IPSJ Transactions on System LSI Design Methodology Vol. 3 161–178 (Aug. 2010)

Invited Paper

Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)

Masaharu Imai,[†1] Yoshinori Takeuchi,[†1]
Keishi Sakanushi[†1] and Nagisa Ishiura[†2]

This paper introduces the concept and technology of Application-domain Specific Instruction-set Processor (ASIP). First, VLSI design trend over the decades is overviewed and processors are shown to be expected one of the main components in the System Level Design. Then, the advantage of ASIP over General Purpose Processor (GPP) and Application Specific Integrated Circuit (ASIC) is illustrated. Next, processor hardware description synthesis technology, application program development tool set generation technology, and processor architecture optimization technology are outlined. Then, as an ASIP development environment example, ASIP Meister is explained. Next, an application of ASIP to medical and healthcare is introduced. Finally, the possibility of ASIP as an important component of Multi Processor SoC (MPSoC) is discussed.

1. Introduction

The progress of semiconductor technology results in the exponential increase of logic transistors in the VLSI chip. According to the report from ITRS [1], the compound growth rate of the number of logic transistors in a VLSI chip is still 58%; which means that the density of logic transistors in a VLSI chip will become double in every 18 months, or 100 times as large in every 10 years. This phenomena is called "Moore's Law" named after Gordon E. Moore from Intel Corporation [2].

On the other hand, according to a report from SEMATECH [3], the compound growth rate of the design productivity (transistors/staff-month) is about 21%. Therefore, the design staff-months of a typical leading-edge VLSI will increase explosively about 30% every year, and causes a serious problem called "Design Productivity Crisis".

In order to overcome this difficulty, following strategies have been taken so far:

(1) **To make the abstraction level of design description higher as possible,**
(2) **To make the design reuse rate higher as possible,** and
(3) **To employ the formal or semi-formal verification methods and tools to verify the design description.**

Figure 1 shows the trends in design abstraction level. Roughly speaking, design method has been changed every decade. Each design method has its own design granularity and the form of description.

1.1 Pre-Register Transfer Level Design

In the 1960s, the design of integrated circuit (ICs) was described as **photo masks** of physical layers of ICs. In this design method, the physical structure of IC is described as a collection of photo masks. In the 1970s, the design was described at **transistor circuit** level. The design was described as a **graph** using nodes as electronic circuit components, such as transistor, registers, capacitors, and inductors, and edges as wires (connections). Such circuits are essentially the **analog circuits**. In the 1980s, the design was described at **logic gate** level. The design was described as a **graph** using nodes as logic gate level components

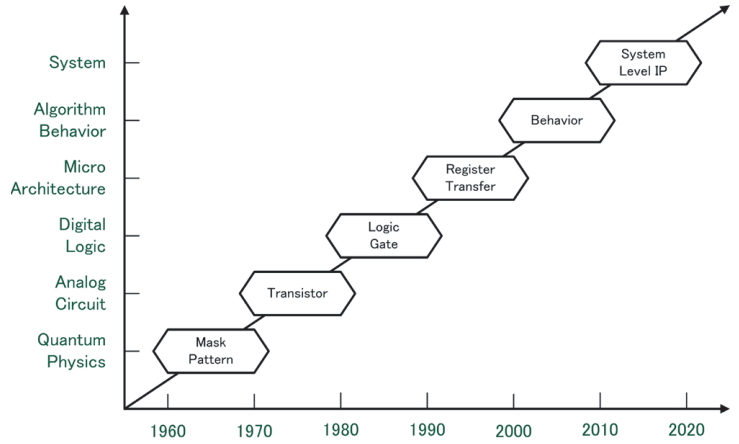


Fig. 1 Trend of abstraction level of Integrated System Design.

†1 Graduate School of Information Science and Technology, Osaka University
†2 Graduate School of Science and Technology, Kwansei Gakuin University

and edges as wires (connections).

1.2 1990s: Register Transfer Level Design

In the late 1980s, **logic synthesis** technology brought a revolution in the VLSI design, where the design was described as **text** in a **Hardware Description Language** (**HDL**), such as VHDL and Verilog HDL. Text representation is a sequence of characters, which is the simple and elementary data structure that can be easily handled by computers.

This design method has the following advantage:

(1) Not only the **structure** of the design entity can be described, but the function of the design entity can also be described in the HDL.
(2) The function of the design entity can be a **data flow** or a **sequential behavior**.
(3) HDL descriptions are **simulatable** by a logic simulator on a computer.
(4) There is a coding guide line for **logic synthesis**. Properly described HDL descriptions according to the coding guide line can be translated into logic gate level descriptions.

1.3 2000s: Behavior Level Design

In the decade of 2000, so-called **behavior synthesis** or **high-level synthesis** has been adopted. Where the design is described at higher level of abstraction than in the conventional HDL description for logic synthesis. For example, the design description in this method can be a conventional behavioral one in an SPL (Software Programming Language), such as ANSI C, or some extension of an SPL, such as SpecC.

While the design productivity could be improved by the behavior synthesis compared to conventional logic synthesis using HDL, this method has following limitations:

(1) Data type is limited to that of SPL. In order to optimize the design, data types, such as bit width of a data item, should be specified.
(2) Parallelism and concurrency of processes, synchronization among processes cannot be specified explicitly. In order to extract such information, design description should be analyzed carefully.
(3) Structural information cannot be described explicitly.

Several different approaches could be taken to overcome these limitations. One approach is to design a new language for behavior synthesis; this approach is taken by SpecC [4]. Second one is to extend some existing language such as C; this approach is taken by Bach-C, BDL, Handel-C, etc. Another one is to use an object oriented language, such as C++. SystemC is a new language based on C++, where the language syntax is the same as C++, but new class libraries are introduced to handle new data types and operations, to perform event driven simulation [5].

The most suitable role of behavior synthesis would be to design dedicated hardware modules. These modules can be used as building blocks in the system design.

1.4 2010s: System Level Design

Design methods can be classified by the granularity of the building blocks used in the design. In the System Level Design, the granularity of the building block is quite large. Typical system level components are instruction set processors, dedicated hardware modules, memory modules, peripheral modules, etc.

In the SoC (Systems on a Chip) design, analog circuits, RF (Radio Frequency) circuits, MEMS (Micro Electro Mechanical Systems) sensors, and actuators can also be integrated on the same die. If these components can be fabricated separately, they can be combined with digital modules using SiP (Systems in a Package). Even in this case, system design should be performed taking all these modules into account.

Among these modules, instruction set processors play an important role both in SoC and SiP design. Design cost of SoC and SiP increases as the newly designed modules become large and complex. Initial fabrication cost including mask cost is tremendously increasing for fine grained nano meter fabrication process. The total cost of the product is defined by the sum of these initial cost and production cost.

The largest advantage of the processor is the programmability, that is, the functionality of the system can be defined and modified after the system is fabricated. Then the systems including instruction set processors can be applied to wider area of applications compared to the systems that does not include instruction set processors. As a result, the production count of such systems will increase and the cost of the systems will be reduced.

1.5 Configuration of the Paper

In the rest of this paper, the advantage of ASIPs compared with general purpose processor and dedicated hardware (ASIC) will be explained first. Then Architecture Description Languages are surveyed. Next, processor description synthesis methods and application program development tool set generation methods are introduced. Then, ASIP Meister is explained as an ASIP development environment example. Next, an SoC that includes an ASIP for biomedical information sensing system is introduced. Finally, the possibility of the ASIP as an important component of Multi Processor SoC (MPSoC) is discussed.

2. Advantage of ASIP

Application systems can be implemented in various ways. Some of the typical implementation methods are as follows:

(1) **Pure Software Approach using GPP (General Purpose Processor)**,
(2) **Dedicated Hardware Approach using ASIC (Application Specific Integrated Circuit) along with GPP**, and
(3) **Dedicated Processor Approach using ASIP (Application-domain Specific Instruction-set Processor)**.

These approaches have advantages and disadvantages as shown in **Fig. 2**.

