Journal of Information Processing Vol.22 No.2 131–141 (Apr. 2014)

[DOI: 10.2197/ipsjjip.22.131]

Regular Paper

A Method of Software Development Tool and Hardware Generation for ASIP with a Co-processor based on the Derivative ASIP Approach

Agus Bejo[1,a)] Dongju Li[1] Tsuyoshi Isshiki[1] Hiroaki Kunieda[1]

Received: May 6, 2013, Accepted: November 1, 2013

Abstract: In this paper, a processor design method using the Derivative ASIP approach is introduced. The concept of Derivative ASIP is basically to develop an ASIP architecture based on existing GPP processor architecture in order to diminish the design effort and shorten the design time. In this approach, the base processor architecture can be enhanced with more co-processor/instruction extensions quickly since all the required development tools have been available for the base processor. In order to support the Derivative ASIP approach, a new tool called the Co-processor/Instruction Extension Generator Tool is developed. This tool generates complementary files suitable for updating the base processor architecture with co-processor/instruction extensions. A complete set of software development tools consisting of a compiler, assembler, disassembler, linker, debugger, simulator and also hardware implementation for the modified ASIP architecture can be generated automatically by using these complementary files. With our proposed tool, a new co-processor/instruction extension can be designed and added to the base architecture more easily. It contributes to the reduction of the architecture exploration time in the design stage. Derivative ARM ASIP architecture enhanced with instruction extensions for the AES algorithm and a co-processor for the fingerprint navigation algorithm is given to demonstrate the effectiveness of our approach.

Keywords: ASIP, LISA, GCC Compiler, ImpulseC, instruction extension.

1. Introduction

Nowadays the need for Application Specific Instruction-set Processors (ASIPs) for embedded system applications has been increasing significantly because of the boom in portable electronic products in the market. It demands higher and higher performance everyday. ASIP is a common method which is effectively used to improve the hardware performance without losing its flexibility. Therefore, ASIP designers are insist on creating a methodology which offers faster design time, less design effort, re-usability and flexibility against architecture modification.

In the conventional design methodology, ASIP is developed by a hardware description language such as VHDL or Verilog. This approach requires big effort, a long design cycle and experienced engineers because it is hard work to describe an architecture model in the hardware description language. There is a bottle-neck in the application software development stage since the application software can not be developed unless the software development tool set has been made available. Meanwhile the software development tool set itself can not be created before the processor architecture design has been accomplished. Moreover, architecture exploration and design verification are usually done manually and iteratively. It consumes a lot of time and tends to be error-prone. Because of these reasons, recently the ASIP design method has been changed from a hardware description language (HDL) to an architecture description language (ADL). ADL offers better flexibility and easiness in term of architecture exploration since it describes the hardware behavior with a higher level of abstraction in term of the description language.

There have been several ADL-based ASIP design approaches proposed such as MIMOLA [1], ISDL [2], nML [3] and LISA [4]. MIMOLA is a structural-based ADL which describes processor architecture in a similar structure as hardware description language. Therefore, MIMOLA is suitable for automatic hardware generation. The drawback of structural-based ADL is that it does not contain instruction-set information which can be extracted to assist automatic compiler generation. In contrast, ISDL is a behavioral-based ADL that allows architecture designer to describe processor architecture with a set of instructions including the behavior of each instruction. This makes behavioral-based ADL more suitable for compiler generation but not for hardware synthesis generation.

It is challenging work to compromise the benefit of those two structures into a single design framework. nML and LISA are two examples among several approaches that try to mix structural-based and behavioral-based ADL. Even though nML is capable of generating hardware implementation and software development tool set consisting of a compiler, assembler, linker and simulator, it suffers from a cycle accurate model (in particular, a pipelining mechanism). nML can not describe processor ar-

[1] Kunieda-Isshiki Laboratory, Department of Communications and Computer Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan

[a)] agusbj.aa@m.titech.ac.jp

chitecture in a cycle-based model. LISA is another mixed ADL approach whose features are almost perfect. It can generate hardware implementation and a complete set of software development tool. It can also describe cycle-based models. However, LISA compiler generation is not suitable for ASIP and it can not support co-processor extensions.

Another problem among those approaches is none of them has a concept to reuse existing processor models and existing software development tool sets. All mentioned approaches must begin the development step from scratch. Whenever new processor architecture will be created, a new set of software development tools must also be developed. This condition causes current ASIP design approaches to consume a longer design time.

There is one approach proposed by Kumura et al. [5] that reuses existing software development tool sets. Their concept is constructed based on the GCC toolchain [12]. It upgrades the existing GCC toolchain consisting of compiler, assembler, linker and debugger by adding plugins to enhance existing processor architecture with instruction extensions. However, it can not be implemented in hardware because there is no solution to generate hardware implementation for the modified processor architecture.

Therefore, in this paper we propose an alternative approach which not only can solve the problem on both LISA and Kumura's approaches but also expand the architecture exploration space to cover co-processor extension. On the one side, our proposed approach adopts Kumura's concept to reuse existing software development tools so that the architecture designer does not have to design the software development tool from scratch. This means the compiler development stage can be eliminated and the overall design time becomes shorter. On the other hand, the LISA design framework is used to create a processor architecture because LISA has the ability to generate synthesizable RTL code from a higher level abstraction language. Another important feature in our approach is that it accommodates third-party tools i.e. the ImpulseC [11] design framework is integrated with the LISA-modeled ASIP architecture. ImpulseC is used to create a co-processor which can not be realized for the LISA design framework. By this approach, a comprehensive solution of the ASIP design approach which covers the software development tool generation and ASIP hardware implementation with wider architecture coverage may satisfy today's ASIP design demands.

In order to support our proposed approach, a tool called Co-processor/Instruction Extension Generator is developed. The purpose of this tool is basically to help the architecture designer to create a new co-processor/instruction extension by shortening turnaround time of each design exploration. In general, the contribution of our proposed approach includes:

- Less overall design effort and shorter design time.
- More practical method to design an optimized ASIP architecture with additional co-processor or instruction extension.
- More flexible and wider architecture exploration space.

2. Derivative ASIP Approach

The Derivative ASIP approach reuses existing software development tool sets in order to save the compiler development time. It creates a processor architecture referring to an existing embedded processor architecture as the base model so that the software development tool set which is already available for the base processor can be easily ported to the newly developed processor architecture. In addition, the processor architecture can be simplified by removing unused features or enhanced with more hardware resources such as registers, interfaces or extended instructions to optimize its performance. Because of this reason our developed processor is called Derivative ASIP which means derived from certain processor architectures and enhanced with extended co-processor/instructions.

