 : {REG_SP,   13,  13},
11 : {REG_LR,   14,  14},
12 : {REG_PC,   15,  15},
13 : …
14 : };
15 :
16 : INSPATTERN   PATTERNLIST[] = {
17 : …
18 : {code_addsi, "ADD %R,%R", DER("ADD %R,%R")},
19 : {code_movsi, "MOV %R,%IMM", DER("MOV %R,#%IMM1,#%IMM2")},
20 : {code_addsi, "ADD %R,%R, ASR %IMM", DER("ADD %R, %R, ASR #%IMM")},
21 : {code_cmpsi, "CMP %R,%IMM", DER("CMP %R,#%IMM1,#%IMM2")},
22 : …
23 : };
24 :
25 : XINST XINSTLIST[] = {
26 : …
27 : tHPFFilter,
28 : …
29 : };
```

Fig. 7 Assembly converter configuration file.

```
1 : XINST tHPFFilter =
2 : {
3 :   "HPF_Execute",
4 :   "XINST_HPF_Execute",
5 :   "Filter HPF r0,r1,r2,r3",
6 :   4
7 : };
```

Fig. 8 *Filter* instruction definition in configuration file.

tion pattern. For example, *MOV %R,%IMM* is an instruction pattern for ARM. The equivalent one in DAA is *DAAMOV %R,#%IMM1,#%IMM2*. Lines 25-29 is optional information used to define instruction extensions. This information is stored as *XINST* data type. It contains 4 elements such as instruction name, function name, assembly code and the number of operand.

5.2 Instruction Extension Generation

The base compiler usually works for GPP architecture only. It does not have ability to generate instruction extensions. Therefore, the assembly converter tool is also employed to handle the instruction extension generation. To explain how the instruction extension can be called from C code, *HPF_Execute* function is given as an example. This function performs high pass filter operation for a line block of data input. It will be replaced by a single instruction *Filter HPF r0,r1,r2,r3*. The information of this instruction must be provided in the configuration file as indicated by *tHPFFilter* data in Fig. 7 line 27. The content of *tHPFFilter* data is shown in Fig. 8. During the assembly conversion, if a function with *XINST_HPF_Execute* name and has 4 arguments is detected, instruction *Filter HPF r0,r1,r2,r3* will be released by assembly converter tool to replace the original assembly code of *HPF_Execute* function.

Figure 9 shows the original C code of *HPF_Execute*

```
1 : void HPF_Execute (short *pSrc, short *pDst, int size, int iData)
2 : {
3 :   int nTotal = iData;
4 :   for(i=0; i<size; i++) {
5 :     short *pSumL, *pSumR;
6 :
7 :     // Update sum of next one block
8 :     pSumL = (pSrc + i);
9 :     pSumR = (pSrc + i + HPF_SIZE);
10 :    nTotal += ((*(pSumR++)) - (*(pSumL++)));
11 :
12 :    // Calculate HPF value for a current pixel
13 :    int nAvg = -(((unsigned short)nTotal) >> (HPF_SHIFT - 1));
14 :    *(pDst++) = (short)(128 + (short)(nAvg >> 1) + *(pSrc++));
15 :  }
16 : }
```

Fig. 9 The original source code of *HPF_Execute* function.

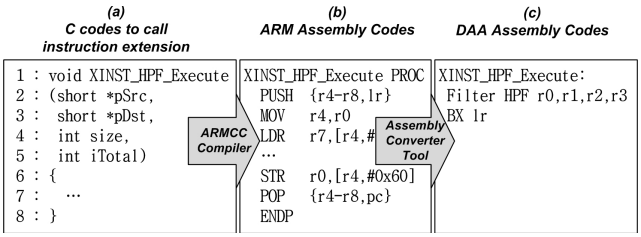

Fig. 10 *Filter* instruction generation.

function. It has 4 arguments *pSrc*, *pDst*, *size* and *iData* each of which indicates input data, output data, data size and initial input data respectively. Referring to the register allocation information which is defined in the configuration file, the data of argument *pSrc*, *pDst*, *size* and *iData* will be hold by register *r0*, *r1*, *r2* and *r3* consecutively. To indicate that a function is declared as an instruction extension, the function name must begin with prefix *XINST_*. Figure 10 (a) line 1 shows that *XINST_HPF_Execute* function is declared in the C source code. Compiled by ARMCC, it generates several lines of ARM assembly codes as shown in Fig. 10 (b). When the assembly converter tool is employed, it scans these ARM assembly codes and checks whether the declared function is an instruction extension or not. If so, the ARM assembly codes will be replaced by a single instruction *Filter HPF r0,r1,r2,r3* as shown in Fig. 10 (c).

Two types of instruction extensions are issues of interest. One is a coarse-grained multi-cycle and the other is a fine-grained single-cycle instruction. The former extensions or the extension to add hardware accelerators are main targets for our proposed approach. Instruction *Filter* is an example of multiple-cycle-execution instruction. However, the latter fine-grained instruction extensions may consume the more execution time and the more hardware resources. They require more cycles than the reduction of instruction cycles to branch to the extended instruction for the function call. In addition, they require an additional set of registers to hold the data of argument and to store the data of a return value for a new extended instruction.