2.1 Pure Software (GPP) Solution

In this approach, the functionality of the target system is implemented as a software that is executed on a GPP.

The largest advantage of this approach would be the flexibility and extensibility of the system's function. The software can be modified much more easily than hardware. Flexibility and extensibility are important to implement embedded systems to reduce the development cost of the systems by re-using the same hardware.

The drawback of this approach would be the power consumption and heat dissipation. When the performance requirement to the system is very high, we need to use a high performance processors, which consume a lot of electric energy and generate a lot of heat dissipation.

In some of the embedded applications such as portable equipments, power consumption and heat dissipation are very critical. If the power consumption is very large, we need to use a battery with a large capacity, which are usually large sized, heavy, and expensive. And, if the heat dissipation is large, we need to apply some cooling mechanism to remove the heat, which will make the final product large, heavy, and expensive.

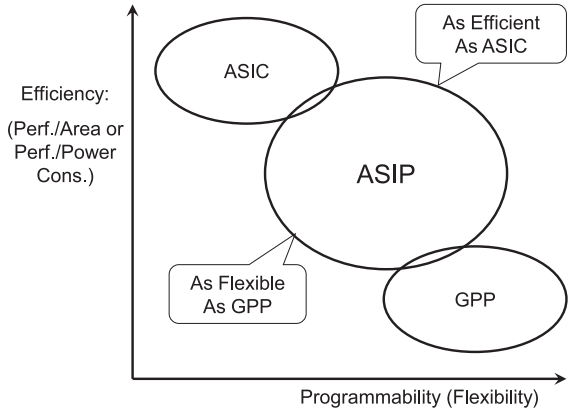


Fig. 2 Advantage of ASIPs.

2.2 Dedicated Hardware (ASIC) Solution

In this approach, dedicated hardware modules (ASICs) are designed to implement the critical functionality.

The advantage of this approach, compared with the Pure Software Approach, is the efficiency both in area and power consumption. Generally speaking, when the same functionality can be implemented by some dedicated hardware (ASIC) and by some software to be executed on GPP, higher performance and lower power consumption can be expected by the former implementation than the latter.

The limitation of this approach is the lack of flexibility and extensibility of the functionality of the system. Because ASICs are designed to execute some specific functions, it is difficult to execute other functions on the same hardware. A GPP is also used in this approach to control ASICs and to ease the limitation by executing other operations than those executed by ASICs.

2.3 Dedicated Processor (ASIP) Solution

In this approach, ASIP (Application-domain Specific Instruction-set Processor) is used to perform the required functionality. Typical ASIPs are based on RISC (Reduced Instruction Set Computer) architecture and enhanced by adding dedicated instructions and special purpose registers.

One of the advantages of this approach over the Pure Software Approach using GPP is that ASIP can be more efficient than GPP in terms of area and power consumption. That is, if the same technology is used to design and implement (or fabricate) both processors, an ASIP that has the same functionality and performance of a GPP can be implemented within a smaller chip area than the GPP, and power consumption of the ASIP can be smaller than that of the GPP.

One of the advantages of this approach over the Dedicated Hardware Approach using ASIC is the flexibility and extensibility. As mentioned in the beginning of this subsection, if the ASIP is an extension of a RISC architecture by adding dedicated instructions, the ASIP has the same flexibility and extensibility as the Pure Software Approach using GPP. That means the ASIPs can be applied to much wider application systems of the same domain than ASICs, which will reduce the cost of ASIPs.

While this approach has various advantages over other two approaches, there are following challenging issues to make this approach effective in the real world:

- How to describe architecture of the processor
- How to generate hardware description of the processor
- How to generate application program development tool set, such as compilers and simulators, for the processor
- How to optimize the architecture of the processor for a given application domain

We will discuss these issues in the following sections.

3. Architecture Description Language

Processor is one of the most sophisticated hardware components used in SoC (System on a Chip). Then it requires a long term effort even for an experienced designer to explore an excellent architecture and to describe the architecture in an HDL (Hardware Description Language). Moreover it is a hard work to debug and verify the design description manually. This is one of the reasons to generate HDL description of processors from a higher abstraction level description.

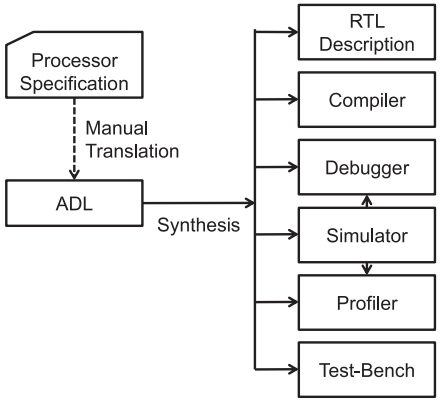


Fig. 3 Role of ADLs.

The language used to describe processor architecture is called Architecture Description Language (ADL)[6]. The history of ADL is relatively old. ISPS (Instruction Set Processor Specifications)[7),8)] is one of the oldest ADL's. The purpose of ADL is to synthesize a hardware description (HDL) of target processor, a compiler, cycle-accurate or untimed instruction set simulators, a test-bench and other utilities such as debugger, profiler, and to verify the design of processors in a formal method. In order to achieve these goals, ADL defines structures, operations, instructions, data types, and interpretation rules (semantics) in the processor. The role of ADLs in the processor synthesis and application program development tool set generation is shown in **Fig. 3**.

The final object of ADL is the design space exploration of not only single processors but also systems which includes several processors in it. By using ADL, some sophisticated design development tools mentioned above are generated and ADLs enable design space exploration of processors. However, it is not easy to cover all object required for design space exploration. In Objective-based classification, ADLs are classified into four categories; compilation-oriented ADLs, simulation-oriented ADLs, synthesis-oriented ADLs, and validation-oriented ADLs. On the other hand, from the contents point of view, ADLs can be classified into three

categories: Structural ADL, Behavioral ADL, and Mixed ADL. In the following sections, advantages and limitations of these descriptions are introduced.

3.1 Structural ADL

Structural ADL is a synthesis-oriented and validation-oriented ADL according to the objective-based classification. MIMOLA is one of structural ADLs. MIMOLA [9)–11)] has been developed since 1976 originally targeted for digital recording system at Kiel University. MIMOLA defines designs using design entities called modules which include port and behavior. By using modules and its connections, target system structure is specified. Owing to the structural based description like HDL RT level description, MIMOLA is suitable for synthesizable HW description synthesis and cycle-based simulator, but it is not easy to extract instruction set information to generate a compiler for the processor from its structural ADL.

3.2 Behavioral ADL

Behavioral ADL is a compilation-oriented and simulation-oriented ADLs. ISDL [12)] is one of behavioral ADLs. ISDL is specified by instruction word format, global definitions, storage resource, assembly syntax, and constraints and optimization information to the compiler. Though storage resources are the structural information, other information is behaviors for generating software developing tools like assembler, simulator, and compiler.

Behavioral ADL, such as ISDL, is suitable to synthesize an instruction set level simulator or a compiler because the necessary information on the instruction set is directly described in the behavioral ADL. However, it is still a challenge to synthesize a high quality HDL description from behavioral ADL description.

3.3 Mixed ADL

The early ADLs were either structure-oriented or behavioral-oriented, and they are used for specific tasks in the previous sections. The later ADLs become Mixed ADL which combines the benefits of structural ADL and behavioral ADL. Structural description makes it easy to generate a good quality HDL description for HW synthesis, and behavioral description makes it easy to describe necessary information on the instruction set to generate a compiler and instruction set level simulator. Mixed ADLs will cover the characteristics of ideal ADLs; compilation-oriented, simulation-oriented, synthesis-oriented, and validation-oriented ADLs. Many ADLs are included in this category, such as nML [13),14)], LISA [15),16)], EXPRESSION [6),17)], GNR [40)], and the ADL used in ASIP Meister [18),19)].

4. Processor Description Synthesis

There are several approaches to synthesize processor hardware description, depending on the architecture models of processor:

- Fixed Architecture Model Approach,
- Flexible Architecture Model Approach, and
- Free Architecture Model Approach.