The base processor architecture can be selected from any existing embedded processor model as long as it has been supported by the GCC compiler. In other words, the base processor selection is actually constrained by the availability of the GCC compiler for the base processor being selected. In this paper, ARM is chosen as the base processor architecture because of its superiority among other embedded processors and the benefit of the availability of the open source GNU Compiler *arm-gcc*.

Figure 1 shows the design flow of the Derivative ASIP approach. Basic LISA design framework is shown on the left side, ImpulseC hardware design framework on the right side and our proposed Co-processor/Instruction Extension Generator tool, indicated by the dotted red box, is added in the middle.

The Derivative ASIP design flow begins with the modeling processor architecture in the LISA description language. Since our approach uses an existing embedded processor architecture as the reference model, the Derivative ASIP instruction-set must be designed equivalently to the base processor instruction-set. At the same time the GCC compiler must be re-ported to the Derivative ASIP architecture. In this case, the GCC compiler porting will not be a large portion of the work because the base processor already includes the GCC compiler. A small modification is sufficient to complete the GCC porting. The new GCC compiler for the Derivative ASIP architecture will be named *<dasip>-gcc*.

To enhance the base processor architecture, a new co-processor

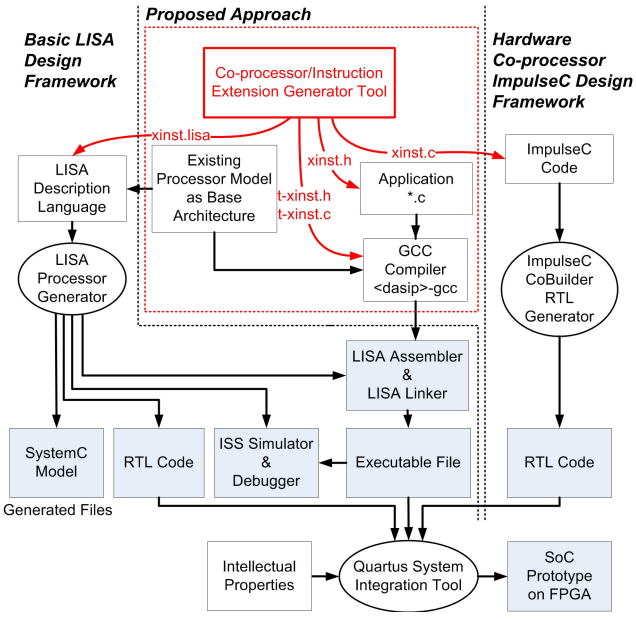

Fig. 1 Derivative ASIP design flow.

or instruction extension can be added. The design of co-processor or instruction extension is usually decided by an architecture designer based on application profiling information. The Co-processor/Instruction Extension Generator tool is employed to help the architecture designer to create the co-processor or instruction extension. It generates complementary files which are used to update the LISA-modeled processor architecture, GCC compiler, ImpulseC-modeled co-processor and application source code. Once the LISA-modeled processor architecture is available, LISA processor generator produces the RTL codes, SystemC model, instruction-set simulator (ISS), LISA assembler and LISA linker automatically.

A modified application's source code that employs instruction extensions, is compiled by *<dasip>-gcc* compiler to generate assembly codes. In the GCC compiler updating process, actually only *cc1* (GNU C compiler) part is being updated. The *gas* (GNU assembler) and *ld* (GNU linker) are not necessary to be updated. These parts will be replaced by *lasm* (LISA assembler) and *llnk* (LISA linker). Therefore, the GCC source code must be pre-modified so that the assembly code generated by the *<dasip>-gcc* compiler must be compatible with LISA assembler. This is a benefit of our approach because in order to update the GCC compiler, we do not need to modify the GNU assembler, GNU linker and GNU debugger as what is done in Kumura's approach [5]. **Figure 2** clearly shows the difference between Kumura's toolchain flow and Derivative ASIP toolchain flow.

To generate an executable file for the Derivative ASIP processor, there are two ways. First is using Kumura's toolchain flow as indicated by the black arrows. Second is using Derivative ASIP toolchain flow as indicated by the red arrows. Dotted-lines indicate that complementary files generated by Co-processor/Instruction Extension Generator tool are used to update the GCC compiler and the LISA-modeled processor architecture. In the Derivative ASIP approach, the assembly code generated by *<dasip>-gcc* compiler will be forwarded to the LISA assembler and the LISA linker to generate LISA object code and LISA executable code respectively. The LISA executable code then can be simulated or executed either on the ISS simulator, SystemC model simulator, RTL simulator or in hardware implementation

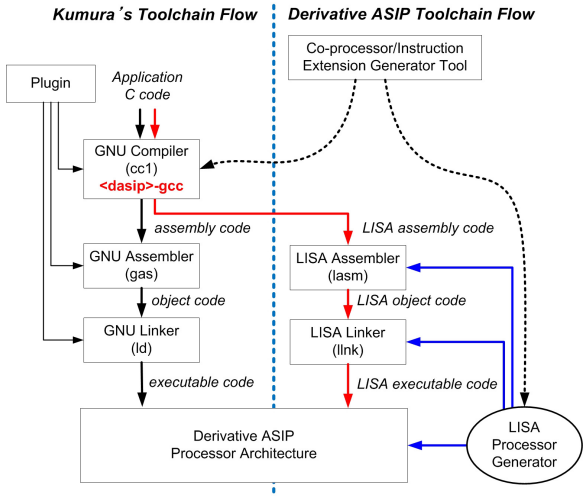

Fig. 2 Derivative ASIP toolchain flow.

such as FPGA.

To expand the architecture exploration space, co-processor design is accommodated by the ImpulseC [11] design framework. A complementary file suitable for ImpulseC environment is used to update a template of ImpulseC code. Corresponding RTL codes for the co-processor will be generated by the ImpulseC RTL Generator tool. This co-processor can be incorporated with the Derivative ASIP architecture at the RTL level. For hardware verification, the Derivative ASIP architecture, co-processor, executable file and other intellectual properties are integrated by using the Quartus System Integration Tool [14]. This integrated system, called system-on-chip, is synthesized to generate a bitstream file for FPGA implementation.

3. Derivative ARM ASIP

This section presents the detail of the Derivative ASIP design. Since ARM is used as the base architecture of Derivative ASIP, the developed Derivative ASIP processor will be called Derivative ARM ASIP or DAA for short. DAA is a 32-bit processor. It is designed according to the Harvard RISC architecture. Referring to its base architecture model, ARM9, DAA has 16 x 32-bit general purpose registers. However, there are only 4 pipeline stages: Fetch (FE), Decode (DC), Execute (EX), and Writeback (WB) which is different from ARM9 with 5 pipeline stages. **Figure 3** shows the LISA code to define the DAA resources consisting of register, pipeline, memory map and interface.

