An Integrated Design Environment for Application Specific Integrated Processor

Jun Sato†1*, Masaharu Imai†1, Tetsuya Hakata†1*, Alauddin Y. Alomary†1, and Nobuyuki Hikichi†2

†1 Dept. of Info. and Comp. Sciences
Toyohashi University of Technology
Toyohashi, 441 Japan

*1 currently with
Dept. of Electrical Engineering
Tsuruoka National College of Technology
Tsuruoka-shi, 997 Japan

†2 Software Tools and Tech. Division
Software Research Associates, Inc.
Tokyo, 102 Japan

*2 currently with
Dept. of Electrical Control
Kumamoto National College of Technology
Kikuchi-gun, 861-21 Japan

Abstract

This paper proposes a new framework for ASIP (Application Specific Integrated Processor) development. The system firstly accepts a set of example programs written in C language and their expected data as input, and profiles these programs both statically and dynamically. Then taking advantage of the profiled results, the system will decide the instruction set and hardware architectures of ASIP, and will synthesize the CPU core design of the ASIP, as well as the software development tools for the ASIP such as compiler and simulator.

Key Words: ASIP (Application Specific Integrated Processor), VLSI design environment, instruction set architecture, CPU core architecture, compiler generator, simulator generator.

1 Introduction

Due to the advance of ASIC (Application Specific Integrated Circuit) technologies, it is now possible to design a very large scale ASIP (Application Specific Integrated Processor) using high-level synthesis tools. Throughout this paper, we define “ASIP” as an application specific microprocessor which contains a CPU core, memory, and peripheral circuits. ASIPs can be very efficient when applied to a specific application such as digital signal processing, servo motor control, etc.[1]. While high-level synthesis tools are quite effective to design an ASIP, there are still problems to be solved: (1) how to decide the specification of a high performance ASIP such as instruction set and CPU core architectures, and (2) how to realize the application program development environment for the ASIP, such as compiler and simulator.

2 Previous Work

There proposed a number of high-level synthesis systems proposed in the literatures[6][7][8][9][10]. While many of these systems concentrate on generating hardware design of VLSIs but not on providing software development environment, some of the systems try to generate a part of software development tool such as compiler[9][10]. Their systems accept an application program written in Pascal as an example program, then generate a RT (Register Transfer) description and a microprogram for the target CPU. The optimization of the target CPU is performed by tuning the microprograms.

On the contrary, our system will generate a set of software development environment for an ASIP as well as hardware design of the ASIP, which includes: a C compiler, a simulator with source-level debugging facility, an assembler, and a linker[4].

3 System Organization

3.1 Outline

Our system consists of the application program analyzer, an architecture information generator, a CPU core generator, and an application program development tool generator as shown in Figure 1.
(1) **Application Program Analyzer**

Application Program Analyzer (APA) statically and dynamically profiles a set of application programs with corresponding data set[2]. The output of APA includes: data types and their access methods, execution counts of operators and functions, etc., used in the application programs.

(2) **Architecture Information Generator**

Architecture Information Generator (AIG) accepts the profiled results from APA, and decides the instruction set and hardware architectures of the ASIP. This task can be performed by tuning parameters in the architecture models for instruction set and CPU core, so that an average performance of the ASIP can be statistically maximized regarding the given application programs. The details of these architecture models will be discussed in sections 3.2 and 3.3.

(3) **CPU Core Generator**

CPU Core Generator (CCG) generates the CPU core design in the form of an I-IDL (Hardware Description Language), such as VHDL, based on the architecture information obtained by AIG. Then the actual CPU core design can be obtained by compiling the HDL description using a high-level logic synthesis tool.

(4) **Application Program Development Tool Generator**

Application Program Development Tool Generator (DTG) produces a set of software tools. This includes a C compiler, a simulator with a C source-level debugger, etc.

### 3.2 Instruction Set Model

The instruction set for an ASIP generated by the system is supposed to be a subset of the instruction set that can be generated by a C compiler. The compiler-generatable instruction set can be divided into two groups: **operators** and **functions**. The reason behind this is that it is relatively easy for a compiler to generate instructions corresponding to operators used in C language compared to generating those corresponding to functions. It is because that the full set of operators is already decided, but the full set of user-defined functions is not known a priori.

We further divide the set of operators into two subgroups: **primitive operators** and **basic operators**. The set of primitive operators is chosen so that other basic operators can be substituted by a combination of primitive operators.

This classification has a tight relation to that of the intermediate language used to construct the compiler described in section 3.4. This concept is very important to generate architectures and program development tools automatically. Thus, we have divided the instruction set into three classes as follows.

(1) **Primitive Instruction Set**

The Primitive Instruction Set (PIS) can be realized by a minimal hardware component as ALU and shifter. The instruction set of the ASIP includes all instructions in PIS.

(2) **Basic Instruction Set**

The Basic Instruction Set (BIS) includes the set of operators used in C language except those included in PIS. Instructions included in BIS can be implemented by hardware modules or firmware.