There is a trade-off between the flexibility of the architecture model and the complexity of synthesis algorithm. Generally speaking, if the architecture model is more restricted, processor architecture description will become simpler, and processor hardware description synthesis will become easier. However, of course, if the processor architecture model is too restricted, we will lose the opportunity to implement an efficient processor which is most suitable for the given application.

4.1 Fixed Architecture Model Approach

This is the simplest approach to generate processor description. In this approach, a base processor is provided by the generation tool provider, that is, the main part of the architecture framework of the processor, such as basic instruction set, instruction pipeline stage structure, general register file, is fixed a priori. ASIP designers add their own instructions for specific application to the base processor.

In this approach, both architecture description generation and application program development tool set generation are easier than other two approaches. This approach is employed by PEAS-I [20),41)] for scalar processor generator, PEAS-II [21)] for VLIW processor generator, Xtensa [22)], and ARC Cores. MIPS Technologies also provide HDL description of their processor core as configurable processors.

4.2 Flexible Architecture Model Approach

In this approach, a flexible architecture model with reasonable assumptions is utilized. The architecture model used in this approach has more freedom than the one used in the Fixed Architecture Model Approach. For example, pipeline stages is configurable, that is, the number of stages can be chosen, the micro

operations performed in each pipeline stage can be specified, and special purpose registers can be added, by the ASIP designers.

Processor description generation and application program development tool set generation are easier than those in the Free Architecture Model Approach, due to the assumptions on the architecture model.

The advantage of this approach over the Fixed Architecture Model Approach is the flexibility of processor architecture. For example, the number of pipeline stages can be decided by the architect, special purpose registers can be added, and complex instructions can be implemented relatively easily. And design productivity can be very high.

This approach is suitable for processor design from the scratch, as well as for modification or extension of pre-designed architecture. It is relatively easy to implement instruction set compatible processors of a legacy processor, but it might be difficult to implement a clock cycle compatible "clone" of a legacy processor, because the architecture model has some restriction.

This approach has another advantage. When the architecture options are parameterized, window based GUI can be utilized efficiently and design productivity can be very high. This approach is employed by ASIP Meister (PEAS-III) [18),19)]. The details of ASIP Meister are explained in Section 7.

4.3 Free Architecture Model Approach

In this approach, less assumptions are settled on the architecture model [23)]. Therefore this approach provides the largest freedom or variety in the processor architecture among three approaches discussed in Section 4. Legacy old fashioned CISC processors could be designed using this approach. This is the best advantage of this approach.

However, there are limitations of this approach. If we want to specify every detail of the processor, the abstraction level of the description will become low as register transfer level (RTL) and the design productivity would be the same level as that of the design methodology using conventional HDLs.

On the contrary, if the abstraction level of the processor description is as high as behavioral level, RT level HDL description should be synthesized. In this case, the quality of generated HDL might not be good enough compared to other approaches, because the behavioral synthesis technology is still challenging and not matured yet as the conventional logic synthesis technology.

5. Application Program Development Tool Set Generation

In order to accomplish an ASIP development project, we need to prepare some AP (Application Program) development tool set for application system developers. In the case of ASIP's, AP development tools should be generated by the ASIP development system, or someone should develop such AP development tools manually [39),42)].

Typical AP development tool set includes following tools:

- Compiler,
- Assembler and Linker,
- Instruction Set level Simulator (ISS),
- Profiler, and
- Debugger.

For large and complex application systems, compiler is essential in order to enhance the design productivity and to shorten the development time. On the contrary, if the application system's functionality is not so complicated, compiler is optional because application programs could be developed in assembly language.

Instruction set level simulator is essential to estimate the performance of the system while the design space exploration is performed, and it is also necessary to validate and improve the application program.

Debugger is necessary to validate or to find the cause of problems if any malfunctions were observed. ICE (In Circuit Emulator) is sometimes used to observe the internal status of a processor.

5.1 Role of Compiler in the ASIP Development

Among the AP development tools, compiler is the most complicated system, and compiler generation is still a very challenging task even in the current information technology era. Generally speaking, requirements to compilers are the efficiency of the object code (shorter length of code, smaller execution cycles, or smaller power consumption) as well as the efficiency of compilation (shorter compilation time). And there is a trade-off between these requirements.

There are two different roles expected to compilers for HW/SW Codesign of ap-

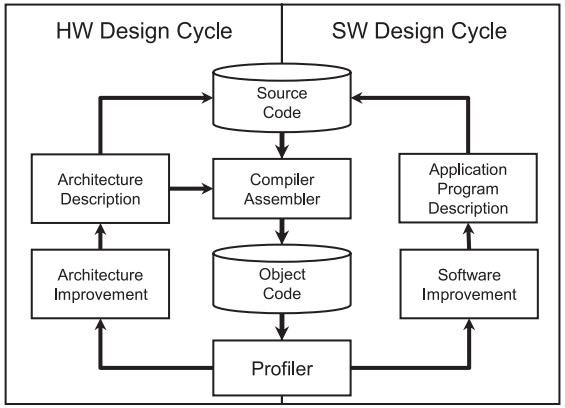


Fig. 4 Role of compiler and profiler in HW/SW Codesign.

plication specific systems. The role of compiler and profiler in HW/SW Codesign is illustrated in **Fig. 4**.

The first role is to assist system designers to explore the design space and to look for an optimum architecture. In this case, shorter compilation time would be more important than the efficiency of object code, if the compiler could provide the designers with a fair estimation of the final code size, execution cycles, and power consumption.

The second role is to generate efficient object code for the final product. In this case, shorter compilation time is less important than the quality of object code, provided the compilation time is not too long.

5.2 Approaches to Compiler Retargeting

There are several ways to obtain a compiler for an ASIP. One is to develop or generate a new compiler for the ASIP. And another way is to retarget some existing compiler for the ASIP, where some portion of the original compiler is modified or extended for the ASIP. In this section, the term "retargeting" is used as to include "generation".

Compiler retargeting systems can be classified into three categories as follows [24]. They employ different approaches and there is a tradeoff among these approaches.

Automatically Retargeting: In this method, a "generic" compiler itself contains a set of well defined parameters or switches that enables complete retargeting to new processors. A full range of knowledge about the target processor architectures should be contained in the compiler a priori. Retargeting time is of the order of seconds to minutes. This approach might look ideal, but it is quite challenging to develop a good compiler by this approach that can generate efficient object code.

Compiler User Retargeting: In this method, compiler user is supposed to provide a compiler generator that retargets a compiler from the specification of processor architecture and instruction set. The compiler generator produces the source code of the target compiler, which will be compiled by a resident compiler. This approach is practical and employed by popular compiler retargeting system, such as GCC [25] from Free Software Foundation and CoSy [26] from ACE (Associated Computer Experts bv). These systems can generate a good compiler that generates good object code. Retargeting time is of the order hours to days. Because the generated compilers focus on target machine independent code optimization, it would be necessary to develop a code optimization path that performs target machine dependent code optimization.

Compiler Developer Retargeting: This is a conventional approach. Compiler developer retargets the compiler manually. Therefore, retargeting time is of the order of weeks to months. The quality of compiler developed by this approach will be better than those by other two approaches. But a large amount of labor is necessary to develop a compiler manually, which will take long time and the development cost is very expensive.

6. Processor Architecture Optimization

One of the most exciting challenges on ASIP design is to decide an appropriate instruction-set which optimizes certain objective, e.g., performance, area, power consumption, and their tradeoff. Many instruction-set optimization methods are proposed so far [27)–32]. In general, the first step of processor optimization is application program analysis step, and the second step is instruction-set optimization step.

6.1 Application Program Analysis

In the application program analysis step, designer grasps relations and frequencies of operations and data. The application program analysis step is generally formalized as follows.

Application program analysis

Input:

- Application program
- Related data
- Basic instruction

Output:

- Data type
- Basic block structure
- Relations between instructions and data
- Frequency of instructions and basic blocks

Application program analysis methods are classified into static analysis methods and dynamic analysis methods.