To maintain the easiness of GCC porting for the DAA processor, the DAA instruction-set is created equivalently to ARM instruction-set. This means that every single instruction on the DAA processor will have an equivalent one in the ARM processor. However, the DAA instruction format does not have to be the same as the ARM one. Moreover, the DAA assembly syntax can be totally different from ARM.

DAA includes basic ARM9 architecture with optimized resources for the target application. DAA, on the one side, simplifies ARM9 architecture by eliminating unused resources such as multi operation modes, bank registers, debugging hardware,

```
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.

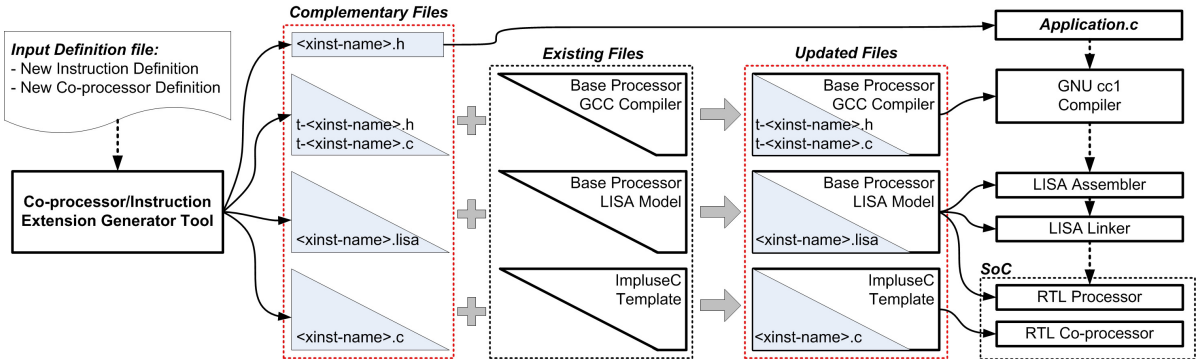

Fig. 4 Co-processor/instruction extension design flow using our proposed tool.

(a)

```
1 : <?xml version="1.0" encoding="UTF-8"?>
2 :
3 : <Xinst>
4 :  <xname>MyNewAdd</xname>
5 :  <nresource type="register" width="32" size="1">RBUF</nresource>
6 :  <nopcode coding="0b0010">NEWADD</nopcode>
7 :  <xcoding>0b0000 0b1101 0b0101 0b0100 NEWADD Rd Rn Rm</xcoding>
8 :  <xsyntax type="LISA">"X"~NEWADD~ " " Rd "," Rn "," Rm</xsyntax>
9 :  <xsyntax type="GCC" nOp="3" op0="32" op1="32" op2="32">XNEWADD %0,%1,%2</xsyntax>
10 :  <xbehavior>MyNewAdd-behavior.c</xbehavior>
11 : </Xinst>
```

(b)