6. Experiment Results

Some experiments were carried out to evaluate the perfor-

mance of DAA architecture developed by our approach.

6.1 Method Comparison

Table 5 shows the comparison among three ASIP design methods, LISA [10], GCC extension [11] and Derivative ASIP. As shown in the table, Derivative ASIP approach has three main benefits. First, it inherits the advantage of hardware implementation from LISA design method. Second, it has more practical and easier way to provide compiler toolchain. Third, it maintains the quality of assembly code as well-optimized as its base compiler.

6.2 Compiler Performance

In this experiment, we compared the performance of several compilers. NIOS II EDS v8.1, LCC V2009.1.1, GCC 4.6.0 and ARMCC RVDS 4.0 were used to compile fingerprint authentication [12] as the target application. DAA compiler was obtained by retargeting ARMCC to DAA architecture with assembly converter tool. Usually, the compiler performance is indicated by the number of cycles and the code size in bytes. The smaller execution cycle and code size the better compiler performance.

Figure 11 shows the execution cycle of *BLS_InitWorkspace*, *BLS_CreateTemplate*, *BLS_Extraction* and *BLS_Matching* functions when the compiler was set without optimization (-O0). Figure 12 shows the total of execution cycle and the size of program code when the compiler optimization was varied from -O0 to -O3. These results clearly show that ARMCC generates more efficient code and consumes less cycle than NIOS [13], LCC and GCC compilers. In case the compiler optimization is set to -O0, the ARMCC efficiency is double than the others. Since DAA compiler inherits the benefit of ARMCC performance, by default DAA will have the same efficiency as ARMCC. Employing *MC* and *Filter* instruction extensions, DAA improves its performance by 22.5% better cycle and 140 bytes less code size than ARMCC.

Table 5 Approach Comparison: (a) LISA, (b) GCC Extension, (c) Derivative ASIP.

Feature	(a)	(b)	(c)
Hardware implementation	Yes	No	Yes
Easy compiler creation	No	Yes	Yes
Well-optimized code	No	No	Yes

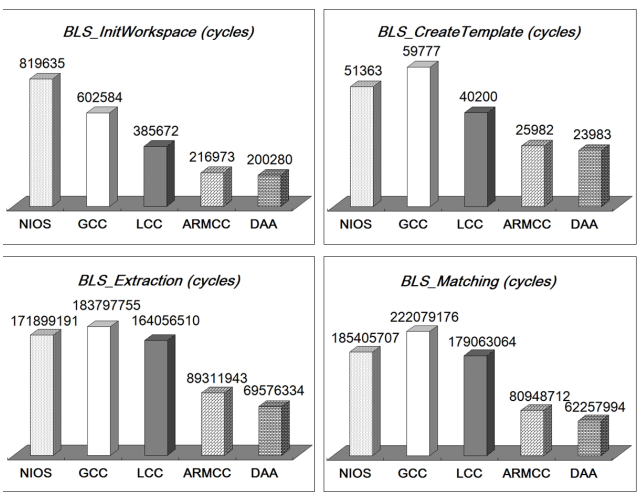

Fig. 11 Execution cycle comparison with compiler optimization -O0.

6.3 Instruction Extension Improvement

DAA processor was designed with two instruction extensions to improves the execution time of *PreprocessFilter* and *Extraction* functions as explained in Sect. 4. Table 6 shows the number of execution cycles before and after employing instruction extensions. It can be seen that with two instruction extensions it reduces the execution cycle of *PreprocessFilter* and *Extraction* functions up to 31.07% and 22.09% respectively.

6.4 Hardware Optimization

This experiment was done to show how much hardware optimization can be achieved by our approach in term of area and power consumption. To estimate the area and power consumption of DAA processor, the RTL file generated by LISA processor generator tool was synthesized by using

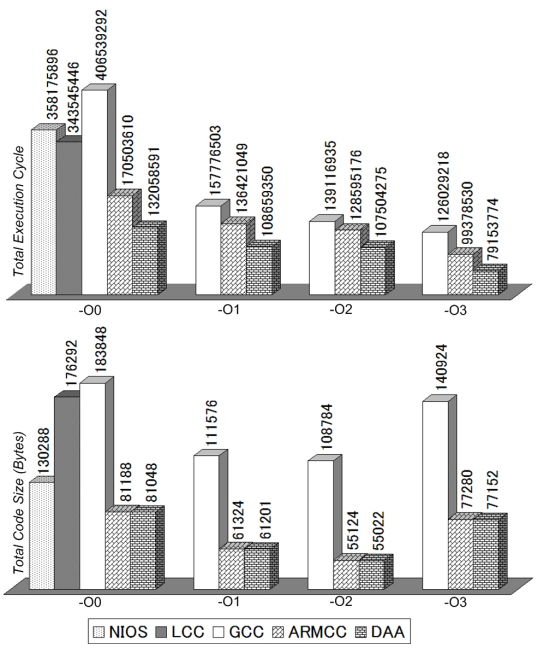

Fig. 12 Compiler performance with various optimization levels.