Generally, static analysis methods compile application programs by basic instructions, construct basic block structure and data flow graphs (DFG), and extract frequency of basic blocks, instructions, or instruction patterns. To grasp more accurate frequency of operations, dynamic profiling is used. Dynamic analysis methods execute application programs with related data on simulator and profile frequency of them.

6.2 Instruction Set Optimization

Bottlenecks and frequently executing operation patterns are candidates of hardware implementations which are used as custom instructions because it is expected that custom instructions reduce execution time and power consumption. However, additional hardware requires more area. Therefore, deciding implementation can be formalized as an optimization problem when there are several candidates of implementations. Generic definition of Instruction-set Optimization problem is as follows.

Instruction-set optimization problem

Input:

- Analyzed results from Application program analysis step
- Hardware and software database
- Constraints

Output:

- Instruction-set
- Implementations (hardware and software modules)

Objective:

- Execution cycle, area, or power consumption or combinations of them

Generally, since execution cycle, area, and power consumption are under trade-off relation, instruction-set optimization step aims to optimizing one of them or to balancing them.

As one of examples, we introduce an instruction-set optimization method [28] developed for PEAS-I. Let $x_i \in M_i$ be implementation of basic operation i where M_i denotes set of implementations of basic operation i. The method finds implementation vector

$$X = (x_0, x_1, \cdots, x_n)$$

which minimizes the objective function

$$T(X) = \sum_{j=1}^{N} \{F_j \times (t(B_j, X) + c_j)\} - b$$

subject to the constraints

$$\sum_{x_i \in S} a(x_i) \leq A_{\max},$$

and

$$\sum_{x_i \in S} p(x_i) \leq P_{\max}.$$

N denotes the total number of basic blocks, F_j denotes the execution frequency of basic block B_j, $t(B_j, X)$ denotes execution cycles needed to execute B_j using configuration X, and $a(x_i), p(x_i)$ denote area and power consumption of x_i, respectively. c_j denotes clock cycles due to delayed branch in B_j and b denotes clock cycles due to un-taken branches in the whole program. Note that these values, except $t(B_j, X)$, are derived from application program analysis step by simulating application program and related data. The number of possible im-

plementation vector X is too huge. Therefore, the method reduces search space based on dependencies between operations and basic blocks derived from application program analysis step and optimizes implementation vector X.

7. ASIP Development Environment Example

In this section, an ASIP development environment named ASIP Meister is introduced.

7.1 Feature of ASIP Meister

ASIP Meister [18),19)] is an ASIP design environment developed by a research group from Osaka University, Shizuoka University, Kwansei Gakuin University, Tsuruoka National Collage of Technology, and Osaka Electro-Communication University. The project had started in 1996 as one of the first collaboration research projects sponsored by STARC (Semiconductor Technology Academic Research Center), which is a consortium of ten semiconductor industries in Japan. ASIP Meister is now a commercial EDA tool distributed by ASIP Solutions, Inc., which is one of the startup companies from Osaka University [33)].

One of the distinguished features of ASIP Meister are a window based GUI (Graphical User Interface), that can be used quite effectively for design specification of both structural and behavioral descriptions of processors.

The input through GUI is stored in a file as XML description, then ASIP Meister generates HDL description, in VHDL, Verilog HDL, and SystemC. ASIP Meister also generates GNU based application program development tool set.

7.2 The Efficiency of Window Based GUI

Window based GUI has several advantages based on a human engineering. First of all, if we use window based GUI properly, design time can be reduced drastically by taking advantage of visual checking of description. Secondly, human readable documentation can be generated easily in the form of table or tree, for example.

In the structural description of a processor, components can be specified by a relatively small number of parameters. For example, general register file can be specified by the bit width of each word, number of words in the register file, number of read ports, and number of write ports. In such a case, component specification can be described very easily by using a GUI. Some snap shots of design windows in ASIP Meister are shown in **Figs. 5**, **6**, **7** and **8**.

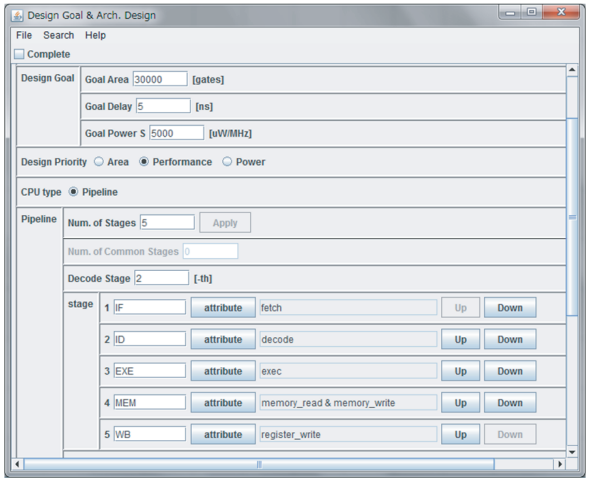


Fig. 5 Processor architecture parameter entry window.

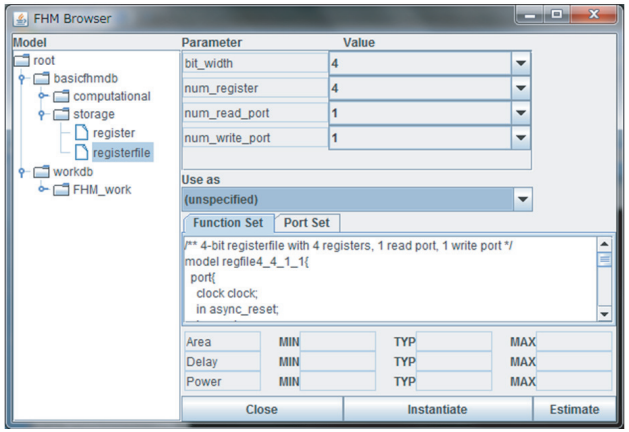


Fig. 6 Register file instantiation using GUI.

Behavioral description can also be described very nicely using GUI. Figure 8 shows an example of instruction behavior description using the GUI of ASIP

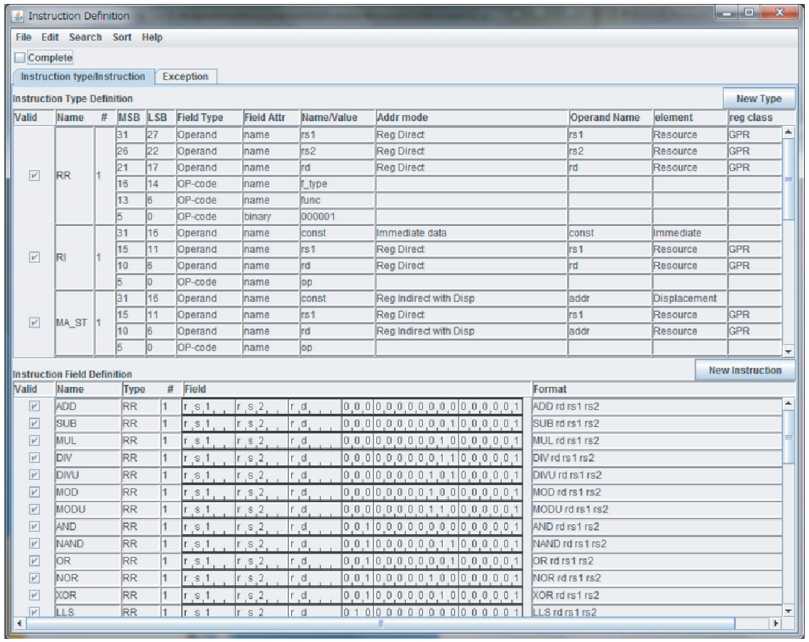


Fig. 7 Instruction format definition and operation code assignment window.