```
1 : behavior (MyNewAdd)
2 : {
3 :   uint32 a,b,c,d;
4 :   a = READREG(Op1);
5 :   b = READREG(Op2);
6 :   c = (a<<2) + (b>>1);
7 :   d = (a*b) + c + RBUF;
8 :   WRITEREG(Op0) = d;
9 :   RBUF = d;
10 : }
11 :
```

Fig. 5 Input definition file: (a) *MyNewAdd.xml*, (b) *MyNewAdd-behabior.c*.

Table 1 DAA and ARM9 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 DAA and ARM9 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
Co-processor/Instruction Extensions	NO	YES

THUMB instruction-set (16-bit encoding), DSP enhancement instructions (64-bit long multiply and long multiply-accumulate instructions) and swap instructions. On the other side, DAA enhances the instruction-set with instruction extensions to speed up the execution time of particular functions. **Table 1** and **Table 2** respectively show the feature similarities and feature differences between DAA as the Derivative ASIP architecture and ARM9 as the base architecture.

4. Co-processor/Instruction Generator Tool

This section explains the concept behind the Co-processor/Instruction Extension Generator tool. **Figure 4** shows the design flow of the co-processor/instruction extension using our proposed tool. An input definition file is required to describe the specification of the co-processor/instruction being created. This input definition file can be provided either in an XML file or through a GUI-based application software. Complementary files consisting of *<xinst-name>.h*, *t-<xinst-name>.h*, *t-<xinst-name>.c*, *<xinst-name>.lisa* and *<xinst-name>.c* are generated by the tool.

The *<xinst-name>.h* file contains a prototype function of the instruction extension which is required by the application source code to call the instruction extension. *t-<xinst-name>.h* and *t-<xinst-name>.c* are two complementary files used for updating a GCC compiler. The *<xinst-name>.lisa* file contains LISA code to update a LISA-modeled architecture, LISA assembler and LISA linker. The *<xinst-name>.c* file contains ImpulseC codes to generate RTL for a co-processor.

There are three cases of target complementary file generation: instruction extension only, co-processor with instruction extension and co-processor only. The architecture designer needs to specify the target of the output file in the tool prior to the complementary file generation. As an example, a new instruction named *MyNewAdd* will be given to demonstrate the complementary file generation of case one and case two. The same concept is applicable for case three.

4.1 Input Definition with XML file

In our proposed tool, the XML format is used to structure and store the information for the co-processor/instruction specification. **Figure 5** (a) shows the XML input definition file for the given example. Basically, the XML file contains 6 pieces of information as follows:

4.1.1 Instruction Name Definition

The instruction name must be unique to avoid collision during compilation time. This information is indicated by the *<xname>* tag.

Syntax : ***XNEWADD Rd,Rn,Rm***

Format :	Opcode *0b0000110101010100* (16-bit)	NEWADD *0b0010* (4-bit)	Rd (4-bit)	Rn (4-bit)	Rm (4-bit)

Fig. 6 *MyNewAdd* instruction syntax and format.

4.1.2 New Resource Definition

This optional information is used to declare a new hardware resource such as registers, memory or I/O interface, using the *<nresource>* tag. In the example case, we introduced a new register *RBUF* with 32-bit width as defined by the attributes *type="register"* and *width="32"*. If a register array is required instead of a single register, the attribute *size="1"* can be changed to any number to specify the array size.

4.1.3 New Opcode Definition

This is also optional information for declaring a new opcode in a new instruction. This information is indicated by the *<nopcode>* tag. If this tag is enabled, this information will be added to the coding definition and syntax definition as explain later. In the example case, we defined a new opcode named *NEWADD* with a 4-bit data format as shown by the attribute *coding="0b0010"*.

4.1.4 Coding Definition

This information, indicated by the *<xcoding>* tag, defines the instruction format as binary code. The binary code information is very important because it leads to how the processor decodes and executes the instructions properly. In the example case, we defined the instruction format as *"0b0000 0b1101 0b0101 0b0100 NEWADD Rd Rn Rm"*. This means that we have a 32-bit instruction length. The instruction code begins with 16-bit opcodes followed by the 4-bit new opcode *NEWADD* as defined in the *<nopcode>* tag and then 3 operand registers Rd, Rn and Rm as shown in **Fig. 6**. Registers Rd, Rn and Rm are general purpose registers (GPR) where the specifications have been determined in the base processor architecture.

4.1.5 Syntax Definition