(3) **Extended Instruction Set**

The Extended Instruction Set (XIS) includes instructions which correspond to library functions or user-defined functions. The instructions in XIS could be implemented by using complex hardware modules, such as coprocessor or firmware.

### 3.3 CPU Architecture Model

The CPU architecture of an ASIP generated by the system is based on the GCC's abstract machine model. In this model, various parameters, such as storage amount and layout, register count and usage, stack layout, addressing mode, floating-point format, etc., can be modified easily. The outline of the current CPU architecture features is as follows. The parameters in the architecture model were chosen as in the following so that the target CPU could be generated easily. However, part of these parameters are subject to modification in the future version of IDEAS.

(1) **Instruction format**

All instructions are 32 bits. Almost all instructions have three operands.

(2) **Storage organization**

The memory is assumed to be byte-aligned and linearly addressable. The storage space can be extended up to 4 GB.

(3) **Register organization**

The CPU contains three sets of registers: constant registers such as the zero-register, general-purpose registers, and special-purpose registers such as floating-point registers. Register count of each register set can be modified.

### 3.4 Application Program Development Tools

The application program development tools generated by the system includes a C compiler, a simulator, an assembler, a linker, and so on. The C compiler can be generated by taking advantage of the GCC. And a simulator with a source-level debugger can also be generated by using the GNU source-level debugger (GDB). Other utilities can be implemented easily. A prototype of the compiler generator has been developed using GCC. Where the intermediate language "RTL" (Register Transfer Language) of the GCC is used as the full set of instructions. Then, we have classified the RTL into Primitive Simple RTL (PRTL), Simple RTL (SRTL) and Extended Simple RTL (XRTL), corresponding to the instruction set classification described in section 3.2.

The instructions included in PRTL, SRTL and XRTL are shown in Table. 1. The CPU core generated by the system have all of the PRTL, but will have parts of SRTL and XRTL, as its instruction set.
4 Discussion

There are at least two different approaches to design a CPU core included in an ASIP. One is to use a pre-designed general purpose CPU or DSP core as the CPU core. And the other is to synthesize an application specific CPU core.

In the first approach, total design period could be reduced, because the CPU core itself is already designed and software development environment such as compiler and debugger would be available. However, it is generally difficult to find the optimum CPU core for a given application, because the pre-designed CPU core has a fixed instruction set. Other drawbacks of this approach would be the copyright, possible licence fee, and the fixed fabrication process.

In the second approach, it is possible to choose the instruction set so that the performance of the ASIP will be maximized for the given application. According to the preliminary study, it was found that the most frequently executed instructions differ from one application to another[2]. This result suggests that the performance of an ASIP can be maximized by choosing an optimal instruction set for a specific application under appropriate constraints such as chip area and power consumption. In this approach, however, an application program development environment should be prepared as well as hardware design tools. The system proposed in this paper is expected to be a solution of this problem. We believe that the key to generating an efficient ASIP is to overcome the "semantic gap" between the hardware architecture and the language compiler.

5 Conclusion

In this paper, an application specific integrated processor design environment has been proposed. Part of the system such as Application Program Analyzer (APA) and part of Application Program Development Tool Generator (DTG) have been already developed[2][3]. Prototypes of Architecture Information Generator (AIG), CPU Core Generator and other tools are under development.

Acknowledgments

Authors would like to express their thanks to the members of the VLSI Design Laboratory of Toyohashi University of Technology for their helpful comments.

References


Table 1: Instructions including PRTL, SRTL and XRTL

<table>
<thead>
<tr>
<th>class</th>
<th>category</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRTL</td>
<td>arithmetic</td>
<td>add, sub, and, or, xor, one, cmp, neg, ashl, ashr, lshr</td>
</tr>
<tr>
<td></td>
<td>transfer</td>
<td>mov</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td>jmp, beq, nop</td>
</tr>
<tr>
<td>SRTL</td>
<td>arithmetic</td>
<td>mul, umul, div, mod, udiv, umod, cmp, cmpstr, tst, trunc, extend, zero, extend, float, floats, trunc, fix, fixuns, fix.trunc, fixuns.trunc</td>
</tr>
<tr>
<td></td>
<td>transfer</td>
<td>mov, strict, movstr, (mov)</td>
</tr>
<tr>
<td></td>
<td>control</td>
<td>call, call, return, bne, bgt, bgtu, bte, bltu, bge, bgeu, ble, bleu, cases, table, jump</td>
</tr>
<tr>
<td></td>
<td>others</td>
<td>ext, extax, inv, seq, sge, sgt, agu, slt, situ, sge, sgeu, sle, sltu</td>
</tr>
<tr>
<td>XRTL</td>
<td></td>
<td>abs, sqrt, if, sin, cos, tan, etc., user-defined functions</td>
</tr>
</tbody>
</table>
Figure 1: System Configuration