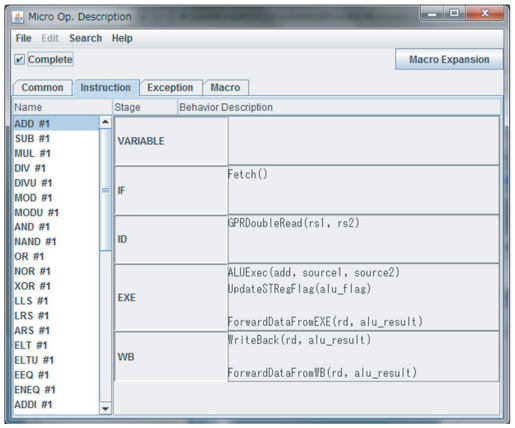


Fig. 8 Behavioral description example of the ADD instruction using GUI.

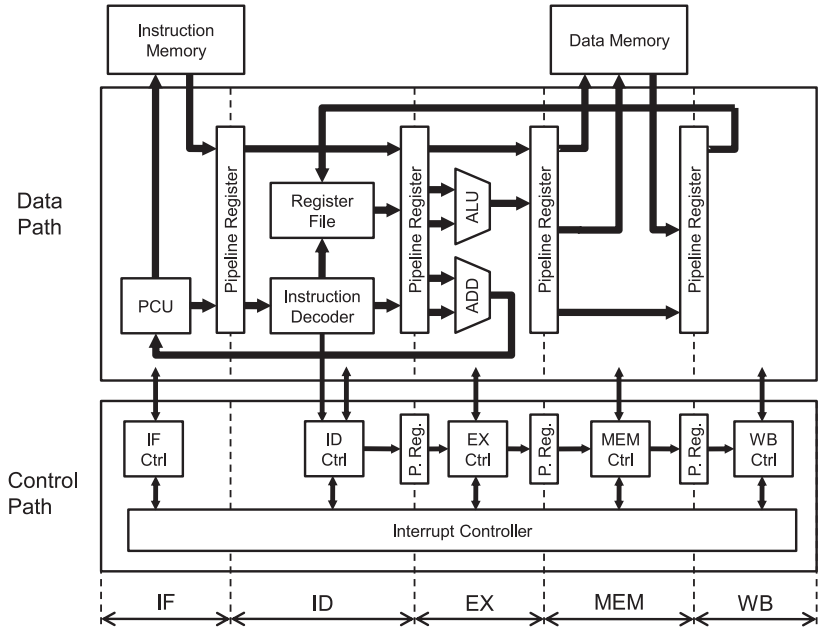


Fig. 9 Processor architecture model example with 5 stage instruction pipeline.

Meister.

Figure 9 shows an example of a structure model of a RISC processor with 5 stage pipeline, generated by ASIP Meister.

7.3 HDL Description Generation

In ASIP Meister, two types of HDLs are generated. One is for synthesis and the other is for simulation. Processor consists of a lot of hardware resources such as ALU, shifter, multiplier, register files, pipeline registers, datapath selectors and so on in the datapath and a controller in the control path.

In ASIP Meister, input description for hardware generation is written in micro-operation description, which is the specific language used in ASIP Meister to describe the instruction behavior in each pipeline stage of each instruction. On micro-operation description, processing behaviors are written in like result= COMPONENT.FUNCTION(port0, port1, $\cdots$); notations, where COMPONENT

```
micro_operation SUB{
    wire [31:0] src0;
    wire [31:0] src1;
    wire [31:0] res;
    stage 2{
      src0 = GPR.read0(rs0);
      src1 = GPR.read1(rs1);
    };
    stage 3 {
      wire [3:0] flag;
      <res, flag> = ALU.sub(src0, src1);
    };
    stage 5 {
      null = GPR.write0(rd, res);
    };
};
```

(a) SUB instruction

```
micro_operation SLLI{
    wire [31:0] src0;
    wire [31:0] res;
    stage 2{
      src0 = GPR.read0(rs0);
    };
    stage 3 {
      res = SFT.add(src0, imm);
    };
    stage 5 {
      null = GPR.write0(rd, res);
    };
};
```

(b) SLLI instruction

Fig. 10 Instruction examples of micro-operation description.

is the resource name used in instruction, FUNCTION is one of functions of used component, and portX indicates the connected signal to specified component. Micro-operation description offers some macro definition for repetitive usage of descriptions. In processors, instruction fetch which moves instruction from memory to instruction register is a common behavior between instructions and macro Fetch() are used as an example of macro in Fig. 8.

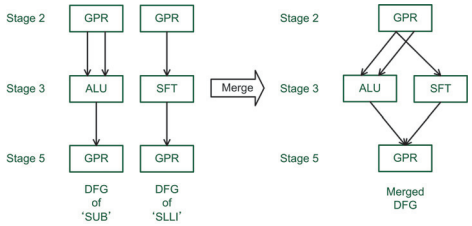
In the first HDL generation phase, micro-operation description is analyzed, and Data Flow Graph (DFG) is generated for each instruction. Second, the generated DFGs are merged into one DFG for generating processor datapath. Here, some input ports in the merged DFGs have a signal conflict where more than two edges are connected to one input port. HDL generation system in ASIP Meister inserts multiplexers as datapath selectors in order to resolve the signal conflicts. Third, pipeline registers are inserted at pipeline stage boundaries. Controllers are generated by analyzing control signal of used components in each instruction, datapath selector control, and pipeline register control which stalls the pipeline processing for structural hazards of resources. Finally, HDL description is generated from the DFG. **Figure 10** shows a micro-operation description of two instructions, subtraction (SUB) and shift left logical with immediate (SLLI) instructions. Since Fig. 10 shows the main part of the instruction, instruction fetch and operation decode part are omitted for simplification. GPR is a general purpose register, ALU is an arithmetic logic unit, and SFT is a shifter. "read" and "write" are functions of GPR which indicates reading from and writing to the GPR, and "sub" and "add" are a function of ALU and SFT, respectively. **Figure 11** (a) shows an example of merging operation of two DFGs. By analyzing the same name GPR in two instructions, generator identifies GPR in two instructions the same component. Figure 11 (b) shows an example of inserting a datapath selector. At resource GPR, both the output of ALU and the output of SFT are input of GPR. Therefore, a datapath selector should be inserted in order to resolve the signal conflict of both outputs. Figure 11 (c) shows the generated structure after HW generation, where data is transferred over the pipeline stage, pipeline register is inserted, and controller is generated.

Originally, though ASIP Meister supports simple single scalar pipeline processor, it recently extends to support VLIW type processor and low power type RISC processor. Furthermore, most recent generator supports not only synthesizable and simulatable HDLs in VHDL or Verilog HDL but also descriptions in SystemC for simulations.

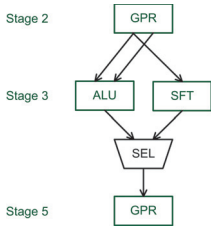
7.4 Application Program Development Tool Set Generation

Although there have been a lot of researches focusing on compiler generation for the "flexible architecture model" (as classified in Section 4) [34], it is still a challenge to automatically deliver industrial quality compilers with optimization capabilities. For this reason, the current version of ASIP Meister mainly supports tool set generation for the "fixed architecture model". Namely, while custom processors are designed by adding new instructions to a base processor, tool sets for the custom processors are auto-generated by extending an existing tool set for the base processor with an ability to handle the new instructions.

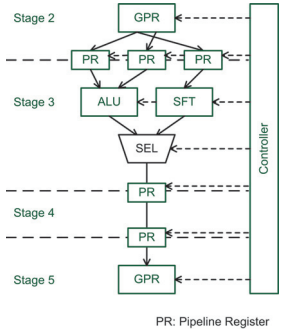
As an infrastructure of the tool set generation, GNU tools are employed, which include GCC (a compiler) [25], Binutils (binary utilities which includes an assembler, a disassembler, a linker, etc.) [35], and GDB (a debugger and a instruction set simulator) [36]. Namely we assume GCC, Binutils, and GDB are available for the base processor, from which the same tools for the extended processors are generated. Popularity of the GNU tools among embedded software developers



(a) Merging DFGs of instructions



(b) Inserting datapath selector



(c) Generated structure model

Fig. 11 Structure model generation.

is of practical importance. Generated tools will provide a good affinity with the other existing tools. Open source license is sometimes vitally important, for the tools for both the base processor and the generated processors may need to be customized to their own purposes.

```
int a, b, m;
int y = _builtin_brownie32_SADD(a,b);
_builtin_brownie32_MAC(m, a, b);
```

Fig. 12 Use of custom instructions via intrinsics.