This information, indicated by the *<xsyntax>* tag, is necessary for compiler and assembler generation. It defines what the assembly code syntax of the new instruction looks like. There are two types of syntax information. One is LISA syntax, as indicated by the attribute *type="LISA"* and one is GCC syntax as indicated by the attribute *type="GCC"*. The GCC syntax has more attributes to define the number of operands and the data type of each operand because data type information is necessary for the GCC compiler but not for the LISA assembler. In our example case, the number of operands is 3 with all data types being word or 32-bit, as indicated by attributes *nOp="3"*, *op0="32"*, *op1="32"* and *op2="32"*. If the *<nopcode>* tag is enabled, the syntax can be constructed based on the information given in the *<nopcode>* tag. For example, we defined *XNEWADD %0, %1, %2* as the syntax so that it is composed of *X* and *NEWADD*. Symbols *%0, %1, %2* indicate operand 0, operand 1 and operand 2 respectively. *X* is an independent symbol which is changeable with any letter or word specified by the designer.

4.1.6 Behavioral Definition

This information, indicated by the *<xbehavior>* tag, defines the behavior of the co-processor/instruction extension. It is described in the C language and placed in separated file named *<xinst-name>-behavior.c*. In the example case, *MyNewAdd* instruction behavior is described in the *MyNewAdd-behabior.c* file as shown in Fig. 5 (b). It has 3 register operands. The instruction behavior begins with reading two input operands using the *READREG(n)* function as shown in lines 4 and 5, where *n* is the operand index. Local variables *a*, *b*, *c* and *d* are declared to hold intermediate data. In the hardware implementation, these local variables are realized by wires. Besides, all statements in the behavior description file are executed concurrently in the same cycle. This is different from standard C language which executes each statement sequentially. Therefore, the local variables can not be updated more than once. In the example case, variables *a* and *b* are used to hold the values of register operand 1 and register operand 2 respectively. Variable *c* computes shift and add operations given by the equation *c = (a<<2)+(b>>1)*. Finally, variable *d* performs multiplication and addition operations given by the equation *d = (a*b)+c+RBUF*. A new resource register *RBUF* is defined in the *<nresource>* tag. Since *RBUF* is a special register, it can be accessed directly like a local variable without using the *READREG(n)* function. The data in variable *d* is outputted to the register operand 0 by using *WRITEREG(Op0)=d* and also stored in the register *RBUF* by using *RBUF=d* as shown in lines 8–9.

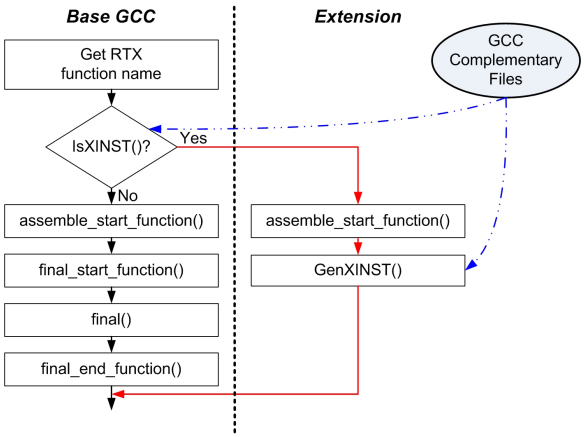

Fig. 7 Extended GCC code generation flow.

4.2 Extended GCC Compiler

The extended GCC compiler here means a base GCC compiler which has been modified to support the Derivative ASIP approach. In our example case, the base GCC compiler is *arm-gcc* and the extended GCC compiler is *daa-gcc*. To extend a base GCC compiler, two modifications are required. First, we have to modify the original base GCC source code in order to make the output assembly code compatible with the LISA assembler. Second, we need to modify the assembly code generation flow to support instruction extension. **Figure 7** shows the internal assembly code generation flow of the extended GCC compiler. *IsXINST()* and *GenXINST()* are two additional functions required for detecting and generating instruction extensions. GCC complementary files which contain information about instruction extension specification are needed to supply data for the *IsXINST()* and *GenXINST()* functions.

GCC compiler generates assembly codes from the Register

(a)

```
13 : OPERATION MyNewAdd IN pipe.DC
14 : {
15 :     DECLARE
16 :     {
17 :       GROUP Rd = {regPC||regLR||regSP||regIP||regFP||reg};
18 :       INSTANCE bypass_Rd_dc;
19 :       GROUP Rn = {regPC||regLR||regSP||regIP||regFP||reg};
20 :       INSTANCE bypass_Rn_dc;
21 :       GROUP Rm = {regPC||regLR||regSP||regIP||regFP||reg};
22 :       INSTANCE bypass_Rm_dc;
23 :       INSTANCE MyNewAdd_ex;
24 :     }
25 :     CODING { 0b0000 0b1101 0b0101 0b0100 NEWADD Rd Rn Rm }
26 :     SYNTAX { "X"~NEWADD~ " " Rd "," Rn "," Rm }
27 :     BEHAVIOR
28 :     {
29 :       bypass_Rd_dc();
30 :       bypass_Rn_dc();
31 :       bypass_Rm_dc();
32 :     }
33 :     ACTIVATION { MyNewAdd_ex }
34 : }
```

(b)

```
36 : OPERATION MyNewAdd_ex POLL IN pipe.EX
37 : {
          ...
48 :     BEHAVIOR
49 :     {
50 :          bypass_Rd_ex();
51 :          bypass_Rn_ex();
52 :          bypass_Rm_ex();
53 :          uint32 a,b,c,d;
54 :          a = alu_in1;
55 :          b = alu_in2;
56 :          c = (a<<2) + (b>>1);
57 :          d = (a*b) + c + RBUF;
58 :          OUT.WBF = 1;
59 :          OUT.BPR1 = Rd;
60 :          OUT.WB1 = d;
61 :          RBUF = d;
62 :     }
63 :     ACTIVATION { writeback_wb }
64 : }
65 :
66 :
```

(c)