Table 6 Instruction extension improvement.

Function	Before*	After*	Improvement
PreprocessFilter	28217042	21529037	31.07%
Extraction	89311943	69576334	22.09%

**unit in cycle*

Table 7 Area and power comparison.

Item	ARM9	DAA
Area	0.613 mm^2	0.555 mm^2
Power	9.5 mW	9.2 mW

Synopsys Design Compiler. Table 7 shows that synthesized with TSMC 90 nm technology at 200 MHz, DAA results in a little bit smaller area and less power consumption than ARM9. This result means that with our approach we can improve the processor performance for certain target application without losing its area and power consumption through architecture optimization.

7. Conclusion

A new approach of ASIP design with Derivative ASIP has been introduced. A practical way for retargeting compiler for the Derivative ASIP architecture using assembly converter tool has also been proposed. Derivative ARM ASIP proved that with our proposed approach a high quality of assembly code and a better performance of processor architecture could be achieve without losing area and power consumption.

References

[1] M. Freericks, The nML Machine Description Formalism, Fachbereich Informatik, Tech. Univ. Berlin, Germany, 1991.
[2] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction-set processors using nML," Proc. Eur. Design and Test Conf. Paris, France, 1995.
[3] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," Proc. Design Automation Conf., 1997.
[4] A. Halambi, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," Proc. Conf. Design Automation and Test Eur, 1999.
[5] P. Paulin, "Design automation challenges for application-specific architecture platforms," Proc. SCOPES, Workshop Software and Compilers for Embedded Systems, 2001.
[6] R. Gonzales, "Xtensa: A configurable and extensible processor," IEEE Micro, vol.20, pp.60–70, 2000.
[7] O. Schliebusch, H. Meyr, and R. Leupers, Optimized ASIP Synthesis from Architecture Description Language Models, Springer, 2007.
[8] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors using a machine description language," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol.20, no.11, pp.1338–1354, 2001.
[9] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. Van Someren, "A methodology and tool suite for C compiler generation from ADL processor models," Proc. Conference on Design, Automation and Test in Europe (DATE), 2004.
[10] M. Hohenauer and R. Leupers, C Compilers for ASIPs, Automatic Compiler Generation with LISA, Springer, 2010.
[11] T. Kumura, S. Taga, N. Ishiura, Y. Takeuchi, and M. Imai, "Software development tool generation method suitable for instruction set extension of embedded processors," IPSJ Trans. System LSI Design Methodology, 2010.
[12] A.A. Mohamed Mostafa, D. Li, and H. Kunieda, "Minutia ridge shape algorithm for fast on line fingerprint identification system," ISPACS Proceeding, 2000.
[13] A. Bejo, N. Hao, S.S. Woo, Z. Ru, D. Li, T. Isshiki, and H. Kunieda, "A reconfigurable hardware design for fingerprint authentication and fingerprint navigation," The International Conference on System on Chip Design Challenges (ICoSoC), 2010.

Agus Bejo received his B.Eng and M.Eng degrees from Electrical Engineering Department, Gadjah Mada University, Indonesia in 2003 and Chulalongkorn University, Thailand in 2007, respectively. Currently, he is a Ph.D student at Kunieda-Isshiki Laboratory, Department of Communications and Computer Engineering, Tokyo Institute of Technology, Japan. His current research focuses on processor architecture and SoC design especially for fingerprint authentication and navigation application.

Dongju Li received the Ph.D degree in Electrical and Electronics from Tokyo Institute of Technology in 1998. She is currently an Associate Professor at Dept. of Communications and Computer Engineering, Graduate School of Science and Engineering, Tokyo Institute of Technology. Her current research interests include embedded algorithm for fingerprint authentication, fingerprint authentication solution for smart phone, VLSI architecture design and methodology and SOC design for multimedia applications such as fingerprint and video CODEC. Dr. Li is a member of IEEE CAS and IEICE since 1998.

Tsuyoshi Isshiki received B.E. and M.E. degrees from Tokyo Institute of Technology in 1990 and 1992, respectively, and received Ph.D in Computer Engineering from University of California at Santa Cruz in 1996. He is currently an Associate Professor at Tokyo Institute of Technology, Dept. of Communications and Computer Engineering. His research interests include multimedia SoC designs, Multiprocessor SoC design methodology and its design tools.

Hiroaki Kunieda received his B.E. M.E. and Doctor of Engineering degree from Tokyo Institute of Technology, Tokyo, Japan in 1973, 1975 and 1978, respectively. Since 1978, he has been with Faculty of Engineering, Tokyo Institute of Technology, where he is now a Professor in the Department of Communications and Computer Engineering, Tokyo Institute of Technology. His research interests include SoC Design, Multi-media LSI design, SoC CAD, and Fingerprint Authentication System. He is a fellow in IEICE and a senior member in IEEE CAS Society.