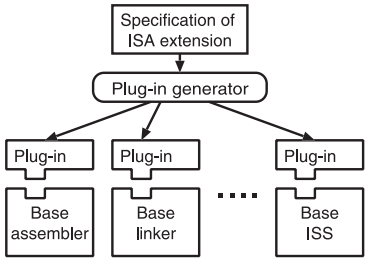


Fig. 13 Plug-in approach for augumenting tool chain for base processors.

The extended GCC provides a way of utilizing newly added instructions in terms of **intrinsics**, or built-in C functions. Each custom instruction is generated in the object file by calling a corresponding intrinsics. **Figure 12** shows an example, where the second line specifies the use of an SADD instruction (with two register reads and one register write) and the third line the use of an MAD instruction (whose first operand is read and written). The tool generator augments the base GCC with the intrinsics for the custom instructions based on the information given in the GUI.

Note that the extended GCC handles register allocation and code optimization for the custom instructions. Especially, code scheduling is conducted considering the pipeline hazards and pipeline forwarding under the optimization option (-O2).

The tools other than GCC are extended by a "plug-in" approach [37]. **Figure 13** illustrates the concept of the plug-in. Each tool for the base processors is modified to have a "plug" while the corresponding tasks to handle the custom instructions are packaged as a "plug-in" to fit into the plug. **Figure 14** shows the internal flow to implement the plug-in. For each instruction, it is tested if the instruction is a base instruction or not. If it is a new instruction the plug-in is called; otherwise the ordinary task is followed. This is a flexible scheme which allows customization of the base tools in a safe way.

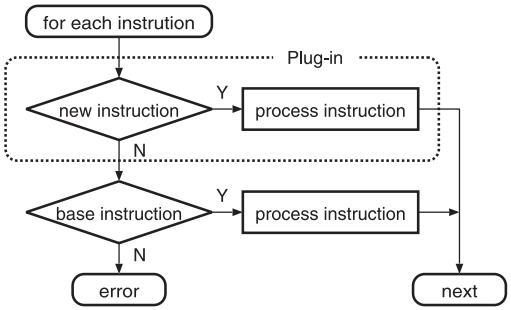


Fig. 14 Internal flow for plug-in implementation.

The machine description for extended GCC as well as plug-ins for all the other tools are generated from the specification of ISA extension, which is captured from the GUI and stored in a single XML file.

8. Case Study: An ASIP Design Example

In this section, MeDIX-I (Medical-Domain specific Instruction eXtention, type I) processor [38] is introduced as one of the successful ASIPs for bio-medical information sensing system. MeDIX-I is being developed in the collaborative research project titled "Development of Healthcare Devices and Systems for Ubiquitous Bioinstrumentation", sponsored by the City Area Program of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan. This project is being performed by Nara Medical University, Osaka University, Tokyo Institute of Technology, Kwansei Gakuin University, and Kinki University.

8.1 Background

The basic requirements to the biomedical information sensing systems for the future medical and health care are as follows. There are needs to measure inner body pressure, electric potential (voltage), acoustic pressure, etc. While this information has been measured in the hospitals, it is desirable for both medical doctor and patients to measure these information under the ordinary living environment, especially for the examination of chronic malady. In this case, following requirements are important: less awareness of the sensing system, less restraint measurement, less invasive measurement, long time continuous measurement, and real-time reporting to hospital.

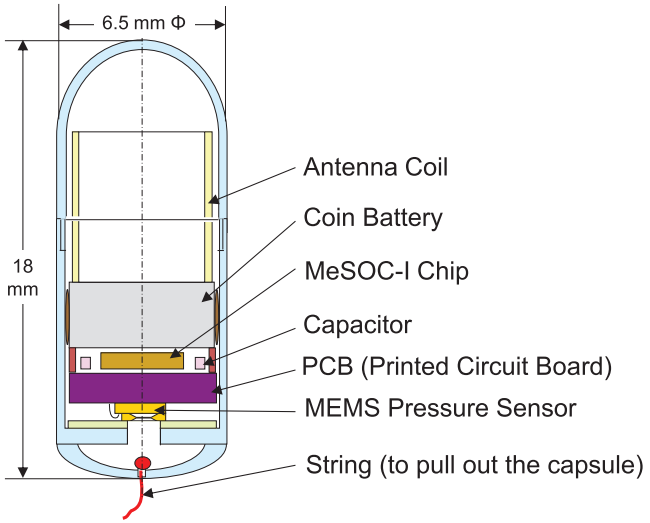


Fig. 15 Cross section of the inner body pressure sensing node.

In order to satisfy these requirements, small size, light weight and less energy consuming devices with wireless communication functions should be used.

Proposed solution to these requirements is the medical domain specific SoC device that integrates interfaces to various sensors, analog to digital convertors (ADC) [43], micro processor unit (MPU), memory (RAM and ROM), RF (radio frequency) communication circuit, as well as clock generator and power management unit. The SoC device is named MeSOC-I (Medical domain specific System On Chip, type I). MeDIX-I is the MPU used in the MeSOC-I.

The cross section of the prototype of inner body pressure sensor node is shown in **Fig. 15**. The major components of the sensing node are a MEMS pressure sensor, MeSOC-I chip in a chip size package, coin battery, antenna coil, and capacitors on a flexible printed circuit board (PCB).

8.2 Functional Requirements to MeSOC-I

The outline of the functional requirements to MeSOC-I is to measure and send the inner body pressure value to the transmitter module that locates on the

Table 1 Features of Brownie Micro 16 processor.

Feature	Value
No. of Pipeline Stages	3
Instruction Bus Width	16 bit
Data Bus Width	16 bit
Register File	16 × 16 bit
No. of Instructions	33
Gate Count	12 K(*)
Max. Clock Frequency	135 MHz(*)
Power Consumption	83 uW / MHz (*)

(*) Logic synthesis condition: TSMC 0.18 μm CMOS Analog Mixed Signal Cell Library, Typical Delay, VDD=1.8 V

surface of the body every 1/30 seconds. The value sent from the sensor node is an average of continuous 32 times measurements of the inner body pressure to reduce the measurement error due to various noises. The measurement is supposed to continue for 72 hours (three days) after the sensing node is implanted in the body.

It is system, electromagnetic induction coupling is used for the communication between inner body capsule and outer body data transmitter. Because the data transmission using electromagnetic induction coupling could be unstable, data transmission error can happen. In order to detect such error and recover whenever possible, ECC (Error Correcting Code) based on the Multi Dimensional Parity Code (MDPC) is used in the physical layer of transmission.

8.3 Extended Instructions in MeDIX-I

The base processor of MeDIX-I is the Brownie Micro 16 (BM16) processor from ASIP Solutions, Inc., which is a 16 bit small RISC processor as shown in **Table 1**. The block diagram of MeDIX-I is shown in **Fig. 16**. The photomicrograph of MeSOC-I is shown in **Fig. 17**.

Extended instructions that have been added to BM16 processor are as follows:

ECC supplementary instructions: ECC supplementary instructions include those to calculate check code and syndrome, and perform error correction. One of the advantages of MDPC is its extensibility, that is, data length to be coded/decoded can be easily extended and can be processed using the same instruction set for different length of data.