```
 1 : void MyNewAddProc(
 2 :   co_stream Op0,
 3 :   co_stream Op1,
 4 :   co_stream Op2)
 5 : {
 6 :   co_stream_open(Op0, O_WRONLY, UINT_TYPE(1));
 7 :   co_stream_open(Op1, O_RDONLY, UINT_TYPE(1));
 8 :   co_stream_open(Op2, O_RDONLY, UINT_TYPE(1));
 9 :
10 :   uint32 a,b,c,d;
11 :   co_stream_read(Op1, &a, sizeof(co_uint32));
12 :   co_stream_read(Op2, &b, sizeof(co_uint32));
13 :   c = (a<<2) + (b>>1);
14 :   d = (a*b) + c + co_register_get(RBUF);
15 :   co_stream_write(Op0, &d, sizeof(co_uint32));
16 :   co_register_put(RBUF,d);
17 :
18 :   co_stream_close(Op0);
19 :   co_stream_close(Op1);
20 :   co_stream_close(Op2);
21 :}
22 :
```

Fig. 8 *MyNewAdd.lisa* file: (a) syntax and coding, (b) behavior, *MyNewAdd.c* file: (c) ImpulseC code for co-processor behavior.

Transfer Language (RTL) representation. In fact, these assembly codes are generated piece by piece per function which is handled by the *rest_of_handle_final()* function. Every function declared in the application source code will be treated as one piece of RTL representation and may generate a piece of assembly codes. Our GCC extension idea relies on this concept. *IsXINST()* is added in the beginning of the *rest_of_handle_final()* function to immediately check whether the declared function is a prototype function corresponding to an instruction extension or not. If so, proper assembly codes will be generated by *GenXINST()*. Otherwise, standard assembly code generation consisting of *assemble_start_function()*, *final_start_function()*, *final()* and *final_end_function()* are executed. Information supplied by the GCC complementary file is useful in assisting the decision in *IsXINST()* and code generation in *GenXINST()*. *assemble_start_function()* handles code segmentation, therefore it is still required before generating assembly code with *GenXINST()*.

IsXINST() needs the function name currently declared in order to recognize an instruction extension. This information is obtained by using *XSTR (XEXP (DECL_RTL (cf_decl), 0), 0)*, where *cf_decl* is the current RTL representation. *XEXP* and *XSTR* are used to convert the RTL representation to an expression and string respectively. The number of operands involved in the current RTL expression is obtained by using *XVECLEN (rtx, 0)*, where *rtx* is an RTL expression.

When an instruction extension is detected, *GenXINST()* generates proper assembly codes as defined in the complementary file. The core information about the complementary file is held in *t-<xinst-name>.h* file. In this file, instruction information is stored in a similar way to the machine description file in the GCC backend. It structures the information as the *XINST* data type which is composed of 4 elements: instruction name, function name, assembly syntax and number of operands. There can be more than one instruction information stored in this file. Appropriate assembly codes are generated based on instruction name matching. For example, if the *MyNewAdd* instruction is detected then the *XNEWADD %0, %1, %2* assembly code will be released. The exact register for operands *%0, %1 and %2* are obtained by using *REGNO(operands[n])*, where *n* is the operand index.

(a)

```
1 : RESOURCE
2 : {
3 :     REGISTER uint32   RBUF;
4 : }
```

(b)

```
 6 : OPERATION NEWADD
 7 : {
 8 :   CODING { 0b0010 }
 9 :   SYNTAX { "NEWADD" }
10 :   EXPRESSION { 0b0010 }
11 : }
```

Fig. 9 *MyNewAdd.lisa* file: (a) new resource, (b) new opcode.

4.3 Extended LISA-modeled Architecture

According to the information about resource, opcode, syntax, coding and behavior given in the XML file, LISA code is automatically generated. **Figures 9** (a), (b) and **8** (a), (b) show the generated LISA code for *MyNewAdd* instruction in case the target of the complementary file generation is instruction extension only.

Figure 9 (a) and (b) show a resource and opcode declaration. A resource is declared with the keyword *RESOURCE* followed by the resource type and its name. In the example case, a 32-bit register resource named *RBUF* is declared. An opcode is declared with the keyword *OPERATION* followed by its name. In the opcode declaration, *OPERATION* only has 3 elements i.e. coding, syntax and expression without behavior. Attribute *coding* in the *<nopcode>* tag of the XML file determines the binary code of the opcode as indicated in *CODING*. If this attribute is empty, the binary code will be defined as *"0b0000"* by default. The *SYNTAX* value is defined by the content of the *<nopcode>* tag. To make it simple, the *EXPRESSION* value is set equal to the *CODING* value.

Figure 8 (a) shows the generated LISA code for syntax and coding declaration. Our base processor architecture has 4 pipeline stages. *"IN pipe.DC"* in *"OPERATION MyNewAdd IN pipe.DC"* indicates that this code will be executed in the decoding stage. The *MyNewAdd* instruction has 3 register operands *Rd*, *Rn* and *Rm* as defined in the *CODING* and *SYNTAX*. These register operands can be either the program counter register, link register, stack pointer register, index register, frame pointer register or GPR as indicated by lines 17, 19 and 21. If a GPR operand is defined, the *bypass_Rx_dc()* function will be added in the *BEHAVIOR* section, where *x* is either *d*, *n* or *m* as indicated by lines 29–31. Function *bypass_Rx_dc()* is used to read the value of reg-

ister *Rx* immediately from the pipeline stages if it is required by the current instruction. When a decoded instruction is matched with *MyNewAdd* coding, it activates the *MyNewAdd_ex* function. This function is executed later in the execution stage as indicated by line 33.

Our tool translates the *MyNewAdd* instruction behavior described in Fig. 5 (b) into LISA code as shown in Fig. 8 (b). *"IN pipe.EX"* in *"OPERATION MyNewAdd_ex POLL IN pipe.EX"* means this function will be executed in the execution stage. Similar to *bypass_Rx_dc()*, *bypass_Rx_ex()* is added in the *BEHAVIOR* section to immediately supply data of register *Rx* from the pipeline stages. *bypass_Rn_ex()*, *bypass_Rm_ex()* and *bypass_Rd_ex()* will result in local variables *alu_in1*, *alu_in2* and *alu_in3* respectively. Therefore, function *READREG(n)* in the behavior description will be translated into *alu_in(i)*, where *n* is the operand index and *i* is either 1, 2 or 3 corresponding to *Rn*, *Rm* or *Rd*. Since in the <*xsyntax*> tag we defined *Rn* as operand 1 and *Rm* as operand 2, the statement *a=READREG(Op1)* and *b=READREG(Op2)* will be translated into *a=alu_in1* and *b=alu_in2* as indicated in lines 54 and 55. In our base architecture, writing data to the GPR register is handled in the writeback stage. A writeback flag, data and destination register must be set before activating the writeback stage. Lines 58–60 show the translation of *WRITEREG(Op0)=d*. *OUT.WBF=1* sets the writeback flag. *OUT.BPR=Rd* defines the destination register as *Rd*. *OUT.WBF=d* sets the data to be written to *d*. *RBUF* is a special register, therefore it can be accessed just like a variable as shown in lines 57 and 61.

4.4 Co-processor Hardware Generation

ImpulseC is a software-to-hardware compiler tool. It is capable of generating HDL (VHDL or Verilog) from C language. The ImpulseC language is a subset of standard ANSI C. However, it includes C extensions in the form of functions and data types. It can describe a standalone function with either single or multiple execution cycles. Instruction extension is always required when a co-processor is employed. In case the target of complementary file generation is a co-processor with instruction extension, both LISA code and ImpulseC code can be generated based on the same XML input definition file. But if the target of complementary file generation is the co-processor only, a separated instruction extension must be defined with a different XML input definition file.

Figure 8 (c) shows the generated ImpulseC code for the function described in Fig. 5 (b). It has 3 I/O interfaces which indicates 3 operands *Op0*, *Op1* and *Op2*. Each interface is declared with the *co_stream* data type. To control the access of the interface, *co_stream_open* and *co_stream_close* are used. Opening and closing the interface must be done before and after accesing the interface as shown in lines 6–8 and 18–20. *O_RDONLY* and *O_WRONLY* in the *co_stream_open* statement determine the interface type either as input or output. Reading a data *x* from interface *in* is done by *co_stream_read(in,&x,sizeof(x))* whereas writing the same data to interface *out* is done by *co_stream_write(out,&x,sizeof(x))*. Lines 11, 12 and 15 respectively show the translation result of statements *a=READREG(Op1)*, *b=READREG(Op2)* and *WRITEREG(Op0)=d* as describe in the behavior description file. In impulseC, a special register is declared by *co_register* data type and created using *co_register_create*. This declaration is done only once in the hardware configuration section which is not shown in this paper. The special register is accessed with *co_register_get* and *co_register_put* for reading and writing data from/to the register as shown in lines 14 and 16.