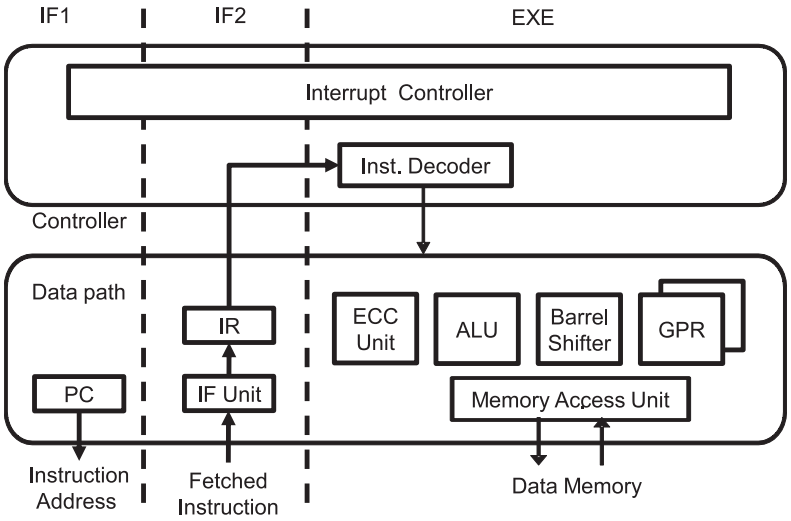


Fig. 16 Block diagram of MeDIX-I.

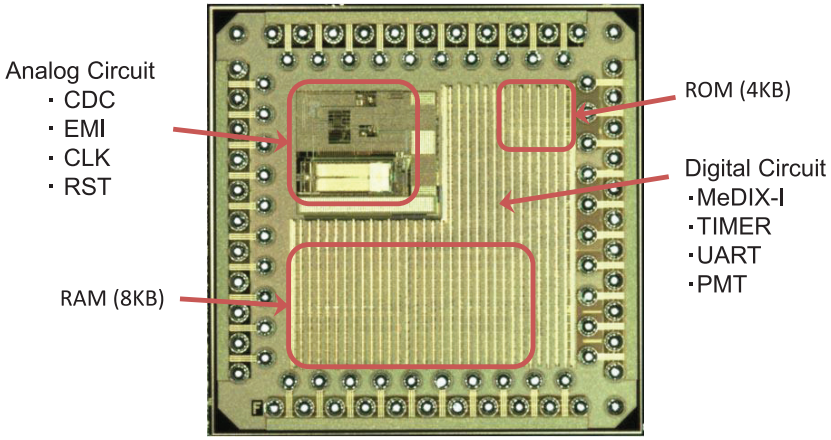


Fig. 17 Photomicrograph of MeSOC-I.

ADC control instructions: ADC (Analog to Digital Converter) control instructions generate control signals and read the converted results from ADC.

Sleep instruction: Sleep instruction has been added to save energy by freez-

Table 2 Comparison of area, performance, power and energy consumption.

	BM16 Only	BM16+ASIC	MeDIX-I
Area [mm^2]	101,508 (100%)	169,536 (+67%)	107,476 (+5.9%)
Exec. Cycles	291 (100%)	49 (−83%)	33 (−89%)
Power [μW/MHz]	34.8 (100%)	57.7 (+66%)	42.3 (+22%)
Energy [nJ]	10.13 (100%)	2.83 (−72%)	1.39 (−86%)

(*) Cell Library: 0.18 μm CMOS analog mixed digital;
Clock Freq.: 1 MHz; Data Bit Length: 64

ing clock signal when all tasks became idle. The processor awakes when external interrupt has occurred.

8.4 Effectiveness of ECC Supplementary Instructions

In order to confirm the effectiveness of MeDIX-I in energy consumption for ECC related process, two different designs have been made. One is based on Pure Software (GPP) Solution, where the base processor BM16 has been used as it is, and all ECC related function has been implemented by software using RISC instructions of BM16. While BM16 is not a so-called GPP, but most of the GPP's do not have effective instructions for ECC related functions.

Another one is based on Dedicated Hardware (ASIC) Solution, where all ECC related functions have been implemented by hardware and connected to BM16 processor as a peripheral module. In this implementation, following protocol is necessary to perform the ECC related functions: (1) check if the status of ECC module is ready, (2) send data and command to ECC module, and (3) receive the results and error status.

For each implementation, chip area was estimated from the logic synthesis result using Design Compiler from Synopsys, then the execution cycles was measured using a ModelSim logic simulator from Mentor Graphics, and finally power consumption and energy consumption were estimated using Power Compiler from Synopsys. The results are shown in **Table 2**. The energy consumption for different data length is compared in **Fig. 18**.

Comparing the experimental results of ASIP Solution with those of Pure Software Solution, the chip area of MeDIX-I is about 6% larger and power consumption is about 22% larger than BM16, but energy consumption is drastically reduced to 14% of BM16 because the execution cycles was reduced to 11% of

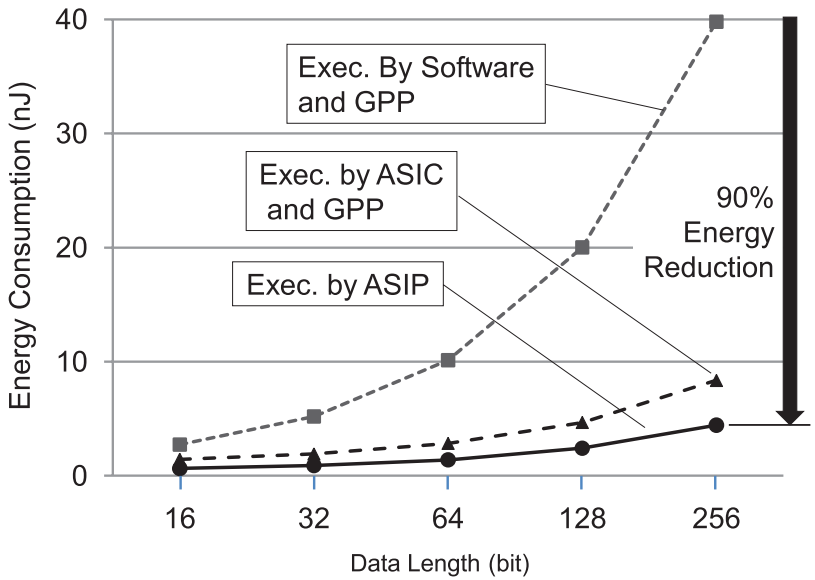


Fig. 18 Comparison of energy consumption.

BM16.

We can also find that the energy consumption of MeDIX-I is about 50% smaller than that of the Dedicated Hardware (ASIC) Solution. This result is due to the redundant hardware component in ECC module and execution cycle overhead to perform the protocol to control ECC module. In the case of MeDIX-I, processor's resources can be used to perform ECC functions, and the data item can be sent and received using general purpose registers without any I/O overhead.

9. Conclusion and Future Direction

In this paper, the advantage of ASIPs over the conventional methodologies to implement embedded systems, such as GPP centric Software Solution and/or ASIC as a peripheral is explained. Then, various technical issues to generate HDL description of ASIP and to generate application program development tool set, such as compiler and simulator, and architecture optimization methods were introduced. Next, ASIP Meister, an ASIP development environment, was in-

troduced. Then, one of successful real-world design examples is for biomedical information sensing system is introduced. In this system, almost all functions are concentrated in one SoC called Medical domain specific System On Chip, Type I (MeSOC-I). It consists of micro processor unit (MPU), analog to digital converters (ADC), memories (RAM and ROM), radio frequency communication (RF) circuits as well as clock generator and power management unit. This MPU controls all components on this SoC and it is an ASIP generated by ASIP Meister.

In system level design era, since processor is indispensable component in SoC, SoC should be expected to have at least one processor component on it. Therefore, the importance of processor design will be more and more increasing. Furthermore, recently multi-processor SoC (MPSoC) have become popular and more-than Moore era is approaching. Efficient processor design, especially ASIP, is still important research area in future.

Acknowledgments The authors would like to express their appreciation to the following researchers and engineers: Prof. Jun Sato from Tsuruoka National College of Technology, Prof. Akichika Shiomi from Shizuoka University, Prof. Akira Kitajima from Osaka Electro-Communication University, and many ex-students, especially Dr. Makiko Itoh (currently with Fujitsu Laboratories, Ltd.) Dr. Yuki Kobayashi (currently with Renesas Electronics Corp.), for their effort to develop ASIP Meister. The author would like to thank Mr. Nobuyuki Hikichi from SRA, for his kind assistance and advice to the development of GNU tool set for ASIP Meister. The authors would like to express their sincere thanks to STARC for their financial support and technical advice as industrial engineers.