```
48 :    BEHAVIOR
49 :    {
             ...
54 :         WRITEHC1(alu_in1);
55 :         WRITEHC2(alu_in2);
             ...
58 :         OUT.WB1 = READHC0();
59 :         RBUF = READHC0();
60 :    }
```

Fig. 10 Modified LISA code for interfacing with co-processor.

If the co-processor with the instruction extension becomes the target of complementary file generation, the behavior section in LISA code will be automatically changed as shown in **Fig. 10**. *WRITEHC1(alu_in1)* and *WRITEHC2(alu_in2)* means writing data obtained from register operand 1 (*Rn*) and register operand 2 (*Rm*) to I/O interface *1* and *2* respectively. *READHC0()* means reading data from I/O interface *0*. These I/O interfaces are directly connected to the co-processor interface. Therefore, the instruction extension behavior for accessing co-processor is not more than just an interface between the processor and co-processor. By having compatible interface specifications, the base processor can communicate with the co-processor.

5. Derivative ASIP Implementation

As evidence of our proposed approach, the DAA architecture was verified by simulation on PC and by hardware implementation on FPGA. Two PC based simulations were carried out. First simulation was done by the Processor Debugger Tool [8]. This simulation tool is available in the LISA design framework. It can only simulate the processor architecture modeled in LISA. The second simulation was done by the Sysnopsys Platform Architect Tool [9]. This tool covers a wider scope of architecture models. It works as a system level design simulation, therefore it can also simulate the co-processor part.

Figure 11 shows the DAA system architecture that has been verified. It consists of the DAA processor, on-chip ROM, on-chip RAM, SPI, UART, GPIO, external memory, navigation co-processor and fingerprint sensor. Qsys interconnect [14], a high-bandwidth interconnect to connect intellectual property (IP) functions and subsystems using Avalon standard interface, was chosen as the backbone of the bus system. All components in this system were implemented on FPGA except the fingerprint sensor and external memory.

The fingerprint authentication application [7] was used for testing the DAA architecture. In this application, the AES algorithm becomes one of critical function that needs to be improved because it is used to secure fingerprint template data. It must be fast enough with acceptable execution time. Therefore, some instruction extensions were created in the DAA architecture especially to improve the execution time of the AES algorithm. A co-processor

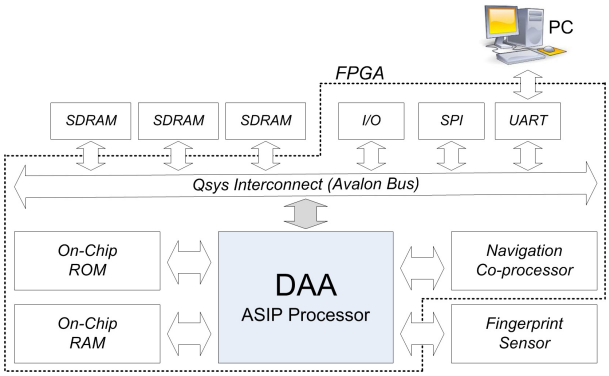

Fig. 11 DAA system architecture.

that performs finger motion computation was also designed for the DAA architecture. This co-processor, called navigation co-processor, is needed to improve the motion computation ability because it demands a high-speed computation which can not be achieved by GPP processor or even by ASIP.

6. Experimental Results and Discussions

In this section, a comparison of the ASIP design approach and experimental results will be discussed to show its effectiveness.

6.1 Design Approach Comparison

Table 3 shows the feature comparison among three ASIP design approaches i.e. the LISA approach [4], Kumura's approach [5] and the Derivative ASIP approach. As shown in the table, the Derivative ASIP approach gains threefold benefit. First, it inherits the merits of hardware generation of the LISA approach. Second, it adopts the advantage of the reuse of existing software development tools concept in Kumura's approach. Third, it improves the design exploration space to support co-processor integration for the base processor architecture.

6.2 Development Time Estimation

Table 4 shows the development time estimation among several ASIP design approaches. In the conventional approach, both the software development tool design and hardware implementation are hand-coded by ASIP designer. It usually takes several months [4], [6] to complete all of the design stages because every stage must be designed manually and sequentially.

The LISA approach employs an architecture description language instead of a hardware description language. With a higher level of abstraction language, a processor architecture model can be designed easily. A set of software development tools and RTL codes for hardware implementation are automatically generated from the high level abstraction language. In previous works, it was reported that the ICORE architecture took 3 months [4] and RISC architecture took 5 months [6] by conventional approach but it only needs 4 weeks by the LISA approach. To refine the LISA modeled architecture including a new instruction it demands another two days [4].

Kuruma's approach proposes a faster way to design ASIP. It reuses the existing processor model and its software development tool set. In Ref. [5], it was mentioned that it took a day or less to write a new intrinsic function. However, this approach can not provide hardware implementation for the developed ASIP.

Table 3 Design approach: (a) LISA approach, (b) Kumura's approach, (c) Derivative ASIP approach.

Feature	(a)	(b)	(c)
Generate hardware implementation	Yes	No	Yes
Reuse existing software development tool set	No	Yes	Yes
Suppport co-processor	No	No	Yes

Table 4 Development time: (a) Conventional approach, (b) LISA approach, (c) Kumura's approach, (d) Derivative ASIP approach.

Work	(a)	(b)	(c)	(d)
Software development tool design	months	weeks	days	days
ASIP hardware implementation	months	weeks	-	weeks
A new co-processor/ instruction design	-	days	days	hours

Table 5 The number of lines of code: (a) input definition in XML file, (b) complementary file generated by tool.

Function		
AES	Number of instructions	10
	(a) Number of lines of input file	238
	(b) Number of lines of output file	1,718
Navigation	Number of instructions	2
	(a) Number of lines of input file	79
	(b) Number of lines of output file	425

The Derivative ASIP approach takes the advantage of the LISA approach and Kuruma's approach. Assisted by the co-processor/instruction extension generator tool, deep knowledge on compiler and hardware design is no longer needed because the complementary files required for updating the compiler and the processor architecture are generated by the tool. This makes the creation of a new co-processor/instruction extension by the Derivative ASIP approach become faster and easier. In our experiment, we took 4 weeks to design the DAA architecture with the LISA design framework. Another 5 days was required to port the GCC and a few hours to create a new co-processor/instruction extension with the Derivative ASIP approach.

6.3 Tool Efficiency

The Co-processor/Instruction Extension Generator tool helps architecture designers describe a new co-processor/instruction with a simpler definition file. This is a more convenient way for non-expert ASIP designers. **Table 5** shows the effectiveness of our proposed tool in terms of the number of lines of code simplification. It appears that the number of lines of code written for input definition files is less than the output files generated by tool. For example, to define 10 instruction extensions for AES algorithm it only needs 238 lines of codes of input definition files. This is much fewer than 1,718 lines of codes of output complementary files. This result shows that employing a co-processor/instruction extension generator tool is very beneficial in reducing the code complexity and saving design time.

6.4 Code Generation

To evaluate the efficiency of the extended GCC compiler, we compared the number of lines of source codes and assembly codes before and after using instruction extensions. The assem-

Table 6 The number of lines of code: (a) original C code, (b) modified C code using instruction extensions, (c) assembly code generated from (a), (d) assembly code generated from (b), (e) hand-optimized assembly code.

Functions	(a)	(b)	(c)	(d)	(e)
aes_subBytes	5	4	30	10	8
aes_subBytes_inv	5	4	30	10	8
aes_addRoundKey	5	4	35	12	12
aes_addRoundKey_cpy	5	4	51	14	14
aes_shiftRows	8	4	72	10	8
aes_shiftRows_inv	8	4	71	10	8
aes_mixColumns	16	4	118	10	8
aes_mixColumns_inv	19	4	155	10	8
aes_expandEncKey	15	4	285	12	12
aes_expandDecKey	14	4	289	12	12
aes256_encrypt	20	20	67	48	48
aes256_decrypt	20	20	62	43	43

bly code generated by the extended GCC compiler was also compared to the hand-optimized one. In this experiment, we used GCC 4.6.0 as the base of the compiler and the AES algorithm as an example application. **Table 6** shows the comparison result. (a) is the original application source code in C. (b) is the modified application source code to employ instruction extensions. (c) is assembly code generated from original code in (a) compiled by the *arm-elf-gcc* compiler. (d) is assembly code generated from modified C code in (b) compiled by the *daa-elf-gcc* compiler. (e) is hand-optimized assembly code.

It can be seen that the number of lines of modified source codes is less than that of the original one and the assembly code generated by *daa-elf-gcc* is less than the one generated by *arm-elf-gcc*. This means that employing instruction extensions may simplify the application source codes and reduce the number of generated assembly codes. If we compare with the hand-optimized result, however, it seems that the assembly code generated by *daa-elf-gcc* is slightly bigger than the hand-optimized one. This is caused by redundant argument handling. In some cases, when an instruction extension is being called, the data passed through as arguments might be ignored because those data have been pre-loaded by previous functions or they have been available in the internal registers. *daa-elf-gcc* compiler can not recognize these kinds of conditions. Therefore, *daa-elf-gcc* will always treat arguments as normal which may generate redundant assembly codes.

Extending a base GCC compiler with instruction extensions will not change the original application source codes a lot. A little modification is sufficient to adjust the application source codes. **Figure 12** shows the application source code of the *aes256_encrypt* function. It looks like the application source code with instruction extensions is quite similar to the original one without instruction extensions.

6.5 DAA Performance

This experiment was carried out to compare the performance of DAA against ARM. Application source codes was compiled by *arm-elf-gcc* and *daa-elf-gcc* for ARM and DAA respectively. **Figure 13** shows the execution cycle of *AES_encryption* and *AES_decryption* functions in each processor. This result shows that compared to ARM, DAA greatly reduces the number of execution cycles. DAA improves its performance by up to 27 and 36 times better than ARM because it employs instruction extensions. Running at 100 MHz, DAA will be able to carry out 10 fingerprint data encryption within 30 millisecond which is more acceptable than 1.2 seconds in ARM.

(a)

```
void aes256_encrypt(aes *ctx,uint8 *buf)
{
 uint8_t i, rcon;
 aes_addRoundKey_cpy (buf,ctx->ekey,
                      ctx->key);
 for(i=1, rcon=1; i<14; ++i)
 {
  aes_subBytes(buf);
  aes_shiftRows(buf);
  aes_mixColumns(buf);
  if(i&1) {
   aes_addRoundKey(buf,&ctx->key[16]);
  } else {
   aes_expandEncKey(ctx->key,&rcon);
   aes_addRoundKey(buf,ctx->key);
  }
 }
 aes_subBytes(buf);
 aes_shiftRows(buf);
 aes_expandEncKey(ctx->key,&rcon);
 aes_addRoundKey(buf,ctx->key);
}
```

(b)

```
void XINST_aes256_encrypt()
{
 uint8_t i, rcon;
 XINST_aes_addRoundKey_cpy(0);

 for(i=1, rcon=1; i<14; ++i)
 {
  XINST_aes_subBytes();
  XINST_aes_shiftRows();
  XINST_aes_mixColumns();
  if(i&1) {
   XINST_aes_addRoundKey(16);
  } else {
   XINST_aes_expandEncKey(&rcon);
   XINST_aes_addRoundKey(0);
  }
 }
 XINST_aes_subBytes();
 XINST_aes_shiftRows();
 XINST_aes_expandEncKey(&rcon);
 XINST_aes_addRoundKey(0);
}
```

Fig. 12 Source code comparison: (a) without instruction extensions, (b) with instruction extensions.

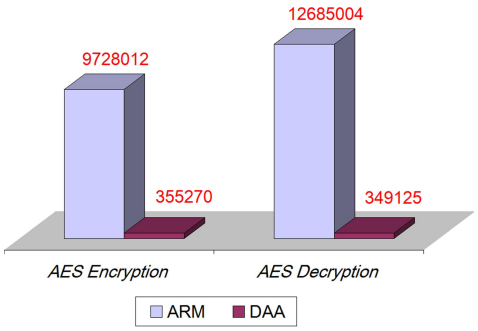

Fig. 13 Execution time of AES algorithm in *cycle* unit.

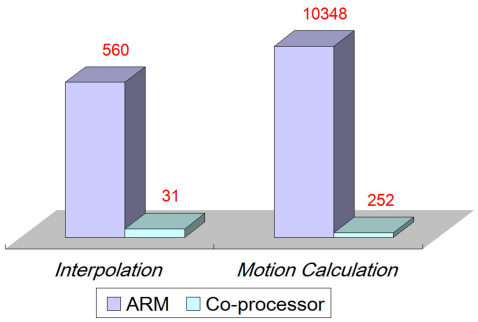

Fig. 14 Execution time of navigation algorithm in *us* unit.

Figure 14 shows the execution time of the navigation algorithm on ARM and navigation co-processor in *us* unit. Employing a co-processor, DAA speeds up the execution time of *interpolation* and *motion calculation* functions 18 and 41 times faster than ARM. With fingerprint sensor resolution 363 dot-per-inches, it will be able to detect finger motion of up to 27.38 cm/s.

6.6 FPGA Implementation

The DAA system architecture depicted in Fig. 11 was verified on the Altera DE2-115 FPGA development board. This hardware verification confirmed that our proposed approach works well at

Table 7 Area and power comparison.

Item	ARM9	DAA
Area	0.613 mm^2	0.571 mm^2
Power	9.5 mW	9.4 mW

the hardware level. Device utilization report indicates that running at 100 MHz clock consumes 24 K LEs, 1,503 registers, 35 I/O pins and 3,145,728 bit (≈ 384 KB) memory usage. This memory requirement consists of 128 KB for on-chip ROM and 256 KB for on-chip RAM.

6.7 Area and Power Estimation

Area and power consumption were also evaluated to see how much hardware efficiency can be achieved. To estimate the area and power consumption of the DAA processor, the RTL code generated by the LISA design framework was synthesized by using the Synopsys Design Compiler tool [10]. **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 [13].

Compared to ARM9 as its base architecture, DAA performance could be increased without additional cost of area and power because DAA architecture has been optimized. It removed redundant resources but enhanced with instruction extensions. In other words, unused resources were compensated by other more useful resources. This result shows that our approach can provide a practical method to modify processor architecture to optimize its performance for certain target applications.

7. Conclusion

A practical ASIP design method with the Derivative ASIP approach was proposed. This approach not only overcame the problem of hardware and compiler generation in the previous approaches but also broadened the architecture exploration space to accommodate co-processor integration. A new tool called Co-processor/Instruction Extension Generator Tool was also developed. It helps ASIP designers create a new co-processor/instruction extension easily. By using our approach, ASIP design exploration can be done in a shorter time.

References

[1] Marwedel, P.: The MIMOLA Design System: Tools for the Design of Digital Processors, *Proc. 21st Conference on Design Automation*, pp.587–593 (1984).

[2] Hadjiyiannis, G., Hanono, S. and Devadas, S.: ISDL: An Instruction Set Description Language for Retargetability, *Proc. 34th Conference on Design Automation*, pp.299–302 (1997).

[3] Fauth, A., Van Praet, J. and Freericks, M.: Describing Instruction-set Processors using nML, *Proc. European Design and Test Conferece*, pp.503–507 (1995).

[4] Hoffmann, A., Kogel, T., Nohl, A., Braun, G., Schliebusch, O., Wahlen, O., Wieferink, A. and Meyr, H.: A Novel Methodology for the Design of Application-Specific Instruction-set Processors using a Machine Description Language, *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, Vol.20, No.11, pp.1338–1354 (Nov. 2011).

[5] Kumura, T., Taga, S., Ishiura, N., Takeuchi, Y. and Imai, M.: Software Development Tool Generation Method Suitable for Instruction Set Extension of Embedded Processors, *IPSJ Trans. System LSI Design Methodology*, Vol.3, pp.207–221 (Aug. 2010).

[6] Bailey, B., Martin, G. and Piziali, A.: *ESL Design and Verification: A Prescription for Electronic System Level Methodology*, Morgan Kaufmann (2007).

[7] Mostafa, A.M., Li, D. and Kunieda, H.: Minutia Ridge Shape Algorithm for Fast On Line Fingerprint Identification System, *Proc. Symposium on Intelligent Signal Processing and Communication Systems*, Vol.2, pp.593–598 (2000).

[8] Synopsys Inc.: Processor Designer and Processor Debugger, available from ⟨http://www.synopsys.com⟩.

[9] Synopsys Inc.: Synopsys Platform Architect, available from ⟨http://www.synopsys.com⟩.

[10] Synopsys Inc.: Synopsys Design Compiler, available from ⟨http://www.synopsys.com⟩.

[11] Impulse Accelerated Technologies: ImpulseC C-to-FPGA Tool, available from ⟨http://www.impulseaccelerated.com⟩.

[12] GNU Project: GNU Compiler Collection (GCC), available from ⟨http://www.gcc.gnu.org⟩.

[13] ARM Holdings plc.: ARM946 performance, available from ⟨http://www.arm.com⟩.

[14] ALTERA Corporation: Design Tools and Services, available from ⟨http://www.altera.com⟩.

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 the Kunieda-Isshiki Laboratory, the Department of Communications and Computer Engineering, Tokyo Institute of Technology, Japan. His current research focuses on processor architecture and SoC design is especially for fingerprint authentication applications.

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

Tsuyoshi Isshiki received his B.E. and M.E. degrees from Tokyo Institute of Technology in 1990 and 1992, respectively, and his Ph.D. in Computer Engineering from the University of California at Santa Cruz in 1996. He is currently an Associate Professor at Tokyo Institute of Technology, the Department 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 degrees from Tokyo Institute of Technology, Tokyo, Japan in 1973, 1975 and 1978, respectively. Since 1978, he has been with the 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 of IEICE and a senior member of IEEE CAS Society.