References

1) *International Technology Roadmap for Semiconductors*, 2009 Edition. http://www.itrs.net/
2) Moore, G.E.: Cramming more components onto integrated circuits, *Electronics*, Vol.38, No.8 (Apr. 19, 1965).
3) http://www.sematech.org/
4) Gajski, D.D., Zhu, J., Domer, R., Aerstlauer, A. and Zhao, S.: *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers (2000).
5) http://www.systemc.org/home/
6) Mishra, P. and Dutt, N. (Eds.): *Processor Description Languages*, Elsevier (2008).
7) Barbacci, M.R., Barnes, G., Cattell, R.G. and Siewiorek, D.P.: *The ISPS Computer Description Language*, Department of Computer Science, Carnegie-Melon University (1977).
8) Barbacci, M.R.: Instruction Set Processor Specifications (ISPS): The notations and its applications, *IEEE Trans. Comput.*, Vol.C–30, No.1, pp.24–40 (Jan. 1981).
9) Marwedel, P.: The MIMOLA Design System: Detailed Description of the Software System, *Proc. 16th Design Automation Conference*, pp.59–63 (1979).
10) Marwedel, P.: MIMOLA—A Fully Synthesizable Language, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.35–64, Elsevier (2008).
11) Zimmermann, G.: The MIMOLA Design System: A Computer Aided Digital Processor Design Method, *Proc. 16th Design Automation Conference*, pp.53–58 (1979).
12) Hadjiyiannis, G., Hanono, S. and Devadas, S.: ISDL: An Instruction Set Description Language, *Proc. Design Automation Conference* (*DAC*), pp.299–302 (1997).
13) Freerics, M.: The nML machine description formalism, *Technical Report 1991/15*, Computer Science Department, T.U. Berlin, Berlin, Germany (1991).
14) Van Praet, J., Lanneer, D., Geurts, W. and Goossens, G.: nML: A Structural Modeling Language for Retargetable Compilation and ASIP Design, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.65–94, Elsevier (2008).
15) Chattpadhyay, A., Myer, H. and Leupers, R.: LISA: A Uniform ADL for Embedded Processor Modeling, Implementation, and Software Toolsuite Generation, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.95–132, Elsevier (2008).
16) Hoffmann, A., Meyr, H. and Leupers, R.: *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers (2002).
17) Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N. and Nicolau, A.: EXPRESSION: A Language for Architecture Expression through Compiler/Simulator Retargetability, *Proc. Design and Test in Europe* (*DATE*), pp.485–490 (1999).
18) Itoh, M., Higaki, S., Sato, J., Shiomi, A., Takeuchi, Y., Kitajima, A. and Imai, M.: PEAS-III: An ASIP Design Environment, *Proc. International Conference on Computer Design: VLSI in Computers and Processors* (*ICCD*), pp.430–436 (Sep. 2000).
19) Kobayashi, Y., Takeuchi, Y. and Imai, M.: ASIP Meister, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.163–182, Elsevier (2008).
20) Sato, J., Hikichi, N., Shiomi, A. and Imai, M.: Effectiveness of a HW/SW Codesign System PEAS–I in the CPU Core Design, *Proc. Asia Pacific Conference on Hardware Description Languages* (*APCHDL*), pp.259–262 (1994).
21) Imai, M., Takeuchi, Y., Ohtsuki, N. and Hikichi, N.: Compiler Generation Techniques for Embedded Processors and their Application to HW/SW Codesign, *System-Level Synthesis*, Jerraya, A.A. and Mermet, J. (Eds.), pp.293–320, Kluwer Academic Publishers (1999).
22) Sanghavi, H.A. and Andrews, N.B.: TIE: An ADL for Designing Application-specific Instruction Set Extension, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.183–216, Elsevier (2008).

23) Schliebusch, O., Meyr, H. and Leupers, R.: *Optimized Synthesis from Architecture Description Language Models*, Springer (2007).
24) Liem, C.: *Retargetable Compilers for Embedded Cores*, Kluwer Academic Publishers (1997).
25) http://gcc.gnu.org/
26) http://www.ace.nl/compiler/cosy.html
27) Atasu, K., Pozzi, L. and Lenne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints, *International Journal of Parallel Programming*, Vol.31, No.6, pp.411–428 (Dec. 2003).
28) Binh, N.N., Imai, M., Shiomi, A. and Hikichi, N.: A Hardware/Software Partitioning Algorithm for Pipelined Instruction Set Processor, *Proc. European Design Automation Conference* (*EURO-DAC*), pp.176–181 (1995).
29) Cheung, N., Henkel, J. and Parameswaran, S.: Rapid Configuration & Instruction Selection for an ASIP: A Case Study, *Proc. in Design, Automation and Test in Europe Conference and Exhibition* (*DATE '03*), pp.802–809 (2003).
30) Cong, J., Fan, Y., Han, G. and Zhang, Z.: Applicatoin-Specific Instruction Generation for Configurable Processor Architectures, *Proc. in 2004 International Symposium on Field Programmable Gate Arrays* (*FPGA'04*), pp.183–189 (Feb. 2004).
31) Peymandoust, A., Pozzi, L., Ienne, P. and De Micheli, G.: Automatic Instruction-Set Extension and Utilization for Embedded Processors, *Proc. 14th International Conference on Application-specific Systems, Architectures and Processors*, The Hague, The Netherlands, pp.108–118 (June 2003).
32) Pozzi, L., Atasu, K. and Ienne, P.: Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets, *IEEE Trans. Comput.–Aided Design of Integrated Circuits and Systems*, Vol.25, No.7, pp.1209–1229 (2006).
33) http://www.asip-solutions.com/
34) Leupers, R. and Marwedel, P.: *Regargetable Compiler Technology for Embedded Systems—Tools and Applications*, Kluwer Academic Publishers (2001).
35) http://www.gnu.org/software/binutils/
36) http://www.gnu.org/software/gdb/
37) 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 (Aug. 2010), to appear.
38) Iwato, H., Sakanushi, K., Takeuchi, Y., Imai, M., Matsuzawa, A. and Hirao, Y.: A Low Power SoC for Pressure Measurement Capsules in Ambulatory Urodynamic Monitoring, *Proc. Cool Chips XIII*, pp.441–456 (Apr. 2010).
39) Araujo, G., Rigo, S. and Azevedo, R.: Processor Design with ArchC, in Ref. 6), pp.275–294 (2008).
40) Gorjiara, B., Reshadi, M. and Gajski, D.: GNR: A Formal Language for Specification, Compilation, and Synthesis of Custom and Embedded Processors, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.329–368, Elsevier (2008).
41) Imai, M.: Synthesis of Application Specific CPU Core, *Proc. Synthesis and Simulation Meeting and International Exchange* (*SASIMI*), V–I (1989).
42) Onder, S.: ADL++: Object-Oriented Specification of Complicated Instruction Sets and Microarchitectures, *Processor Description Languages*, Mishra, P. and Dutt, N. (Eds.), pp.247–274, Elsevier (2008).
43) Vo, T.M., Kuramochi, Y., Miyahara, M., Kurashina, T. and Matsuzawa, A.: A 10-bit, 290fJ/conv. steps, 0.13mm2 , zero-static power, self-timed capacitance to digital converter, *Proc. International Conference on Solid State Devices and Materials 2009* (Oct. 2009).

(Received June 1, 2010)
(Released August 16, 2010)

(Invited by Editor-in-Chief: *Hedetoshi Onodera*)

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

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

Keishi Sakanushi received his B.E., M.E., and D.E. degrees in electrical and electronics engineering from Tokyo Institute of Technology, Tokyo, Japan, in 1997, 1999, and 2002, respectively. He has been an Assistant Professor in the Graduate School of Information Science and Technology, Osaka University, Osaka, Japan, since April 2002 when the graduate school was founded. His research is in VLSI layout design methodology and optimization, embedded system design methodology and optimization. He is a member of IEEE, IPSJ and IEICE.

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